# Accelerating Genome Sequence Analysis via Efficient Hardware/Algorithm Co-Design

*Submitted in partial fulfillment of the requirements*
*for the degree of Doctor of Philosophy*
*in Electrical and Computer Engineering*

**Damla Senol Cali**

B.S., Computer Engineering, Bilkent University
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

**August 2021**

*To my parents - Mine and Sinan,*

*my sister - Irmak,*

*my husband - Tunca.*

# Acknowledgments

I have many people to thank who have supported me in very different ways during my Ph.D. journey over the last six years. First and foremost, I am very grateful for my advisors, Prof. Onur Mutlu and Prof. Saugata Ghose. Onur has generously provided many opportunities, offered guidance, mentorship, and patience throughout my Ph.D., which have been key to my growth and success. His passion for excellence in all aspects of research has enabled me to push my boundaries and be a part of several top-notch research projects. I have learned greatly from his feedback and insights during the numerous iterations we made for the paper submissions, video recordings, and conference talks. In addition to the high standards he set for research, the collaborative environment he has offered within the SAFARI Research Group and also with our industry partners has helped me to work with a diverse group of people all around the world and exposed me to many different research directions.

I am very grateful to Saugata for being a great mentor and offering help whenever I need it. During my all ups and downs throughout my Ph.D. journey, he has supported me with great patience and guidance. His tremendous amount of help and support during paper submissions has helped me to stay motivated, positive, and encouraged despite the challenging and stressful deadlines. In addition to the calming and encouraging environment he has provided, his passion for teaching has been inspirational.

I am very grateful to Can Alkan, my dissertation committee member and my mentor since my undergrad. His passion and vision for genomics have been one of the key enablers of all the research I have conducted during my Ph.D.

I am also grateful to my dissertation committee member, James Hoe, for his valuable feedback and for always letting me think about the big picture of my research.

This long and challenging journey would have been impossible without the great support and friendship I have had in my research group. Despite the thousands of miles we have between us, I have never felt distant or lonely. First, I want to thank Jeremie

Kim, my very first friend at SAFARI, who helped me to survive my first year. I am very grateful for his support and the contributions he made to my research throughout my Ph.D. I am very thankful for Zulal Bingol, an endless source of support and positivity, a friend and colleague whom I can trust and count on whenever I need the most. I am very thankful for dear Nastaran Hajinazar, who always makes me smile and brings joy and happiness to everyone around her. I am very thankful for Can Firtina, who is a great friend since my undergrad and a very helpful colleague during my Ph.D. I want to thank Giray Yaglikci for his friendship and our endless chats in the office. I want to thank Gagandeep Singh for providing a fun working environment and for our TV series discussions. I want to thank Amirali Boroumand for being a great friend and keeping me sane during my Ph.D. I want to thank Konstantinos Kanellopoulos for our long hours of brainstorming and bringing an immense wealth of knowledge. I want to thank Geraldo Francisco Oliveira for his friendship and delicious cake recipes. I want to thank Nour Almadhoun Alserr for her kindness and support during the final year of my Ph.D. I also want to thank Nika Mansourighiasi for her friendship and our helpful research discussions. I am tremendously lucky to have all these invaluable lifelong friendships, and I am immensely grateful to all of them.

I am also very grateful to Lavanya Subramanian, Rachata Ausavarungnirun, Mohammed Alser, and Juan Gómez-Luna for their mentorship. They have provided great insights and support throughout my Ph.D. journey with their technical expertise, many fruitful discussions, and their patience and kindness towards me.

I want to acknowledge all the other members of our research group at CMU and ETH for being both great friends and colleagues: Minesh Patel, Minh Sy Quang Truong, Banu Cavlak, Joel Lindegger, Ziyi Zuo, Jisung Park, Hasan Hassan, Max Rumpf, Haiyu Mao, Aditya Manglik, Sam Cheung, Akanksha Baranwal, Nandita Vijaykumar, Kevin Hsieh, Donghyuk Lee, Hongyi Xin, Yixin Luo, Vivek Seshadri, Kevin Chang, and all other past and current SAFARI, ARCANA, and Bilkent CompGen members for their discussions, feedback, collaboration, and support.

I would also like to thank my internship mentors and managers, Sree Subramoney and Gurpreet S. Kalsi, who provided a stimulating environment and an industrial perspective on hardware acceleration during my internships at Intel Labs. I sincerely thank Intel for these opportunities.

I gratefully acknowledge the generous support from Intel, Google, Microsoft, Samsung, VMware, and other industrial partners of SAFARI Research Group, the Semiconductor Research Corporation, and the National Institutes of Health (grant HG006004).

Finally, I want to thank my family and friends for their love and support during this long journey. Especially, I am very grateful to my parents, Mine and Sinan. I cannot thank my mom enough for her endless support, encouragement, unconditional love, and sacrifice. She has taught me how to be a very strong woman. I cannot thank my dad enough for being a wonderful role model with his ambition, determination, high standards for success, and the inspiration to pursue my dreams. I am very grateful for his endless support and love. I am very grateful to my sister, Irmak, for being my best friend, an endless source of joy, laughter, and love. I cannot thank my sister enough for making me smile even on my hardest days and always being by my side. I am very grateful to my husband, Tunca, for always believing in me and providing endless care and love. Despite all my highs and downs, with his unwavering support, understanding, and encouragement, I have always found my way. Without the comfortable, joyful, and loving environment he has provided, I could not have accomplished any of my achievements. It would have been impossible for me to become who I am today without my parents, my sister, and my husband, and I will forever be indebted and grateful to them. This dissertation would in no way be possible without them, so it is dedicated to my loving family: Mine, Sinan, Irmak, and Tunca.

# Abstract

*Genome sequence analysis plays a pivotal role in enabling many medical and scientific advancements in personalized medicine, outbreak tracing, the understanding of evolution, and forensics. Modern genome sequencing machines can rapidly generate massive amounts of genomics data at low cost. However, the analysis of genome sequencing data is currently bottlenecked by the computational power and memory bandwidth limitations of existing systems, as many of the steps in genome sequence analysis must process a large amount of data. Moreover, as sequencing technologies advance, the growth in the rate that sequencing devices generate genomics data is far outpacing the corresponding growth in computational power, placing greater pressure on these bottlenecks.*

*Our goals in this dissertation are to (1) understand where the current tools and algorithms do not perform well in order to develop better tools and algorithms, and (2) understand the limitations of existing hardware systems when running these tools and algorithms in order to design efficient customized accelerators. Towards this end, we propose four major works, where we characterize the real-system behavior of the genome sequence analysis pipeline and its associated tools, expose the bottlenecks and tradeoffs of the pipeline and tools, and co-design fast and efficient algorithms along with scalable and energy-efficient customized hardware accelerators for the key pipeline bottlenecks to enable faster genome sequence analysis.*

*First, we comprehensively analyze the tools in the genome assembly pipeline for long reads in multiple dimensions (i.e., accuracy, performance, memory usage, and scalability), uncovering bottlenecks and tradeoffs that different combinations of tools and different underlying systems lead to. We show that we need high-performance, memory-efficient, low-power, and scalable designs for genome sequence analysis in order to exploit the advantages that genome sequencing provides. Second, we propose GenASM, an acceleration framework that builds upon bitvector-based approximate string matching (ASM) to accelerate multiple steps of the genome sequence analysis pipeline. We co-design our highly-parallel, scalable and memory-efficient algorithms with low-power and area-efficient hardware accelerators. We*

*evaluate GenASM for three different use cases of ASM in genome sequence analysis and show that GenASM is significantly faster and more power- and area-efficient than state-of-the-art software and hardware tools for each of these use cases. Third, we implement an FPGA-based prototype for GenASM, where state-of-the-art 3D-stacked memory (HBM2) offers high memory bandwidth and FPGA resources offer high parallelism by instantiating multiple copies of the GenASM accelerators. Fourth, we propose SeGraM, the first hardware acceleration framework for sequence-to-graph mapping and alignment. Instead of representing the reference genome as a single linear DNA sequence, genome graphs provide a better representation of the diversity among populations by encoding variations across individuals in a graph data structure, avoiding a bias towards any one reference. SeGraM enables the efficient mapping of a sequenced genome to a graph-based reference, providing more comprehensive and accurate genome sequence analysis. For SeGraM, we co-design algorithms and accelerators for memory-efficient minimizer-based seeding and bitvector-based, highly-parallel sequence-to-graph alignment. Compared to state-of-the-art software tools for sequence-to-graph mapping and alignment, we show that SeGraM significantly increases the throughput and reduces the power consumption for both short and long reads.*

*Overall, we demonstrate that genome sequence analysis can be accelerated by co-designing scalable and energy-efficient customized accelerators along with efficient algorithms for the key steps of genome sequence analysis. We also hope that this dissertation inspires future work in co-designing algorithms and hardware together to create powerful frameworks that accelerate other genomics workloads and emerging applications.*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Problem and Dissertation Statement

Genome sequencing, which determines the DNA sequence of an organism, plays a pivotal role in enabling many medical and scientific advancements in personalized medicine[14, 92, 105, 54, 32], evolutionary theory [82, 254, 255], and forensics [325, 39, 25]. Modern genome sequencing machines [280, 210, 241, 242, 245, 135, 136, 137] can rapidly generate massive amounts of genomics data at low cost [281, 16, 212, 20], but are unable to extract an organism's complete DNA in one piece. Instead, these machines extract smaller random fragments of the original DNA sequence, known as *reads*. These reads then pass through a computational process known as *genome sequence analysis*. However, the analysis of genome sequencing data is currently bottlenecked by the computational power and memory bandwidth limitations of existing systems, as many of the steps in genome sequence analysis must process a large amount of data. Moreover, as sequencing technologies advance, the growth in the rate that sequencing devices generate genomics data is far outpacing the corresponding growth in computational power, placing greater pressure on these bottlenecks.

Our goals in this dissertation are to (1) understand where the current tools and algorithms do not perform well in order to develop better tools and algorithms, and (2)

understand the limitations of existing hardware systems when running these tools and algorithms in order to design efficient customized accelerators. Towards this end, we propose four major works. First, we comprehensively analyze the tools in the genome assembly pipeline for long reads in multiple dimensions (i.e., accuracy, performance, memory usage, and scalability), uncovering bottlenecks and tradeoffs that different combinations of tools and different underlying systems lead to. We show that we need high-performance, memory-efficient, low-power, and scalable designs for genome sequence analysis in order to exploit the advantages that genome sequencing provides. Second, we propose GenASM, an acceleration framework that builds upon bitvector-based approximate string matching (ASM) to accelerate multiple steps of the genome sequence analysis pipeline. We co-design our highly-parallel, scalable and memory-efficient algorithms with low-power and area-efficient hardware accelerators. We evaluate GenASM for three different use cases of ASM in genome sequence analysis and show that GenASM is significantly faster and more power- and area-efficient than state-of-the-art software and hardware tools for each of these use cases. Third, we implement an FPGA-based prototype for GenASM, where state-of-the-art 3D-stacked memory (HBM2) offers high memory bandwidth and FPGA resources offer high parallelism by instantiating multiple copies of the GenASM accelerators. Fourth, we propose SeGraM, the first hardware acceleration framework for sequence-to-graph mapping and alignment. Instead of representing the reference genome as a single linear DNA sequence, genome graphs provide a better representation of the diversity among populations by encoding variations across individuals in a graph data structure, avoiding a bias towards any one reference. SeGraM enables the efficient mapping of a sequenced genome to a graph-based reference, providing more comprehensive and accurate genome sequence analysis. For SeGraM, we co-design algorithms and accelerators for memory-efficient minimizer-based seeding and bitvector-based, highly-parallel sequence-to-graph alignment. Compared to state-of-the-art software tools for sequence-to-graph mapping and alignment, we show that SeGraM significantly increases the throughput and reduces the power consumption for both short and long reads.

Our dissertation statement is as follows: ***Genome sequence analysis can be accelerated by co-designing fast and efficient algorithms along with scalable and energy-efficient customized hardware accelerators for the key bottleneck steps of the pipeline.*** s

## 1.2 Overview of Our Approach

In line with our dissertation statement, we present four works, where we **characterize** the real-system behavior of the genome sequence analysis pipeline and its associated tools, **expose** the bottlenecks and tradeoffs of the pipeline and tools, and **co-design** fast and efficient algorithms along with scalable and energy-efficient customized hardware accelerators for the key pipeline bottlenecks to enable faster genome sequence analysis.

In our first work, we present the first work that analyzes state-of-the-art tools associated with each step of the genome assembly pipeline using long reads. We analyze the tools in multiple dimensions that are important for both developers and users/practitioners: accuracy, performance, memory usage and scalability. We reveal new bottlenecks and tradeoffs that different combinations of tools and different underlying systems lead to, based on our extensive experimental analyses. We also provide guidelines for both practitioners, such that they can determine the appropriate tools and tool combinations that can satisfy their goals, and tool developers, such that they can make design choices to improve current and future tools.

In our second work, we propose GenASM, the first approximate string matching (ASM) acceleration framework for genome sequence analysis. GenASM performs bitvector-based ASM, which can efficiently accelerate multiple steps of genome sequence analysis. We modify the underlying ASM algorithm (Bitap) to significantly increase its parallelism and reduce its memory footprint. Using this modified algorithm, we design the first hardware accelerator for Bitap. Our hardware accelerator consists of specialized systolic-array-based compute units and on-chip SRAMs that are designed to match the rate of computation with

memory capacity and bandwidth, resulting in an efficient design whose performance scales linearly as we increase the number of compute units working in parallel. We demonstrate that GenASM provides significant performance and power benefits for three different use cases in genome sequence analysis. First, GenASM accelerates read alignment for both long reads and short reads. For long reads, GenASM outperforms state-of-the-art software and hardware accelerators by 116× and 3.9×, respectively, while reducing power consumption by 37× and 2.7×. For short reads, GenASM outperforms state-of-the-art software and hardware accelerators by 111× and 1.9×. Second, GenASM accelerates pre-alignment filtering for short reads, with 3.7× the performance of a state-of-the-art pre-alignment filter, while reducing power consumption by 1.7× and significantly improving the filtering accuracy. Third, GenASM accelerates edit distance calculation, with 22–12501× and 9.3–400× speedups over the state-of-the-art software library and FPGA-based accelerator, respectively, while reducing power consumption by 548–582× and 67×. We also briefly discuss four other use cases that can benefit from GenASM.

In our third work, we propose BitMAc, which is an FPGA-based prototype for GenASM. In BitMAc, we map our GenASM algorithms on Stratix 10 MX FPGA with a state-of-the-art 3D-stacked memory (HBM2), where HBM2 offers high memory bandwidth and FPGA resources offer high parallelism by instantiating multiple copies of the GenASM accelerators. After re-modifying the GenASM algorithms for a better mapping to existing FPGA resources, we show that BitMAc provides 64% logic utilization and 90% on-chip memory utilization, while having 48.9 W of total power consumption. We compare BitMAc with state-of-the-art CPU-based and GPU-based read alignment tools. Compared to the alignment steps of the CPU-based read mappers, (1) for long reads, BitMAc provides 761× and 136× speedup, while reducing power consumption by 1.9× and 2.0×, and (2) for short reads, BitMAc provides 92× and 130× speedup, while reducing power consumption by 2.2× and 2.0×. We also show that BitMAc provides significant speedup compared to the GPU-based baseline, while reducing the power consumption.

In our fourth work, we propose SeGraM, the first hardware acceleration framework for sequence-to-graph mapping and alignment. Reference genomes are conventionally represented as a linear sequence. However, this linear representation of the reference genome results with ignoring the variations that exist in a population (i.e., genetic diversity) and introducing biases for the downstream analysis. To address these limitations, recently, graph-based representations of the genomes (i.e., *genome graphs*) have gained attention. As shown in many prior works [18, 301, 279, 95, 166, 21, 111, 96, 36, 217, 164, 175, 157, 158], sequence-to-sequence mapping is one of the major bottlenecks of the genome sequence analysis pipeline and need to be accelerated using specialized hardware. Since graph-representation of the genome is much more complex than the linear representation, sequence-to-graph mapping is placing a greater pressure on this bottleneck. Thus, in this work, our goal is to design a high-performance, scalable, power- and area-efficient hardware accelerator for sequence-to-graph mapping that support both short and long reads. We base SeGraM on a memory-efficient minimizer-based seeding algorithm and a bitvector-based, highly-parallel sequence-to-graph alignment algorithm. We *co-design* both of our algorithms with high-performance, area- and power-efficient hardware accelerators. SeGraM consists of two components: (1) MinSeed, which provides hardware support to execute our minimizer-based seeding algorithm, and (2) BitAlign, which provides hardware support to execute our bitvector-based sequence-to-graph alignment algorithm. For sequence-to-graph mapping with long reads, we find that SeGraM achieves 8.8× and 7.3× speedup over 12-thread execution of state-of-the-art sequence-to-graph mapping tools (GraphAligner and vg, respectively), while reducing power consumption by 4.9× and 6.5×. For sequence-to-graph mapping with short reads, we find that SeGraM achieves 168× and 726× speedup over 12-thread execution of GraphAligner and vg, respectively, while reducing power consumption by 4.7× and 4.9×. For sequence-to-graph alignment, we show that BitAlign provides 41×–539× speedup over PaSGAL, a state-of-the-art sequence-to-graph alignment tool.

## 1.3 Contributions

This dissertation makes the following **key contributions**:

1. We present the first work that analyzes state-of-the-art tools associated with each step of the genome assembly pipeline using long reads. For the 5 different steps of the pipeline, we analyze 12 different tools and make 21 observations for these tools.

   (a) We analyze the tools in multiple dimensions that are important for both developers and users/practitioners: accuracy, performance, memory usage and scalability.

   (b) We reveal new bottlenecks and tradeoffs that different combinations of tools and different underlying systems lead to, based on our extensive experimental analyses.

   (c) We show that basecalling is the most important step of the pipeline to overcome the high error rates of nanopore sequencing technology.

   (d) We show that there is a tradeoff between accuracy and performance when choosing the tool for the assembly step. Miniasm, coupled with an additional polishing step can lead to faster overall assembly than using Canu itself, while producing high-quality assemblies.

   (e) We make observations that can guide researchers and practitioners in making conscious and effective choices for each step of the genome assembly pipeline using long reads. Also, with the help of bottlenecks we find, developers can improve the current tools or build new ones that are both accurate and fast, in order to overcome the high error rates of the long read sequencing technologies.

   (f) We show that we need high-performance, memory-efficient, low-power, and scalable designs for genome sequence analysis in order to exploit the advantages that genome sequencing provides.

2. We present GenASM, a novel approximate string matching acceleration framework for genome sequence analysis. GenASM is a power- and area-efficient hardware implementation of our new Bitap-based algorithms. GenASM is a fast, efficient, and flexible framework for both short and long reads, which can be used to accelerate *multiple steps* of the genome sequence analysis pipeline.

    (a) To avoid implementing more complex hardware for the dynamic programming based algorithm [86, 158, 301, 117, 35, 155, 267, 53], we base GenASM upon the *Bitap* algorithm [34, 317]. Bitap uses only fast and simple bitwise operations to perform approximate string matching, making it amenable to efficient hardware acceleration. To our knowledge, GenASM is the first work that enhances and accelerates Bitap.

    (b) We modify Bitap to add efficient support for long reads and enable parallelism within each ASM operation. We also propose the *first* Bitap-compatible traceback algorithm. We open source our software implementations of the GenASM algorithms [268].

    (c) In GenASM, we *co-design* our modified Bitap algorithm and our new Bitap-compatible *traceback* algorithm with an area- and power-efficient hardware accelerator. Our hardware accelerator (1) balances the compute resources with available memory capacity and bandwidth per compute unit to avoid wasting resources, (2) achieves high performance and power efficiency by using specialized compute units that we design to exploit data locality, and (3) scales linearly in performance with the number of parallel compute units that we add to the system.

    (d) We show that GenASM can accelerate *three use cases* of approximate string matching (ASM) in genome sequence analysis (i.e., read alignment, pre-alignment filtering, edit distance calculation).

(e) We find that GenASM is greatly faster and more power-efficient for all three use cases than state-of-the-art software and hardware baselines.

3. We propose BitMAc, where we leverage a modern FPGA with high-bandwidth memory (HBM) for presenting an FPGA-based prototype for our GenASM accelerators. In BitMAc, we map GenASM on Stratix 10 MX FPGA with a state-of-the-art 3D-stacked memory (HBM2), where HBM2 offers high memory bandwidth. We exploit intra-level parallelism by instantiating multiple processing elements (PEs) for the DC execution, and inter-level parallelism by running multiple independent GenASM executions in parallel.

   (a) We implement our DC and TB accelerator datapaths using SystemVerilog and incorporate the M20Ks and the HBM2 interface for both top and bottom HBM2 stacks using M20K and HBM2 IPs. After re-modifying the GenASM algorithms for a better mapping to existing FPGA resources, the final and complete BitMAc design has 4 BitMAc accelerators connected to each pseudo-channel (128 in total), where each BitMAc accelerator contains a DC accelerator with 16 PEs, a TB accelerator, an FSM, and 13.2KB of M20Ks. We synthesize and place & route the complete BitMAc design clocked at 200 MHz.

   (b) We show that BitMAc provides 64% logic utilization and 90% on-chip memory utilization, while having 48.9 W of total power consumption.

   (c) We compare BitMAc with state-of-the-art CPU-based and GPU-based read alignment tools and show that BitMAc provides significant speedup compared to the the baselines, while reducing the power consumption.

   (d) We show that due to the simplicity of the GenASM algorithms, BitMAc is a low-cost and scalable solution for bitvector-based sequence alignment.

4. We propose SeGraM, a hardware acceleration framework for sequence-to-graph mapping and alignment. SeGraM targets both the seeding and alignment steps of sequence-to-graph mapping, with support for both short (e.g., Illumina) and long

(e.g., PacBio, ONT) read sequencing technologies. For seeding, we base SeGraM on a memory-efficient minimizer-based seeding algorithm, and for alignment, we develop a new bitvector-based, highly-parallel sequence-to-graph alignment algorithm. We *co-design* both of our algorithms with high-performance, area- and power-efficient hardware accelerators.

(a) To our knowledge, SeGraM is the first acceleration framework for sequence-to-graph mapping and alignment. SeGraM aims to alleviate existing performance bottlenecks for both short and long read analysis.

(b) We propose MinSeed, the first hardware accelerator for minimizer-based seeding. MinSeed can be used for the seeding steps of both graph-based mapping and traditional sequence-to-sequence mapping.

(c) We propose BitAlign, the first hardware accelerator for sequence-to-graph alignment. BitAlign is based upon a new bitvector-based sequence-to-graph alignment algorithm that we develop, and can be also used as a sequence-to-sequence aligner.

(d) We couple SeGraM with high-bandwidth memory (HBM) to enable more effective data movement, exploit the high internal bandwidth, and improve the overall performance and energy efficiency.

(e) We evaluate SeGraM using a combination of accelerator synthesis and detailed performance modeling. We find that SeGraM is significantly more efficient than state-of-the-art sequence-to-graph mapping and sequence-to-graph alignment tools.

## 1.4 Dissertation Outline

This dissertation is organized into 8 chapters. Chapter 2 describes necessary background on genome sequencing, sequencing technologies, genome sequence analysis, and

genome graphs. Chapter 3 presents our experimental study of the genome assembly pipeline using long reads. Chapter 4 presents GenASM, a high-performance and low-power approximate string matching acceleration framework for genome sequence analysis. Chapter 5 presents BitMAc, an FPGA-based near-memory prototype of the GenASM accelerators. Chapter 6 presents SeGraM, the first hardware acceleration framework for sequence-to-graph mapping. Chapter 7 presents the expected long-term impact of the works in this dissertation and more generally, accelerating genome sequence analysis. Finally, Chapter 8 presents conclusions and future research directions that are enabled by this dissertation.

# Chapter 2

# Background

We describe the necessary background on genome sequencing, genome sequence analysis, genome assembly and read mapping pipelines, genome graphs, and sequence-to-graph mapping to help the reader to understand our observations and proposed designs for accelerating genome sequence analysis.

## 2.1 Genome Sequencing

Genome sequencing, which determines the DNA sequence of an organism, plays a pivotal role in enabling many medical and scientific advancements in personalized medicine [14, 92, 105, 54, 32], evolutionary theory [82, 254, 255], and forensics [325, 39, 25]. Modern genome sequencing machines [280, 210, 241, 242, 245, 135, 136, 137] can rapidly generate massive amounts of genomics data at low cost [281, 16, 212, 20], but are unable to extract an organism's complete DNA in one piece. Instead, these machines extract smaller random fragments of the original DNA sequence, known as *reads*.

State-of-the-art sequencing machines produce broadly one of two kinds of reads. *Short reads* (consisting of no more than a few hundred DNA base pairs [50, 295]) are generated using short-read sequencing (SRS) technologies [263, 303], which have been on the market for more than a decade. Because each read fragment is so short compared to the

entire DNA (e.g., a human's DNA consists of over 3 billion base pairs [306]), short reads incur a number of reproducibility (e.g., non-deterministic mapping) and computational challenges [90, 322, 321, 15, 296, 20, 18, 323, 212]. *Long reads* (consisting of thousands to millions of DNA base pairs) are generated using long-read sequencing (LRS) technologies, of which Oxford Nanopore Technologies' (ONT) nanopore sequencing [280, 198, 200, 56, 70, 205, 42, 176, 147, 159, 151, 258] and Pacific Biosciences' (PacBio) single-molecule real-time (SMRT) sequencing [83, 266, 264, 313, 219, 304, 202, 26] are the most widely used ones. LRS technologies are relatively new, and they avoid many of the challenges faced by short reads.

LRS technologies have three key advantages compared to SRS technologies. First, LRS devices can generate very long reads, which (1) reduces the non-deterministic mapping problem faced by short reads, as long reads are significantly more likely to be unique and therefore have fewer potential mapping locations in the reference genome; and (2) span larger parts of the repeated or complex regions of a genome, enabling detection of genetic variations that might exist in these regions [304]. Second, LRS devices perform real-time sequencing, and can enable concurrent sequencing and analysis [257, 266, 193]. Third, ONT's pocket-sized device (MinION [210]) provides portability, making sequencing possible at remote places using laptops or mobile devices. This enables a number of new applications, such as rapid infection diagnosis and outbreak tracing (e.g., COVID-19, Ebola, Zika, swine flu [257, 316, 122, 153, 61, 113, 307, 84]). Unfortunately, LRS devices are much more error-prone in sequencing (with a typical error rate of 10–15% [151, 312, 30, 304]) compared to SRS devices (typically 0.1% [106, 256, 108]), which leads to new computational challenges [280].

## 2.2 Genome Sequence Analysis

Since the whole genome of most organisms cannot be sequenced all at once, the genome is broken into smaller fragments. After each fragment is sequenced, small pieces of DNA

sequences (i.e., *reads*) are generated. The locations of the sample fragments on the whole genome are usually random. Thus, the sequences of DNA fragments (i.e., *reads*) should pass through computational mechanisms to gather meaningful information out of them, which is called *genome sequence analysis*.

There are two types of genome sequence analysis mechanisms: (1) assemble the reads without a template reference sequence (i.e., *de novo assembly*), and (2) map the reads with respect to a reference sequence (i.e., *read mapping*).

## 2.3 Genome Assembly Pipeline Using Long Reads

Figure 2-1 shows each step of the genome assembly pipeline using long reads. The output of nanopore sequencers is raw signal data that represents changes in electric current when a DNA strand passes through nanopore. Thus, the pipeline starts with the raw signal data. The first step, *basecalling*, translates this raw signal output of MinION into bases (A, C, G, T) to generate DNA reads. It is important to note that basecalling

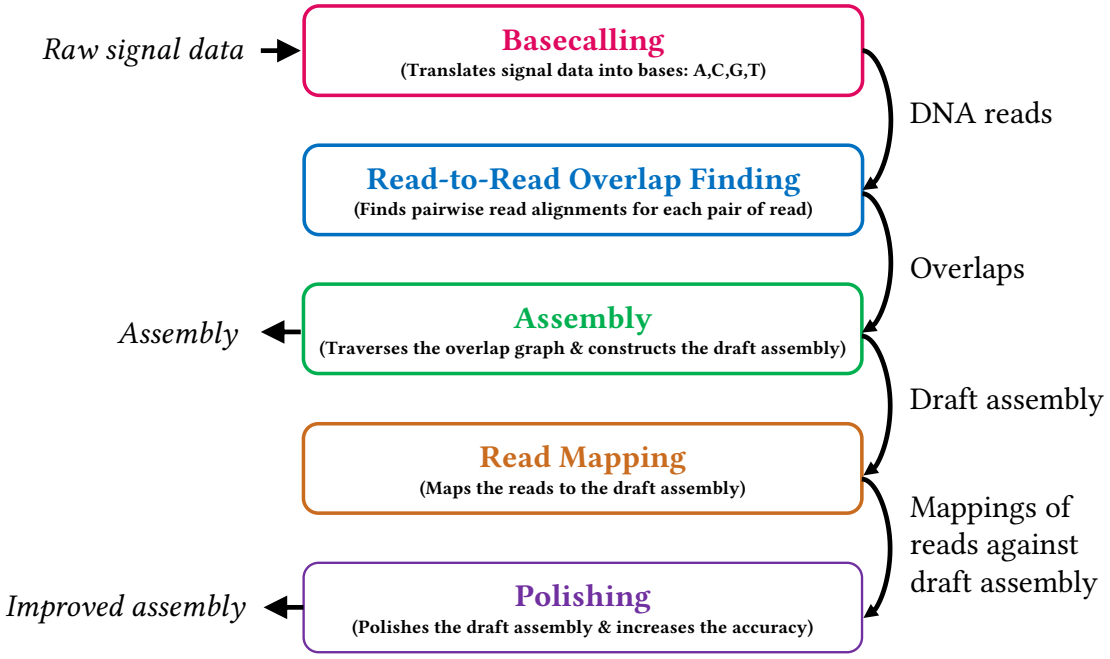



**Figure 2-1: Genome assembly pipeline using long reads, with its five steps and the associated tools for each step.**

is the only step unique to nanopore sequencing, and the rest of the steps are applicable for both of the long read sequencing technologies (ONT and PacBio). The second step computes all pairwise read alignments or suffix-prefix matches between each pair of reads, called *read-to-read overlaps*. Overlap-layout-consensus (OLC) algorithms are used for the assembly of nanopore sequencing reads since OLC-algorithms perform better with longer error-prone reads [252]. OLC-based assembly algorithms generate an *overlap graph*, where each node denotes a read and each edge represents the suffix-prefix match between the corresponding two nodes. The third pipeline step, *genome assembly*, traverses this overlap graph, producing the layout of the reads and then constructing a draft assembly. To increase the accuracy of the assembly, further *polishing*, i.e., post-assembly error correction, may be required. The fourth step of the pipeline is mapping the original basecalled reads to the generated draft assembly from the previous step (i.e., read mapping). The fifth and final step of the pipeline is polishing the assembly with the help of mappings from the previous step.

## 2.4 Read Mapping Pipeline

Another common approach for genome sequence analysis is to perform *read mapping*, where each *read* of an organism's sequenced genome is matched against the *reference genome for the organism's species* to find the read's original location. As Figure 2-2 shows, typical read mapping [182, 180, 14, 322, 174, 185] is a four-step process, which is also known as *seed-and-extend strategy*. First, read mapping starts with *indexing* ❶, which is an offline pre-processing step performed on a known reference genome. Second, once a sequencing machine generates reads from a DNA sequence, the *seeding* process ❶ queries the index structure to determine the candidate (i.e., potential) mapping locations of each read in the reference genome using substrings (i.e., *seeds*) from each read. Third, for each read, *pre-alignment filtering* ❷ uses filtering heuristics to examine the similarity between a read and the portion of the reference genome at each of the read's candidate mapping

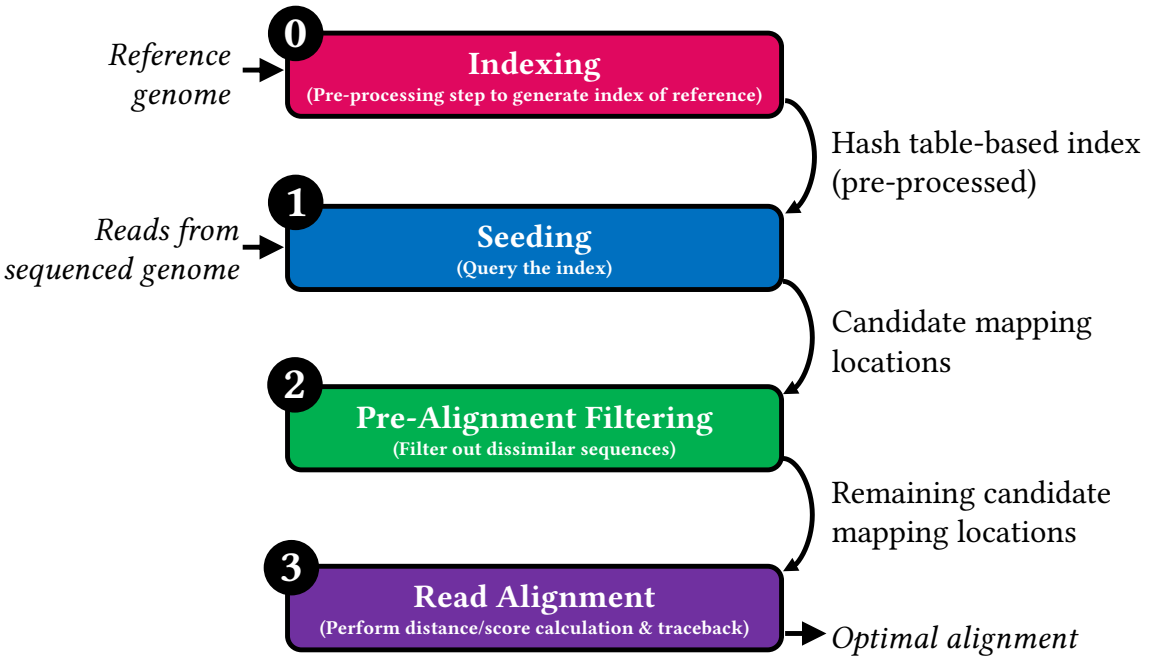



**Figure 2-2: Four steps of read mapping.**

locations. These filtering heuristics aim to eliminate most of the dissimilar pairs of reads and candidate mapping locations to decrease the number of required alignments in the next step. Fourth, for all of the remaining candidate mapping locations, *read alignment* ❸ runs a dynamic programming based algorithm to determine which of the candidate mapping locations in the reference matches best with the input read. As part of this step, traceback is performed between the reference and the input read to find the *optimal alignment*, which is the alignment with the highest likelihood of being correct (based on a scoring function [109, 207, 310]). The optimal alignment is defined using a *CIGAR string* [183], which shows the sequence and position of each match, substitution, insertion, and deletion for the read with respect to the selected mapping location of the reference.

## 2.5 Genome Graphs

Genetic variation between individuals is observed by comparing the differences between their two genomes. These differences, such as single-nucleotide polymorphisms (i.e., SNPs), insertions and deletions (i.e., indels), and structural variations (i.e., SVs), maintain genetic

diversity between populations and within communities [7]. However, the presence of these genomic portions creates limitations for mapping the sequenced reads to a reference genome [247, 72, 41, 116], since the reference is commonly represented as a linear DNA sequence [273]. Using a single reference introduces *reference allele bias*, by emphasizing the *alleles* (i.e., gene variants) that are present in the reference individual [247]. Alternate locus sequences (i.e., ALT, alternative subsequences for diverging regions of the reference DNA sequence [100, 148]) produced along with recent linear reference versions or utilizing pangenome models to include the collection of population genomes [6, 247] can alleviate the effect of reference allele bias. These factors lead to low read mapping accuracy around the diversity regions (SNPs, indels and SVs) and eventually cause false detection of SVs [261]. Thus, the current practices for variant detection techniques mostly depend on complex combinations of alignment patterns [13].

Sequence graphs of a genome are better suited for expressing the differences or ambiguities in diversity regions than linear reference sequences [99]. Therefore, there is a growing trend towards utilizing genome graphs [251, 228, 262, 99, 261, 163, 76] to more efficiently and accurately express the reference and its associated diversity annotations. Genome graphs are also more effective for presenting pangenomes [247] and extending the linear reference with alternate locus sequences to mitigate reference allele bias [72].

Genome graphs represent the reference genome *and* known genetic variations in the population as a graph-based data structure. As we show in Figure 2-3, a node represents one or more base pairs, and edges connect the base pairs in a node to all of the possible base pairs that come next in the sequence, with multiple outgoing edges from a node capture genetic variations. Thus, different paths in the graph translate to different sequences.

Genome graphs are growing in popularity for a number of applications, such as variation calling [99], genome assembly [58, 251, 326, 283], error correction [270], and multiple sequence alignment [246, 177]. With an increasing importance and usage of genome graphs, having efficient tools for mapping genomic sequences to these graphs become crucial.

**Reference #1:** ACGTACGT

**Reference #2:** ACGGACGT

**Reference #3:** ACGTTACGT

**Reference #4:** ACGACGT

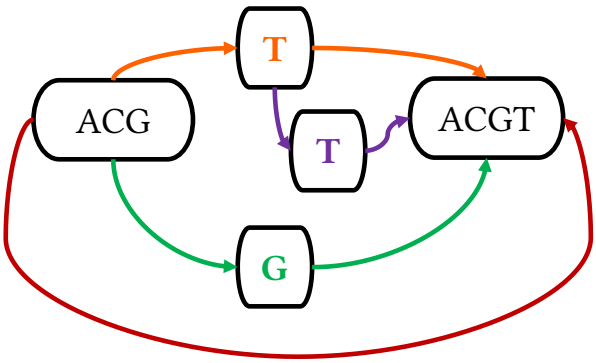



**Figure 2-3: Example of a genome graph that represents 4 related but different genomic sequences.**

## 2.6 Sequence-to-Graph Mapping

Similar to conventional sequence-to-sequence mapping (Section 2.4), sequence-to-graph mapping follows the *seed-and-extend strategy*. After constructing the graph using a linear reference genome and the associated variations for that genome, the nodes' of the graph are indexed as a pre-processing step. Later, this index is used in the *seeding* step, which aims to find seed matches between the query read and a region of the graph. After optionally clustering or filtering these seed matches with a *filtering* or *chaining* step, *alignment* is performed between all of the remaining seed locations of the graph and the query read. Even though sequence-to-sequence mapping is a well-studied problem, sequence-to-graph mapping is a newer problem and there are only a few existing tools, which are optimized for only short reads or only long reads, or specialized for a specific use cases.

# Chapter 3

# Bottleneck Analysis of the Genome Assembly Pipeline Using Long Reads

Due to the repetitive regions in the genome, the short-read length of the most dominant NGS technologies (*e.g.*, 100-150 bp reads) causes errors and ambiguities for read mapping [296, 90], and poses computational challenges and accuracy problems to *de novo* assembly [15]. Repetitive sequences are usually longer than the length of a short read and an entire repetitive sequence cannot be spanned by a single short read. Thus, short reads lead to highly-fragmented, incomplete assemblies [198, 15, 200]. However, a long read can span an entire repetitive sequence and enable continuous and complete assemblies.The demand for sequencing technologies that can produce longer reads has resulted in the emergence of even newer alternative sequencing technologies.

Nanopore sequencing technology [56] is one example of such technologies that can produce long read lengths. Nanopore sequencing is an emerging single-molecule DNA sequencing technology, which exhibits many attractive qualities, and in time, it could potentially surpass current sequencing technologies. Nanopore sequencing promises

high sequencing throughput, low cost, and longer read length, and it does *not* require an amplification step before the sequencing process [205, 42, 176, 147].

Using biological nanopores for DNA sequencing was first proposed in the 1990s [159], but the first nanopore sequencing device, MinION [210], was only recently (in May 2014) made commercially available by Oxford Nanopore Technologies (ONT). MinION is an inexpensive, pocket-sized, portable, high-throughput sequencing apparatus that produces data in real-time. These properties enable new potential applications of genome sequencing, such as rapid surveillance of Ebola, Zika or other epidemics [257], near-patient testing [258], and other applications that require real-time data analysis. In addition, the MinION technology has two major advantages. First, it is capable of generating ultra-long reads (*e.g.*, 882 kilobase pairs or longer [151, 195]). MinION's long reads greatly simplify the genome assembly process by decreasing the computational requirements [198, 199]. Second, it is small and portable. MinION is named as the first DNA sequencing device used in outer space to help the detection of life elsewhere in the universe with the help of its size and portability [223]. With the help of continuous updates to the MinION device and the nanopore chemistry, the first nanopore human reference genome was generated by using only MinION devices [151].

Nanopores are suitable for sequencing because they:

- Do not require any labeling of the DNA or nucleotide for detection during sequencing,
- Rely on the electronic or chemical structure of the different nucleotides for identification,
- Allow sequencing very long reads, and
- Provide portability, low cost, and high throughput.

Despite all these advantageous characteristics, nanopore sequencing has one major drawback: high error rates. In May 2016, ONT released a new version of MinION with a new nanopore chemistry called R9 [259], to provide higher accuracy and higher speed,

which replaced the previous version R7. Although the R9 chemistry improves the data accuracy, the improvements are not enough for cutting-edge applications. Thus, nanopore sequence analysis tools have a critical role to overcome high error rates and to take better advantage of the technology. Also, *faster* tools are critically needed to 1) take better advantage of the real-time data production capability of MinION and 2) enable real-time data analysis.

Our goal in this work is to comprehensively analyze current publicly-available tools for nanopore sequence analysis to understand their advantages, disadvantages, and bottlenecks. It is important to understand where the current tools do *not* perform well, to develop better tools. To this end, we analyze the tools associated with the multiple steps in the genome assembly pipeline using nanopore sequence data in terms of accuracy, speed, memory efficiency, and scalability.

## 3.1 Steps and Tools

### 3.1.1 Basecalling

When a strand of DNA passes through the nanopore (which is called the *translocation* of the strand through the nanopore), it causes drops in the electric current passing between the walls of the pore. The amount of change in the current depends on the type of base passing through the pore. Basecalling, the initial step of the entire pipeline, translates the raw signal output of the nanopore sequencer into bases (A, C, G, T) to generate DNA reads. Most of the current basecallers divide the raw current signal into discrete blocks, which are called *events*. After event-detection, each event is decoded into a most-likely set of bases. In the ideal case, each consecutive event should differ by one base. However, in practice, this is not the case because of the non-stable speed of the translocation. Also, determining the correct length of the homopolymers (*i.e.*, repeating stretches of one kind of base, *e.g.*, AAAAAAA) is challenging. Both of these problems make *deletions* the dominant error of

nanopore sequencing [276, 69]. Thus, basecalling is the most important step of the pipeline that plays a critical role in decreasing the error rate.

In this work, we analyze five state-of-the-art basecalling tools (Table 3-1). For a detailed comparison of these and other basecallers (including Albacore [12], which is not freely available, and Chiron [293]), we refer the reader to an ongoing basecaller comparison study [314]. Note that this ongoing study does *not* capture the accuracy and performance of the entire genome assembly pipeline using nanopore sequence data.

**Table 3-1: State-of-the-art nanopore basecalling tools.**

| Tool | Strategy | Multi-threading Support | Reference |
|---|---|---|---|
| Metrichor | RNN | (cloud-based) | [206] |
| Nanonet | RNN | with `-jobs` parameter | [221] |
| Scrappie | RNN | with `export OMP_NUM_THREADS` command | [277] |
| Nanocall | HMM | with `–threads` parameter | [67, 220] |
| DeepNano | RNN | no support; split dataset and run it in parallel | [40, 71] |

### Metrichor

Metrichor [206] is ONT's cloud-based basecaller, and its source code is not publicly available. Before the R9 update, Metrichor was using Hidden Markov Models (HMM) [78] for basecalling [259]. After the R9 update, it started using recurrent neural networks (RNN) [274, 249] for basecalling [259].

### Nanonet

Nanonet [221] has also been developed by ONT, and it is available on Github. Since Metrichor requires an Internet connection and its source code is not available, Nanonet is an offline and open-source alternative for Metrichor. Nanonet is implemented in Python. It also uses RNN for basecalling [221]. The tool supports multi-threading by sharing the computation needed to call each single read between concurrent threads. In other words, only one read is called at a time.

### Scrappie

Scrappie [277] is the newest proprietary basecaller developed by ONT. It is named as the first basecaller that explicitly addresses basecalling errors in homopolymer regions. In order to determine the correct length of homopolymers, Scrappie performs transducer-based basecalling [276]. For versions R9.4 and R9.5, Scrappie can perform basecalling with the raw current signal, without requiring event detection. It is a C-based local basecaller and is still under development [276].

### Nanocall

Nanocall [67] uses Hidden Markov Models for basecalling, and it is independently developed by a research group. It was released before the R9 update when Metrichor was also using an HMM-based approach for basecalling, to provide the first offline and open-source alternative for Metrichor. However, after the R9 update, when Metrichor started to perform basecalling with a more powerful RNN-based approach, Nanocall's accuracy fell short of Metrichor's accuracy [220]. Thus, although Nanocall supports R9 and upper versions of nanopore data, its usefulness is limited [220]. Nanocall is a C++-based command-line tool. It supports multi-threading where each thread performs basecalling for *different* groups of raw reads.

### DeepNano

DeepNano [40] is also independently developed by a research group before the R9 update. It uses an RNN-based approach to perform basecalling. Thus, it is considered to be the first RNN-based basecaller. DeepNano is implemented in Python. It does not have multi-threading support.

### 3.1.2 Read-to-Read Overlap Finding

Previous genome assembly methods designed for accurate and short reads (*i.e.*, de Bruijn graph (DBG) approach [251, 58]) are not suitable for nanopore reads because of the high error rates of the current nanopore sequencing devices [168, 69, 200, 55]. Instead, overlap-layout-consensus (OLC) algorithms [186] are used for nanopore sequencing reads since they perform better with longer, error-prone reads. OLC-based assembly algorithms start with finding the read-to-read overlaps, which is the second step of the pipeline. Read-to-read overlap is defined to be a common sequence between two reads [55]. GraphMap [290] and Minimap [181] are the commonly-used state-of-the-art tools for this step (Table 3-2).

**Table 3-2: State-of-the-art read-to-read overlap finding tools.**

| Tool | Strategy | Multi-threading Support | Reference |
|---|---|---|---|
| GraphMap | *k*-mer similarity | with `–threads` parameter | [290, 112] |
| Minimap | minimizer similarity | with `-t` parameter | [181, 209] |

Note: Both GraphMap and Minimap also have read mapping functionality.

#### GraphMap

GraphMap first partitions the entire read dataset into *k*-length substrings (i.e. *k*-mers), and then creates a hash table. GraphMap uses gapped *k*-mers, *i.e.*, k-mers that can contain insertions or deletions (indels) [290, 45]. In the hash table, for each *k*-mer entry, three pieces of information are stored: 1) k-mer string, 2) the index of the read, and 3) the position in the read where the corresponding *k*-mer comes from. GraphMap detects the overlaps by finding the *k*-mer similarity between any two given reads. Due to this design, GraphMap is a highly sensitive and accurate tool for error-prone long reads. It is a command-line tool written in C++. GraphMap is used for both 1) read-to-read overlap finding with the `graphmap owler` command and 2) read mapping with the `graphmap align` command.

### Minimap

Minimap also partitions the entire read dataset into *k*-mers, but instead of creating a hash table for the full set of *k*-mers, it finds the minimum representative set of *k*-mers, called *minimizers*, and creates a hash table with only these minimizers. Minimap finds the overlaps between two reads by finding minimizer similarity. The goals of using minimizers are to 1) reduce the storage requirement of the tool by storing fewer *k*-mers and 2) accelerate the overlap finding process by reducing the search span. Minimap also sorts *k*-mers for cache efficiency. Minimap is fast and cache-efficient, and it does not lose any sensitivity by storing minimizers since the chosen minimizers can represent the whole set of *k*-mers. Minimap is a command-line tool written in C. Like GraphMap, it can both 1) find overlaps between two read sets and 2) map a set of reads to a full genome.

## 3.1.3 Genome Assembly

After finding the read-to-read overlaps, OLC-based assembly algorithms generate an *overlap graph*. Genome assembly is performed by traversing this graph, producing the layout of the reads and then constructing a draft assembly. Canu [169] and Miniasm [181] are the commonly-used error-prone long-read assemblers (Table 3-3).

**Table 3-3: State-of-the-art assembly tools.**

| Tool | Strategy | Multi-threading Support | Reference |
|---|---|---|---|
| Canu | OLC with error correction | auto configuration | [169, 47] |
| Miniasm | OLC without error correction | no support | [181, 208] |

### Canu

Canu performs error-correction as the initial step of its own pipeline. It finds the overlaps of the raw uncorrected reads and uses them for the error-correction. The purpose of error-correction is to improve the accuracy of the bases in the reads [169, 48]. After the error-correction step, Canu finds overlaps between corrected reads and constructs a draft assembly after an additional trimming step. However, error-correction is a computationally

expensive step. In its own pipeline, Canu implements its own read-to-read overlap finding tool such that the users do *not* need to perform that step explicitly before running Canu. Most of the steps in the Canu pipeline are multi-threaded. Canu detects the resources that are available in the computer before starting its pipeline and automatically assigns number of threads, number of processes and amount of memory based on the available resources and the assembled genome's estimated size.

**Miniasm**

Miniasm skips the error-correction step since it is computationally expensive. It constructs a draft assembly from the uncorrected read overlaps computed in the previous step. Although Miniasm lowers computational cost and thus accelerates and simplifies assembly by doing so, the accuracy of the draft assembly depends directly on the accuracy of the uncorrected basecalled reads. Thus, further polishing may be necessary for these draft assemblies. Miniasm does not support multi-threading.

### 3.1.4 Read Mapping and Polishing

In order to increase the accuracy of the assembly, especially for the rapid assembly methods like Miniasm, which do not have the error-correction step, further polishing may be required. Polishing, *i.e.*, post-assembly error-correction, improves the accuracy of the draft assembly by mapping the reads to the assembly and changing the assembly to increase local similarity with the reads [196, 305, 69]. The first step of polishing is mapping the basecalled reads to the generated draft assembly from the previous step. One of the most commonly-used long read mappers for nanopore data is BWA-MEM [180]. Read-to-read overlap finding tools, GraphMap and Minimap (Section 3.1.2), can also be used for this step, since they also have a read mapping mode (Table 3-4).

After aligning the basecalled reads to the draft assembly, the final polishing of the assembly can be performed with Nanopolish [196] or Racon [305] (Table 3-5).

**Table 3-4: State-of-the-art read mapping tools.**

| Tool | Strategy | Multi-threading Support | Reference |
|---|---|---|---|
| BWA-MEM | Burrows-Wheeler Transform | with `-t` parameter | [180, 46] |
| GraphMap | *k*-mer similarity | with `–threads` parameter | [290, 112] |
| Minimap | minimizer similarity | with `-t` parameter | [181, 209] |

**Table 3-5: State-of-the-art polishing tools.**

| Tool | Strategy | Multi-threading Support | Reference |
|---|---|---|---|
| Nanopolish | Hidden Markov Model | with `–threads`, `-P` parameters | [196, 222] |
| Racon | Partial order alignment graph | with `–threads` parameter | [305, 260] |

### Nanopolish

Nanopolish uses the raw signal data of reads along with the mappings from the previous step to improve the assembly base quality by evaluating and maximizing the probabilities for each base with a Hidden Markov Model-based approach [196]. It can increase the accuracy of the draft assembly by correcting the homopolymer-rich parts of the genome. Although this approach can increase the accuracy significantly, it is computationally expensive, and thus time consuming. Nanopolish developers recommend BWA-MEM as the read mapper before running Nanopolish [222].

### Racon

Racon constructs partial order alignment graphs [177, 305] in order to find a consensus sequence between the reads and the draft assembly. After dividing the sequence into segments, Racon tries to find the best alignment to increase the accuracy of the draft assembly. Racon is a fast polishing tool, but it does not promise a high increase in accuracy as Nanopolish promises. However, multiple iterations of Racon runs or a combination of Racon and Nanopolish runs can improve accuracy significantly. Racon developers recommend Minimap as the read mapper to use before running Racon, since Minimap is both fast and sensitive [305].

## 3.2 Experimental Methodology

### 3.2.1 Dataset

In this work, we use *Escherichia coli* genome data as the test case, sequenced using the MinION with an R9 flowcell [194].

MinION sequencing has two types of workflows. In the 1D workflow, only the template strand of the double-stranded DNA is sequenced. In contrast, in the 2D workflow, with the help of a hairpin ligation, both the template and complement strands pass through the pore and are sequenced. After the release of R9 chemistry, 1D data became very usable in contrast to previous chemistries. Thus, we perform the analysis of the tools on 1D data.

MinION outputs one file in the *fast5* format for each read. The fast5 file format is a hierarchical data format, capable of storing both raw signal data and basecalled data returned by Metrichor. This dataset includes 164,472 reads, *i.e.*, fast5 files. Since all these files include both raw signal data and basecalled reads, we can use this dataset for both 1) using the local basecallers to convert raw signal data into the basecalled reads and 2) using the already basecalled reads by Metrichor.

### 3.2.2 Evaluation Systems

In this work, for accuracy and performance evaluations of different tools, we use three separate systems with different specifications. We use the first computer in the first part of the analysis, accuracy analysis. We use the second and third computers in the second part of the analysis, performance analysis, to compare the scalability of the analyzed tools in the two machines with different specifications (Table 3-6).

We choose the first system for evaluation since it has a larger memory capacity than a usual server and, with the help of a large number of cores, the tasks can be parallelized easily in order to get the output data quickly. We choose the second system, called *desktop*, since it represents a commonly-used desktop server. We choose the third system, called

**Table 3-6: Specifications of evaluation systems.**

| Name | Model | CPU Specifications | Main Memory Specifications | NUMA* Specifications |
|---|---|---|---|---|
| System 1 | 40-core Intel® Xeon® E5-2630 v4 CPU @ 2.20GHz | 20 physical cores<br>2 threads per core<br>40 logical cores with hyper-threading** | 128GB DDR4<br>2 channels,<br>2 ranks/channel<br>Speed: 2400MHz | 2 NUMA nodes, each with 10 physical cores, 64GB of memory and an 25MB of last level cache (LLC) |
| System 2 (*desktop*) | 8-core Intel® Core i7-2600 CPU @ 3.40GHz | 4 physical cores<br>2 threads per core<br>8 logical cores with hyper-threading** | 16GB DDR3<br>2 channels,<br>2 ranks/channel<br>Speed: 1333MHz | 1 NUMA node, with 4 physical cores, 16GB of memory and an 8MB of LLC |
| System 3 (*big-mem*) | 80-core Intel® Xeon® E7-4850 CPU @ 2.00GHz | 40 physical cores<br>2 threads per core<br>80 logical cores with hyper-threading** | 1TB DDR3<br>8 channels,<br>4 ranks/channel<br>Speed: 1066MHz | 4 NUMA nodes, each with 10 physical cores, 256GB of memory and an 24MB of LLC |

* NUMA (Non-Uniform Memory Access) is a computer memory design, where a processor accesses its local memory faster (*i.e.*, with lower latency) than a non-local memory (*i.e.*, memory local to another processor in another NUMA node). A NUMA node is composed of the local memory and the CPU cores (See Observation 6 in Section 3.3.1 for detail).

** Hyper-threading is Intel's simultaneous multithreading (SMT) implementation (See Observation 5 in Section 3.3.1 for detail).

*big-mem*, because of its large memory capacity. This *big-mem* system can be useful for those who would like to get results more quickly.

### 3.2.3 Accuracy Metrics

We compare each draft assembly generated after the assembly step and each improved assembly generated after the polishing step with the reference genome, by using the `dnadiff` command under the MUMmer package [211]. We use six metrics to measure accuracy, as defined in Table 3-7: 1) number of bases in the assembly, 2) number of contigs, 3) average identity, 4) coverage, 5) number of mismatches, and 6) number of indels.

### 3.2.4 Performance Metrics

We analyze the performance of each tool by running the associated command-line of each tool with the `/usr/bin/time -v` command. We use four metrics to quantify

performance as defined in Table 3-8: 1) wall clock time, 2) CPU time, 3) peak memory usage, and 4) parallel speedup.

**Table 3-7: Accuracy metrics.**

| Metric Name | Definition | Preferred Values |
|---|---|---|
| Number of bases | Total number of bases in the assembly | $\simeq$ Length of reference genome |
| Number of contigs | Total number of segments in the assembly | Lower ($\simeq 1$) |
| Average identity | Percentage similarity between the assembly and the reference genome | Higher ($\simeq 100\%$) |
| Coverage | Ratio of the number of aligned bases in the reference genome to the length of reference genome | Higher ($\simeq 100\%$) |
| Number of mismatches | Total number of single-base differences between the assembly and the reference genome | Lower ($\simeq 0$) |
| Number of indels | Total number of insertions and deletions between the assembly and the reference genome | Lower ($\simeq 0$) |

**Table 3-8: Performance metrics.**

| Metric Name | Definition | Preferred Values |
|---|---|---|
| Wall clock time | Elapsed time from the start of a program to the end | Lower |
| CPU time | Total amount of time the CPU spends in user mode (*i.e.*, to run the program's code) and kernel mode (*i.e.*, to execute system calls made by the program)* | Lower |
| Peak memory usage | Maximum amount of memory used by a program during its whole lifetime | Lower |
| Parallel speedup | Ratio of the time to run a program with 1 thread to the time to run it with N threads | Higher |

* If wall clock time < CPU time for a specific program, it means that the program runs in parallel.

## 3.3 Results and Analysis

In this section, we present our results obtained by analyzing the performance of different tools for each step in the genome assembly pipeline using nanopore sequence data in terms of accuracy and performance, using all the metrics we provide in Table 3-7 and Table 3-8. Additionally, Table 3-9 shows the tool version, the executed command, and the output of each analyzed tool. We divide our analysis into three main parts.

In the first part of our analysis, we examine the first three steps of the pipeline (*cf.* Figure 2-2). To this end, we first execute each basecalling tool (*i.e.*, one of Nanonet, Scrappie, Nanocall or DeepNano). Since Metrichor is a cloud-based tool and its source code is not

**Table 3-9: Versions, commands to execute, and outputs for each analyzed tool.**

| | Command* | Output |
|---|---|---|
| **Basecalling Tools** | | |
| Nanonet–v2.0 | `nanonetcall fast5_dir/ --jobs N --chemistry r9` | `reads.fasta` |
| Scrappie–v1.0.1 | (1)`export OMP_NUM_THREADS=N`<br>(2)`scrappie events --segmentation Segment_Linear:split_hairpin fast5_dir/ ...` | `reads.fasta` |
| Nanocall–v0.7.4 | `nanocall -t N fast5_dir/` | `reads.fasta` |
| DeepNano–e8a621e | `python basecall.py –directory fast5_dir/ --chemistry r9` | `reads.fasta` |
| **Read-to-Read Overlap Finding Tools** | | |
| GraphMap–v0.5.2 | `graphmap owler -L paf -t N -r reads.fasta -d reads.fasta` | `overlaps.paf` |
| Minimap–v0.2 | `minimap -Sw5 -L100 -m0 -tN reads.fasta reads.fasta` | `overlaps.paf` |
| **Assembly Finding Tools** | | |
| Canu–v1.6 | `canu -p ecoli -d canu-ecoli genomeSize=4.6m -nanopore-raw reads.fasta` | `draft.fasta` |
| Miniasm–v0.2 | `miniasm -f reads.fasta overlaps.paf` | `draft.gfa –> draft.fasta` |
| **Read Mapping Tools** | | |
| BWA-MEM–0.7.15 | (1)`bwa index draft.fasta`<br>(2)`bwa mem -x ont2d -t N draft.fasta reads.fasta` | `mappings.sam –> mappings.bam` |
| Minimap–v0.2 | `minimap -tN draft.fasta reads.fasta` | `mappings.paf` |
| **Polishing Tools** | | |
| Nanopolish–v0.7.1 | (1)`python nanopolish_makerange.py draft.fasta \| parallel -P M`<br>(2)`nanopolish variants --consensus polished.{1}.fa -w {1} -r reads.fasta -b mappings.bam -g draft.fasta -t N`<br>(3)`python nanopolish_merge.py polished.*.fa` | `polished.fasta` |
| Racon–v0.5.0 | `racon (–sam) –bq -1 -t N reads.fastq mappings.paf/(mappings.sam) draft.fasta` | `polished.fasta` |

* N corresponds to the number of threads and M corresponds to the number of parallel jobs.

available, we cannot execute Metrichor and get the performance metrics for it. After recording the performance metrics for each basecaller run, we execute either GraphMap or Minimap followed by Miniasm, or Canu itself, and record the performance metrics for each run. We obtain a draft assembly for each combination of these basecalling, read-to-read

overlap finding and assembly tools. For each draft assembly, we assess its accuracy by comparing the resulting draft assembly with the existing reference genome. We show the accuracy results in Table 3-10. We show the performance results in Table 3-11. We will refer to these tables in Sections 3.3.1 – 3.3.3.

In the second part of our analysis, we examine the last two steps of the pipeline (*cf.* Figure 2-2). To this end, for each obtained draft assembly, we execute each possible combination of different read mappers (*i.e.*, BWA-MEM or Minimap) and different polishers (*i.e.*, Nanopolish or Racon), and record the performance metrics for each step (*i.e.*, read mapping and polishing). We obtain a polished assembly after each run, and assess its accuracy by comparing it with the existing reference genome. For these two analyses, we use the first system, which has 40 logical cores, and execute each tool using 40 threads, which is the possible maximum number of threads for that particular machine. We show the accuracy results in Table 3-12. We show the performance results in Table 3-13. We will refer to these tables in Section 3.3.4.

In the third part of our analysis, we assess the scalability of all of the tools that have multi-threading support. For this purpose, we use the second and third systems to compare the scalability of these tools on two different system configurations. For each tool, we change the number of threads and observe the corresponding change in speed, memory usage, and parallel speedup. These results are depicted in Figures 3-1 – 3-5, and we will refer to them throughout Sections 3.3.1 – 3.3.4.

Sections 3.3.1 – 3.3.4 describe the major observations we make for each of the five steps of the pipeline (*cf.* Figure 2-2) based on our extensive evaluation results.

### 3.3.1 Basecalling Tools

As we discuss in Section 3.1.1, ONT's basecallers Metrichor, Nanonet and Scrappie, and another basecaller developed by Boza *et al.* (2017), DeepNano, use Recurrent Neural Networks (RNNs) for basecalling whereas Nanocall developed by David *et al.* (2016) uses Hidden Markov Models (HMM) for basecalling.

### Accuracy

Using RNNs is a more powerful basecalling approach than using HMMs since an RNN 1) does not make any assumptions about sequence length [292] and 2) is not affected by the repeats in the sequence [292, 67, 40]. However, it is still challenging to determine the correct length of the homopolymers even with an RNN.

In order to compare the accuracy of the analyzed basecallers, we group the accuracy results by each basecalling tool and compare them according to the defined accuracy metrics.

According to this analysis and the accuracy results shown in Table 3-10, we make the following key observation.

**Observation 1:** *The pipelines that start with Metrichor, Nanonet, or Scrappie as the basecaller have similar identity and coverage trends among all of the evaluated scenarios (i.e., tool combinations for the first three steps), but Scrappie has a lower number of mismatches and indels. However, Nanocall and DeepNano cannot reach these three basecallers' accuracies: they have lower identity and lower coverage.*

Since Nanonet is the local version of Metrichor, Nanonet and Metrichor's similar accuracy trends are expected. In addition to the power of the RNN-based approach, Scrappie tries to solve the homopolymer basecalling problem. Although Scrappie is in an early stage of development, it leads to a smaller number of indels than Metrichor or

**Table 3-10: Accuracy analysis results for the first three steps of the pipeline.**

| | | | | | | # Bases | # Contigs | Identity (%) | Coverage (%) | # Mismatches | # Indels |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Metrichor | + | – | + | Canu | 4,609,499 | 1 | 98.05 | 99.92 | 12,378 | 76,990 |
| 2 | Metrichor | + | Minimap | + | Miniasm | 4,402,675 | 1 | 87.71 | 94.85 | 249,096 | 372,704 |
| 3 | Metrichor | + | GraphMap | + | Miniasm | 4,500,155 | 2 | 86.22 | 96.95 | 237,747 | 360,199 |
| 4 | Nanonet | + | – | + | Canu | 4,581,728 | 1 | 97.92 | 99.97 | 11,971 | 83,248 |
| 5 | Nanonet | + | Minimap | + | Miniasm | 4,350,175 | 1 | 85.50 | 92.76 | 237,518 | 394,852 |
| 6 | Nanonet | + | GraphMap | + | Miniasm | 4,272,545 | 1 | 85.36 | 91.16 | 232,748 | 389,968 |
| 7 | Scrappie | + | – | + | Canu | 4,614,149 | 1 | 98.46 | 99.90 | 6,777 | 63,597 |
| 8 | Scrappie | + | Minimap | + | Miniasm | 4,877,399 | 8 | 86.94 | 90.04 | 184,669 | 363,025 |
| 9 | Scrappie | + | GraphMap | + | Miniasm | 4,368,417 | 1 | 86.78 | 89.86 | 189,192 | 372,245 |
| 10 | Nanocall | + | – | + | Canu | 1,299,808 | 86 | 93.33 | 28.93 | 21,985 | 61,217 |
| 11 | Nanocall | + | Minimap | + | Miniasm | 4,492,964 | 5 | 80.52 | 42.92 | 177,589 | 221,092 |
| 12 | Nanocall | + | GraphMap | + | Miniasm | 4,429,390 | 3 | 80.51 | 41.32 | 171,455 | 213,435 |
| 13 | DeepNano | + | – | + | Canu | 7,151,596 | 106 | 92.75 | 99.16 | 38,803 | 211,551 |
| 14 | DeepNano | + | Minimap | + | Miniasm | 4,252,525 | 1 | 82.38 | 65.00 | 199,122 | 335,761 |
| 15 | DeepNano | + | GraphMap | + | Miniasm | 4,251,548 | 1 | 82.39 | 64.92 | 197,914 | 335,064 |

Nanonet. Nanocall's poor accuracy results are due to the simple HMM-based approach it uses. Although DeepNano performs better than Nanocall with the help of its RNN-based approach, it results in a higher number of indels and a lower coverage of the reference genome.

### Performance

RNN and HMM are computationally-intensive algorithms. In HMM-based basecalling, the Viterbi algorithm [93] is used for decoding. The Viterbi algorithm is a sequential technique and its computation cannot currently be parallelized with multithreading. However, in RNN-based basecalling, multiple threads can work on different sections of the neural network and thus RNN computation can be parallelized with multithreading.

In order to measure and compare the performance of the selected basecallers, we first compare the recorded wall clock time, CPU time and memory usage metrics of each scenario for the first step of the pipeline. Based on the results provided in Table 3-11, we make the following key observation.

**Observation 2:** *RNN-based Nanonet and DeepNano are 2.6x and 2.3x faster than HMM-based Nanocall, respectively. Although Scrappie is also an RNN-based tool, it is 5.7x faster than Nanonet due to its C implementation as opposed to Nanonet's Python implementation.*

**Table 3-11: Performance analysis results for the first three steps of the pipeline.**

| | | | | | | **Step 1:** Basecaller | | | **Step 2:** Read-to-Read Overlap Finder | | | **Step 3:** Assembly | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Wall Clock Time (h:m:s) | CPU Time (h:m:s) | Memory Usage (GB) | Wall Clock Time (h:m:s) | CPU Time (h:m:s) | Memory Usage (GB) | Wall Clock Time (h:m:s) | CPU Time (h:m:s) | Memory Usage (GB) |
| 1 | Metrichor | + | – | + | Canu | –* | –* | –* | – | – | – | 44:12:31 | 502:18:56 | 5.76 |
| 2 | Metrichor | + | Minimap | + | Miniasm | | | | 2:15 | 41:37 | 12.30 | 1:09 | 1:09 | 1.96 |
| 3 | Metrichor | + | GraphMap | + | Miniasm | | | | 6:14 | 1:52:57 | 56.58 | 1:05 | 1:05 | 1.82 |
| 4 | Nanonet | + | – | + | Canu | | | | – | – | – | 11:32:40 | 129:07:16 | 5.27 |
| 5 | Nanonet | + | Minimap | + | Miniasm | 17:52:42 | 714:21:45 | 1.89 | 1:13 | 18:55 | 9.45 | 33 | 33 | 0.69 |
| 6 | Nanonet | + | GraphMap | + | Miniasm | | | | 3:18 | 48:27 | 29.16 | 32 | 32 | 0.65 |
| 7 | Scrappie | + | – | + | Canu | | | | – | – | – | 33:47:41 | 385:51:23 | 5.75 |
| 8 | Scrappie | + | Minimap | + | Miniasm | 3:11:41 | 126:19:06 | 13.36 | 2:52 | 1:10:26 | 12.40 | 1:29 | 1:29 | 1.98 |
| 9 | Scrappie | + | GraphMap | + | Miniasm | | | | 7:26 | 2:16:02 | 38.31 | 1:23 | 1:23 | 1.87 |
| 10 | Nanocall | + | – | + | Canu | | | | – | – | – | 1:35:23 | 27:58:29 | 3.77 |
| 11 | Nanocall | + | Minimap | + | Miniasm | 47:04:53 | 1857:37:56 | 37.73 | 1:15 | 16:08 | 12.19 | 20 | 20 | 0.47 |
| 12 | Nanocall | + | GraphMap | + | Miniasm | | | | 5:14 | 1:09:04 | 56.78 | 16 | 16 | 0.30 |
| 13 | DeepNano | + | – | + | Canu | | | | – | – | – | 1:15:48 | 17:31:07 | 3.61 |
| 14 | DeepNano | + | Minimap | + | Miniasm | 23:54:34 | 811:14:29 | 8.38 | 1:50 | 24:30 | 11.71 | 1:03 | 1:03 | 1.31 |
| 15 | DeepNano | + | GraphMap | + | Miniasm | | | | 5:18 | 1:17:06 | 54.64 | 58 | 58 | 1.10 |

* We cannot get the performance metrics for Metrichor since its source code is not available for us to run the tool by ourselves.

For a deeper understanding of these tools' advantages, disadvantages and bottlenecks, we also perform a scalability analysis for each basecaller by running it on the *desktop* server and the *big-mem* server separately, with 1, 2, 4, 8 (maximum for the *desktop* server), 16, 32, 40, 64 and 80 (maximum for the *big-mem* server) threads, and measuring the performance metrics for each configuration. Metrichor and DeepNano are not included in this analysis because Metrichor is a cloud-based tool and its source code is not available for us to change its number of threads, and DeepNano does not support multi-threading. Figure 3-1 shows the speed, memory usage and parallel speedup results of our evaluations. We make four observations.

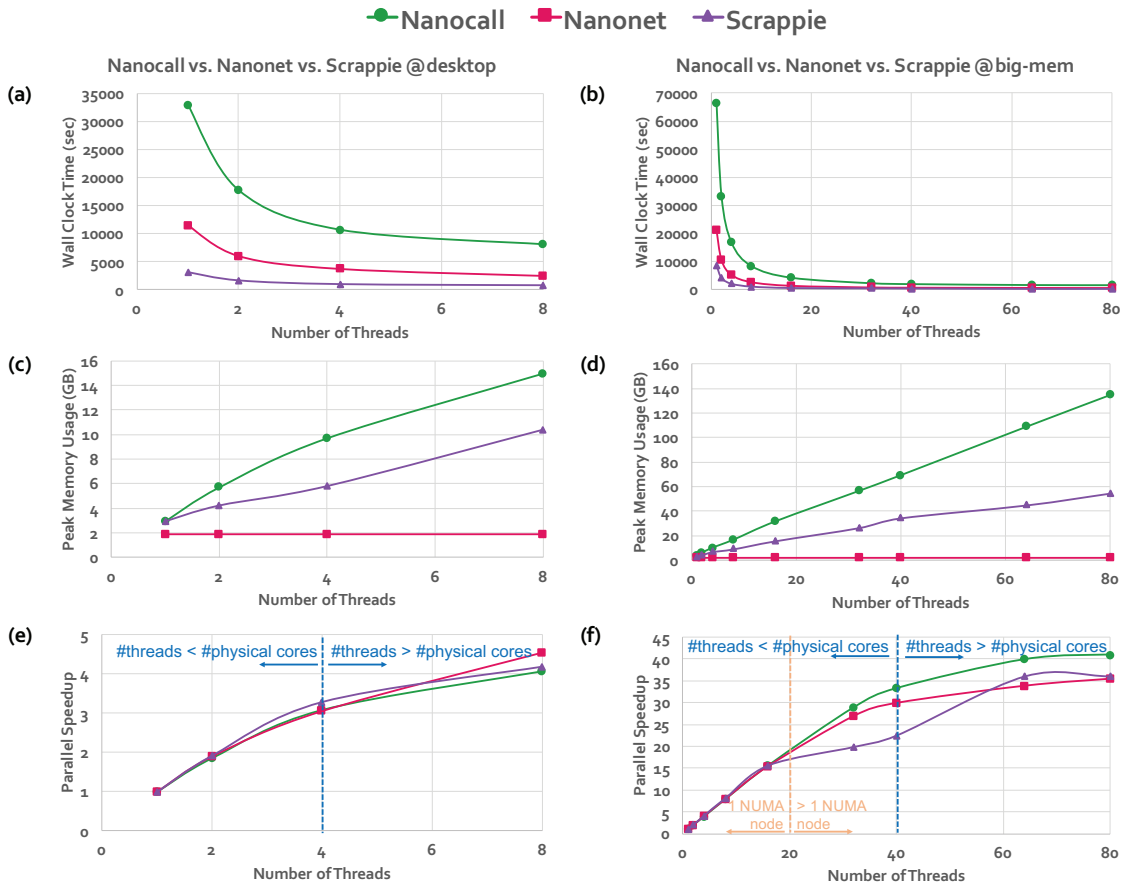



**Figure 3-1: Scalability results of Nanocall, Nanonet and Scrappie. Wall clock time (a, b), peak memory usage (c, d), and parallel speedup (e, f) results obtained on the *desktop* and *big-mem* systems. The left column (a, c, e) shows the results from the *desktop* system and the right column (b, d, f) shows the results from the *big-mem* system.**

**Observation 3:** *When we compare desktop's and big-mem's single thread performance, we observe that desktop is approximately 2x faster than big-mem (cf. Figure 3-1a and 3-1b).*

This is mainly because of *desktop*'s higher CPU frequency (see Table 6). It is an indication that all of these three tools are computationally expensive. Larger memory capacity or larger Last-Level Cache (LLC) capacity of *big-mem* cannot make up for the higher CPU speed in *desktop* when there is only one thread.

**Observation 4:** *Scrappie and Nanocall have a linear increase in memory usage when number of threads increases. In contrast, Nanonet has a constant memory usage for all evaluated thread units (cf. Figure 3-1c and 3-1d).*

In Scrappie and Nanocall, each thread performs the basecalling for different groups of raw reads. Thus, each thread allocates its own memory space for the corresponding data. This causes the linear increase in memory usage when the level of parallelism increases. In Nanonet, all of the threads share the computation of each read, and thus memory usage is not affected by the amount of thread parallelism.

**Observation 5:** *When the number of threads exceeds the number of physical cores, the simultaneous multithreading overhead prevents continued linear speedup of Nanonet, Scrappie and Nanocall (cf. Figure 3-1e and 3-1f).*

Simultaneous multithreading (SMT) (*i.e.*, running more than one thread per physical core [204, 201, 297, 299, 81, 298, 324, 124]), or more specifically Intel's hyper-threading (*i.e.*, since we use Intel's hyper-threading enabled machines (see Table 3-6)) helps to decrease the total runtime but it does *not* provide a linear speedup with the number of threads because of the CPU-intensive workload of Scrappie, Nanocall and Nanonet. If the threads executed are CPU-bound and do not wait for the memory or I/O requests, hyper-threading does not provide linear speedup due to the contention it causes in the shared resources for the computation. This phenomenon has been analyzed extensively in other application domains [204, 201, 297].

**Observation 6:** *Data sharing between threads degrades the parallel speedup of Nanonet when cores from multiple NUMA nodes take role in the computation (cf. Figure 3-1f).*

In Nanonet, data is shared between threads and each thread performs different computations on the same data. There are 4 NUMA nodes in *big-mem* (*cf.* Table 3-6), and when data is shared between multiple NUMA nodes, this negatively affects the speedup of Nanonet because accessing the data located in another node (*i.e.*, non-local memory) requires longer latency than accessing the data located in local memory. When multiple NUMA nodes start taking role in the computation, Nanocall performs better in terms of scalability since it does *not* require data sharing between different threads.

**Summary.** Based on the observations we make about the analyzed basecalling tools, we conclude that the choice of the tool for this step plays an important role to overcome the high error rates of nanopore sequencing technology. Basecalling with Recurrent Neural Networks (*e.g.*, Metrichor, Nanonet, Scrappie) provides higher accuracy and higher speed than basecalling with Hidden Markov Models, and the newest basecaller of ONT, Scrappie, also has the potential to overcome the homopolymer basecalling problem.

### 3.3.2 Read-to-Read Overlap Finding Tools

As we discuss in Section 3.1.2, GraphMap and Minimap are the commonly-used tools for this step. GraphMap finds the overlaps using $k$-mer similarity, whereas Minimap finds them by using minimizers instead of the full set of $k$-mers.

#### Accuracy

As done in GraphMap, finding the overlaps with the help of full set of $k$-mers is a highly-sensitive and accurate approach. However, it is also resource-intensive. For this reason, instead of the full set of $k$-mers, Minimap uses a minimum representative set of $k$-mers, minimizers, as an alternative approach for finding the overlaps.

In order to compare the accuracy of these two approaches, we categorize the results in Table 3-10 based on read-to-read overlap finding tools. In other words, we look at the rows with the same basecaller (*i.e.*, red-labeled tools) and same assembler (*i.e.*, green-labeled tools) but different read-to-read overlap finder (*i.e.*, blue-labeled tools). After that, we

compare them according to the defined accuracy metrics. We make the following major observation.

**Observation 7:** *Pipelines with GraphMap or Minimap end up with similar values for identity, coverage, number of indels and mismatches. Thus, either of these read-to-read overlap finding tools can be used in the second step of the nanopore sequencing assembly pipeline to achieve similar accuracy.*

Minimap and GraphMap do not have a significantly different effect on the accuracy of the generated draft assemblies. This is because Minimap does not lose any sensitivity by storing minimizers instead of the full set of *k*-mers.

### Performance

In order to compare the performance of GraphMap and Minimap, we categorize the results in Table 3-11 based on read-to-read overlap finding tools, in a similar way we describe the results in Table 3-10 for the accuracy analysis. We also perform a scalability analysis for each of these tools by running them on the *big-mem* server with 1, 2, 4, 8, 16, 32, 40, 64 and 80 threads, and measuring the performance metrics. Because of the high memory usage of GraphMap, data necessary for the tool does not fit in the memory of the *desktop* server and the GraphMap job exits due to a bad memory allocation exception. Thus, we could not perform the scalability analysis of GraphMap in the *desktop* server.

Figure 3-2 depicts the speed, memory usage and parallel speedup results of the scalability analysis for GraphMap and Minimap. We make the following three observations according to the results from Table 3-11 and Figure 3-2.

**Observation 8:** *The memory usage of both GraphMap and Minimap is dependent on the hash table size but independent of number of threads. Minimap requires 4.6x less memory than GraphMap, on average.*

This is mainly because Minimap stores only minimizers instead of all *k*-mers. Storing the full set of *k*-mers in GraphMap requires a larger hash table, and thus higher memory

usage than Minimap. The high amount of memory requirement causes GraphMap to not run on our *desktop* system for none of the selected number of thread units.

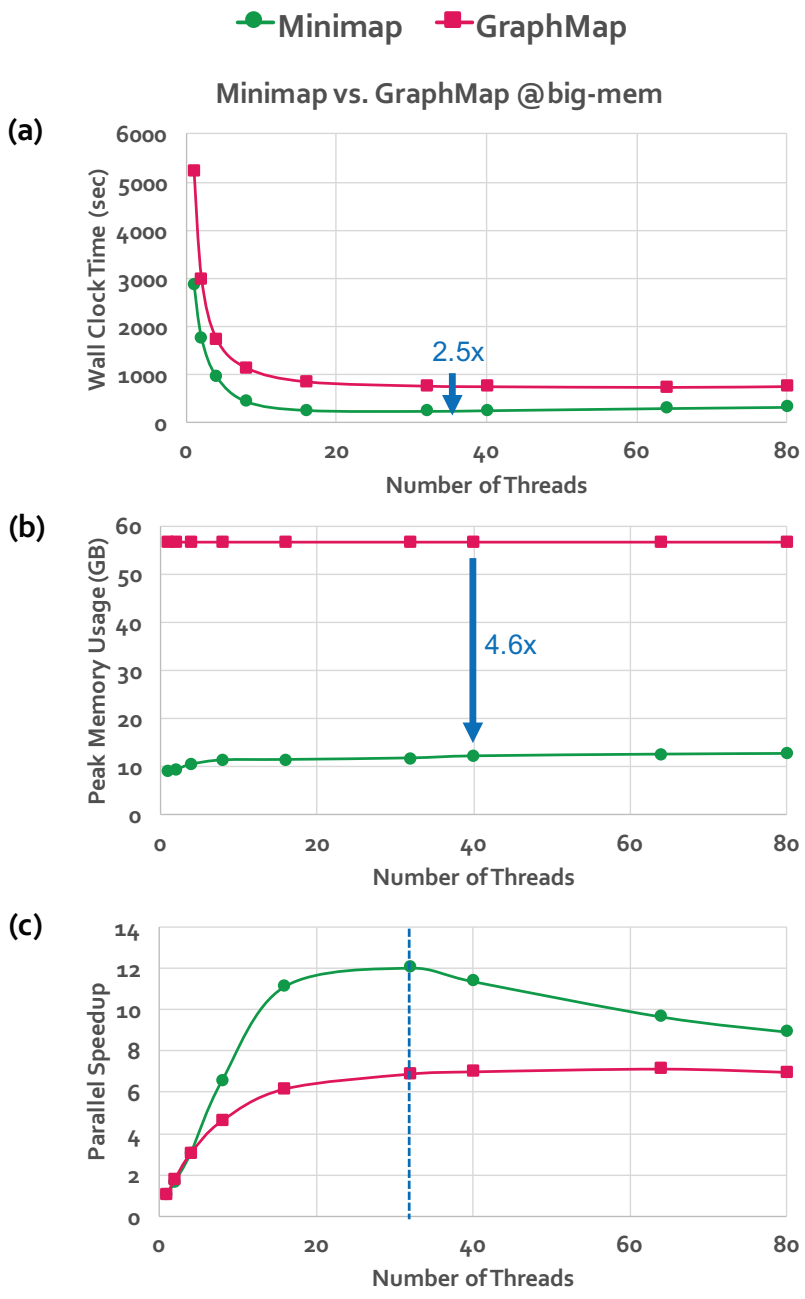



**Figure 3-2: Scalability results of Minimap and GraphMap. Wall clock time (a), peak memory usage (b), and parallel speedup (c) results obtained on the *big-mem* system.**

**Observation 9:** *Minimap is 2.5x faster than GraphMap, on average, across different scenarios in Table 3-11.*

Since GraphMap stores all $k$-mers, GraphMap needs to scan its very large dataset while finding the overlaps between two reads. However, in Minimap, the size of dataset that needs to be scanned is greatly shrunk by storing minimizers, as we describe in Observation 8. Thus, Minimap performs much less computation, leading to its 2.5x speedup. Another indication of the different memory usage and its effect on the speed of computation is the Last-Level Cache (LLC) miss rates of these two tools. The LLC miss rate of Minimap is 36% whereas the LLC miss rate of GraphMap is 55%. Since the size of data needed by GraphMap is much larger than the LLC size, GraphMap experiences LLC misses more frequently. As a result, GraphMap stalls for longer, waiting for data accesses from main memory, which negatively affects the speed of the tool.

**Observation 10:** *Minimap is more scalable than GraphMap. However, after 32 threads, there is a decrease in the parallel speedup of Minimap (cf. Figure 3-2c).*

Because of its lower computational workload and lower memory usage, we find that Minimap is more scalable than GraphMap. However, in Minimap, threads that finish their work wait for the other active threads to finish their workloads, before starting new work, in order to prevent higher memory usage. Because of this, when the number of threads reaches a high number (*i.e.*, 32 in Figure 3-2c), synchronization overhead greatly increases, causing the parallel speedup to reduce. GraphMap does not suffer from such a synchronization bottleneck and hence does not experience a decrease in speedup. However, GraphMap's speedup saturates when the number of threads reaches a high number due to data sharing between threads.

**Summary.** According to the observations we make about GraphMap and Minimap, we conclude that storing minimizers instead of all $k$-mers, as done by Minimap, does not affect the overall accuracy of the first three steps of the pipeline. Moreover, by storing minimizers, Minimap has a much lower memory usage and thus much higher performance than GraphMap.

### 3.3.3 Assembly Tools

As we discuss in Section 3.1.3, Canu and Miniasm are the commonly-used tools for this step.[1]

#### Accuracy

In order to compare the accuracy of these two tools, we categorize the results in Table 3-10 based on assembly tools. We make the following observation.

**Observation 11:** *Canu provides higher accuracy than Miniasm, with the help of the error-correction step that is present in its own pipeline.*

#### Performance

In order to compare the performance of Canu and Miniasm, we categorize the results in Table 3-11 based on assembly tools, in a way similar to what we did in Table 3-10 for the accuracy analysis. We could not perform a scalability analysis for these tools since Canu has an auto-configuration mechanism for each sub-step of its own pipeline, which does not allow us to change the number of threads, and Miniasm does not support multi-threading. We make the following observation according to the results in Table 3-11.

**Observation 12:** *Canu is much more computationally intensive and greatly (i.e., by 1096.3x) slower than Miniasm, because of its very expensive error-correction step.*

**Summary.** According to the observations we make about Canu and Miniasm, there is a tradeoff between accuracy and performance when deciding on the appropriate tool for this step. Canu produces highly accurate assemblies but it is resource intensive and slow. In contrast, Miniasm is a very fast assembler but it cannot produce as accurate draft assemblies as Canu. We suggest that Miniasm can potentially be used for fast initial

[1] In addition, we attempted to evaluate MECAT [318], another assembler. We were unable to draw any meaningful conclusions from MECAT, as its memory usage exceeds the 1TB available in our *big-mem* system.

analysis and then further polishing can be applied in the next step in order to produce higher-quality assemblies.

### 3.3.4 Read Mapping and Polishing Tools

As we discuss in Section 3.1.4, further polishing may be required for improving the accuracy of the low-quality draft assemblies. For this purpose, after aligning the reads to the generated draft assembly with BWA-MEM or Minimap, one can use Nanopolish or Racon to perform polishing and obtain improved assemblies (*i.e.*, consensus sequences).

Nanopolish accepts mappings only in *Sequence Alignment/Map (SAM)* format [183] and it works only with draft assemblies generated with the Metrichor-basecalled reads. On the other hand, Racon accepts both *Pairwise Mapping format (PAF)* mappings [181] and *SAM*-format mappings, but it requires the input reads and draft assembly files to be in *fastq* format [57], which includes quality scores. However, by using the `-bq -1` parameter, it is possible to disable the filtering used in Racon, which requires quality scores. Since our basecalled reads are in *fasta* format [250], in our experiments, we convert these *fasta* files into *fastq* files and disable the filtering with the corresponding parameter.

BWA-MEM generates mappings in *SAM* format whereas Minimap generates mappings in *PAF* format. Since Nanopolish requires *SAM* format input, we generate the mappings only with BWA-MEM and use them for Nanopolish polishing, in our analysis. On the other hand, since Racon accepts both formats, we generate the mappings and the overlaps with both BWA-MEM and Minimap, respectively, and use them for Racon polishing, in our analysis.

#### Accuracy

Table 3-12 presents the accuracy metrics results for Nanopolish (*i.e.*, Rows 1-3) and Racon (*i.e.*, Rows 4-23) pipelines. Based on these results, we make two observations.

**Observation 13:** *Both Nanopolish and Racon significantly increase the accuracy of the draft assemblies.*

**Table 3-12: Accuracy analysis results for the full pipeline with a focus on the last two steps.**

| | | | | | | | | | # Bases | # Contigs | Identity (%) | Coverage (%) | # Mismatches | # Indels |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Metrichor | + | – | + | Canu | + | BWA-MEM | + | Nanopolish | 4,683,072 | 1 | 99.48 | 99.93 | 8,198 | 15,581 |
| 2 | Metrichor | + | Minimap | + | Miniasm | + | BWA-MEM | + | Nanopolish | 4,540,352 | 1 | 92.33 | 96.31 | 162,884 | 182,965 |
| 3 | Metrichor | + | GraphMap | + | Miniasm | + | BWA-MEM | + | Nanopolish | 4,637,916 | 2 | 92.38 | 95.80 | 159,206 | 180,603 |
| 4 | Metrichor | + | – | + | Canu | + | BWA-MEM | + | Racon | 4,650,502 | 1 | 98.46 | 100.00 | 18,036 | 51,842 |
| 5 | Metrichor | + | – | + | Canu | + | Minimap | + | Racon | 4,648,710 | 1 | 98.45 | 100.00 | 17,906 | 52,168 |
| 6 | Metrichor | + | Minimap | + | Miniasm | + | BWA-MEM | + | Racon | 4,598,267 | 1 | 97.70 | 99.91 | 24,014 | 82,906 |
| 7 | Metrichor | + | Minimap | + | Miniasm | + | Minimap | + | Racon | 4,600,109 | 1 | 97.78 | 100.00 | 23,339 | 79,721 |
| 8 | Nanonet | + | – | + | Canu | + | BWA-MEM | + | Racon | 4,622,285 | 1 | 98.48 | 100.00 | 16,872 | 52,509 |
| 9 | Nanonet | + | – | + | Canu | + | Minimap | + | Racon | 4,620,597 | 1 | 98.49 | 100.00 | 16,874 | 52,232 |
| 10 | Nanonet | + | Minimap | + | Miniasm | + | BWA-MEM | + | Racon | 4,593,402 | 1 | 98.01 | 99.97 | 20,322 | 72,284 |
| 11 | Nanonet | + | Minimap | + | Miniasm | + | Minimap | + | Racon | 4,592,907 | 1 | 98.04 | 100.00 | 20,170 | 70,705 |
| 12 | Scrappie | + | – | + | Canu | + | BWA-MEM | + | Racon | 4,673,871 | 1 | 98.40 | 99.98 | 13,583 | 60,612 |
| 13 | Scrappie | + | – | + | Canu | + | Minimap | + | Racon | 4,673,606 | 1 | 98.40 | 99.98 | 13,798 | 60,423 |
| 14 | Scrappie | + | Minimap | + | Miniasm | + | BWA-MEM | + | Racon | 5,157,041 | 8 | 97.87 | 99.80 | 18,085 | 78,492 |
| 15 | Scrappie | + | Minimap | + | Miniasm | + | Minimap | + | Racon | 5,156,375 | 8 | 97.87 | 99.94 | 17,922 | 77,807 |
| 16 | Nanocall | + | – | + | Canu | + | BWA-MEM | + | Racon | 1,383,851 | 86 | 93.49 | 28.82 | 19,057 | 65,244 |
| 17 | Nanocall | + | – | + | Canu | + | Minimap | + | Racon | 1,367,834 | 86 | 94.43 | 28.74 | 15,610 | 55,275 |
| 18 | Nanocall | + | Minimap | + | Miniasm | + | BWA-MEM | + | Racon | 4,707,961 | 5 | 90.75 | 97.11 | 91,502 | 347,005 |
| 19 | Nanocall | + | Minimap | + | Miniasm | + | Minimap | + | Racon | 4,673,069 | 5 | 92.23 | 97.10 | 72,646 | 291,918 |
| 20 | DeepNano | + | – | + | Canu | + | BWA-MEM | + | Racon | 7,429,290 | 106 | 96.46 | 99.24 | 27,811 | 102,682 |
| 21 | DeepNano | + | – | + | Canu | + | Minimap | + | Racon | 7,404,454 | 106 | 96.03 | 99.21 | 34,023 | 110,640 |
| 22 | DeepNano | + | Minimap | + | Miniasm | + | BWA-MEM | + | Racon | 4,566,253 | 1 | 96.76 | 99.86 | 25,791 | 125,386 |
| 23 | DeepNano | + | Minimap | + | Miniasm | + | Minimap | + | Racon | 4,571,810 | 1 | 96.90 | 99.97 | 24,994 | 119,519 |

For example, Nanopolish increases the identity and coverage of the draft assembly generated with the Metrichor+Minimap+Miniasm pipeline from 87.71% and 94.85% (Row 2 of Table 3-10), respectively, to 92.33% and 96.31% (Row 2 of Table 3-12). Similarly, Racon increases them to 97.70% and 99.91% (Rows 6–7 of Table 3-12), respectively.

**Observation 14:** *For Racon, the choice of read mapper does not affect the accuracy of the polishing step.*

We observe that using BWA-MEM or Minimap to generate the mappings for Racon results in almost identical accuracy metric results. For example, when we use BWA-MEM before Racon for the draft assembly generated with the Metrichor + Canu pipeline (Row 4 of Table 3-12), Racon results with 98.46% identity, 100.00% coverage, 18,036 mismatches and 51,842 indels. When we use Minimap, instead (Row 5 of Table 3-12), Racon results with 98.45% identity, 100.00% coverage, 17,096 mismatches and 52,168 indels, which is almost identical to the BWA-MEM results.

### Performance

In the first part of the performance analysis for Nanopolish, we divide the draft assemblies into 50kb-segments and polish 4 of these segments in parallel with 10 threads for each segment. For Racon, each draft assembly is polished using 40 threads, but the tool, by

default, divides the input sequence into windows of 20kb length. Table 3-13 presents the performance results for Nanopolish (*i.e.*, Rows 1-3) and Racon (*i.e.*, Rows 4-23) pipelines. Based on these results, we make the following two observations.

**Observation 15:** *Nanopolish is computationally much more intensive and thus greatly slower than Racon.*

Nanopolish runs take days to complete whereas Racon runs take minutes. This is mainly because Nanopolish works on each base individually, whereas Racon works on the windows. Since each window is much longer (*i.e.*, 20kb) than a single base, the computational workload is greatly smaller in Racon. Also, Racon only uses the mappings/overlaps for polishing, whereas Nanopolish uses raw signal data and an HMM-based approach in order to generate the consensus sequence, which is computationally more expensive.

**Observation 16:** *BWA-MEM is computationally more expensive than Minimap.*

Although the choice of BWA-MEM and Minimap for the read mapping step does not affect the accuracy of the polishing step, these two tools have a significant difference in performance.

**Table 3-13: Performance analysis results for the full pipeline with a focus on the last two steps.**

| | | | | | | | | | | **Step 4:** Read Mapper | | | **Step 5:** Polisher | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | Wall Clock Time (h:m:s) | CPU Time (h:m:s) | Memory Usage (GB) | Wall Clock Time (h:m:s) | CPU Time (h:m:s) | Memory Usage (GB) |
| 1 | Metrichor | + | – | + | Canu | + | BWA-MEM | + | Nanopolish | 24:43 | 15:47:21 | 5.26 | 5:51:00 | 191:18:52 | 13.38 |
| 2 | Metrichor | + | Minimap | + | Miniasm | + | BWA-MEM | + | Nanopolish | 12:33 | 7:50:54 | 3.75 | 122:52:00 | 4458:36:10 | 31.36 |
| 3 | Metrichor | + | GraphMap | + | Miniasm | + | BWA-MEM | + | Nanopolish | 12:47 | 7:57:58 | 3.60 | 129:46:00 | 4799:03:51 | 31.31 |
| 4 | Metrichor | + | – | + | Canu | + | BWA-MEM | + | Racon | 24:20 | 15:43:40 | 6.60 | 14:44 | 9:09:22 | 8.11 |
| 5 | Metrichor | + | – | + | Canu | + | Minimap | + | Racon | 3 | 1:35 | 0.26 | 15:12 | 9:45:33 | 14.55 |
| 6 | Metrichor | + | Minimap | + | Miniasm | + | BWA-MEM | + | Racon | 12:10 | 7:48:10 | 5.19 | 15:43 | 9:33:39 | 9.98 |
| 7 | Metrichor | + | Minimap | + | Miniasm | + | Minimap | + | Racon | 3 | 1:24 | 0.26 | 20:28 | 8:57:40 | 18.24 |
| 8 | Nanonet | + | – | + | Canu | + | BWA-MEM | + | Racon | 9:08 | 5:53:18 | 4.84 | 6:33 | 4:02:10 | 4.47 |
| 9 | Nanonet | + | – | + | Canu | + | Minimap | + | Racon | 2 | 54 | 0.26 | 6:45 | 4:17:26 | 7.93 |
| 10 | Nanonet | + | Minimap | + | Miniasm | + | BWA-MEM | + | Racon | 4:40 | 2:58:02 | 3.88 | 7:08 | 4:19:30 | 5.35 |
| 11 | Nanonet | + | Minimap | + | Miniasm | + | Minimap | + | Racon | 2 | 46 | 0.26 | 7:01 | 4:18:48 | 9.53 |
| 12 | Scrappie | + | – | + | Canu | + | BWA-MEM | + | Racon | 33:41 | 21:11:06 | 8.66 | 13:32 | 8:24:44 | 7.58 |
| 13 | Scrappie | + | – | + | Canu | + | Minimap | + | Racon | 3 | 1:39 | 0.27 | 18:45 | 7:43:17 | 13.20 |
| 14 | Scrappie | + | Minimap | + | Miniasm | + | BWA-MEM | + | Racon | 22:41 | 14:31:00 | 6.08 | 14:37 | 8:53:59 | 9.50 |
| 15 | Scrappie | + | Minimap | + | Miniasm | + | Minimap | + | Racon | 3 | 1:27 | 0.27 | 15:10 | 9:02:45 | 12.72 |
| 16 | Nanocall | + | – | + | Canu | + | BWA-MEM | + | Racon | 4:52 | 3:01:15 | 3.80 | 11:07 | 3:26:52 | 5.63 |
| 17 | Nanocall | + | – | + | Canu | + | Minimap | + | Racon | 3 | 1:16 | 0.22 | 7:28 | 2:50:35 | 3.62 |
| 18 | Nanocall | + | Minimap | + | Miniasm | + | BWA-MEM | + | Racon | 16:06 | 10:27:20 | 5.06 | 18:56 | 11:32:45 | 11.47 |
| 19 | Nanocall | + | Minimap | + | Miniasm | + | Minimap | + | Racon | 4 | 1:18 | 0.26 | 11:49 | 7:08:59 | 10.98 |
| 20 | DeepNano | + | – | + | Canu | + | BWA-MEM | + | Racon | 17:36 | 11:30:20 | 4.43 | 12:48 | 7:13:04 | 8.88 |
| 21 | DeepNano | + | – | + | Canu | + | Minimap | + | Racon | 3 | 1:24 | 0.28 | 11:39 | 6:55:01 | 3.73 |
| 22 | DeepNano | + | Minimap | + | Miniasm | + | BWA-MEM | + | Racon | 8:15 | 5:22:29 | 4.11 | 14:16 | 8:34:32 | 10.30 |
| 23 | DeepNano | + | Minimap | + | Miniasm | + | Minimap | + | Racon | 3 | 1:10 | 0.26 | 12:29 | 7:55:32 | 17.11 |

For a deeper performance analysis of these read mapping and polishing tools, we perform a scalability analysis for each read mapper and each polisher by running them on the *desktop* system and the *big-mem* system separately, with 1, 2, 4, 8 (maximum for *desktop* server), 16, 32, 40, 64 and 80 (maximum for *big-mem* server) threads, and measuring the performance metrics. Figure 3-3 shows the the speed, memory usage and parallel speedup of BWA-MEM and Minimap. We make two observations.

**Observation 17:** *On both systems, Minimap is greatly faster than BWA-MEM (cf. Figure 3-3a and 3-3b). However, when the number of threads reaches high value, Minimap's performance degrades due to the synchronization overhead between its threads (cf. Figure 3-3f).*

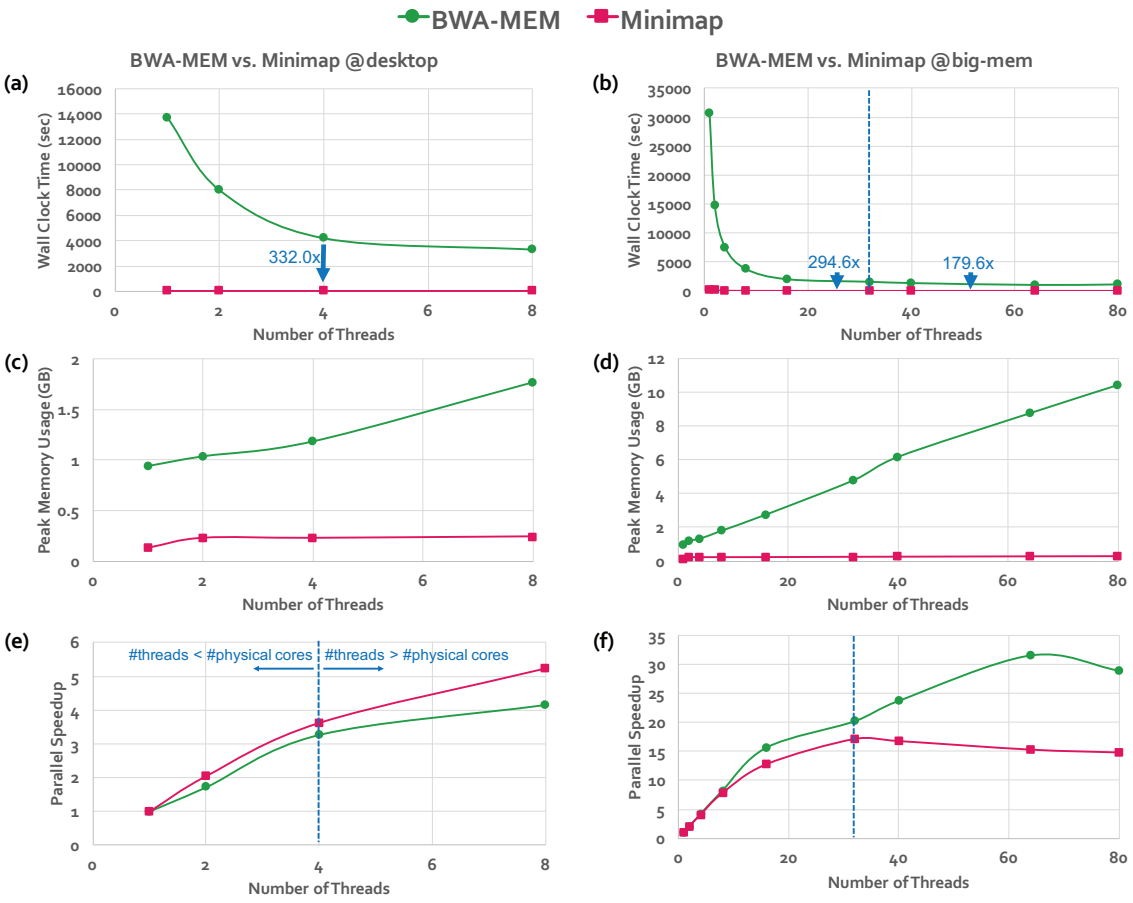



**Figure 3-3: Scalability results of BWA-MEM and Minimap. Wall clock time (a, b), peak memory usage (c, d), and parallel speedup (e, f) results obtained on the *desktop* and *big-mem* systems. The left column (a, c, e) shows the results from the *desktop* system and the right column (b, d, f) shows the results from the *big-mem* system.**

On the *desktop* system, Minimap is 332.0x faster than BWA-MEM, on average (see Figure 3-3a). On the *big-mem* system, Minimap is 294.6x and 179.6x faster than BWA-MEM, on average, when the number of threads is smaller and greater than 32, respectively. This is due to the synchronization overhead that increases with the number of threads used in Minimap (see Observation 10). As we also show in Figure 3-3f, Minimap's speedup reduces when the number of threads exceeds 32, which is another indication of the synchronization overhead that causes Minimap to slow down.

**Observation 18:** *Minimap's memory usage is independent of the number of threads and stays constant. In contrast, BWA-MEM's memory usage increases linearly with the number of threads (cf. Figure 3-3c and 3-3d).*

In Minimap, memory usage is dependent on the hash table size and is independent of number of threads (see Observation 8). In contrast, in BWA-MEM, each thread separately performs computation for different groups of reads (as in Scrappie and Nanocall, see Observation 4). This causes the linear increase in memory usage of BWA-MEM when the number of threads increases.

Figure 3-4 shows the scalability results for Racon on the *big-mem* system. We obtain the results on both of the systems. However, we only show the results for the *big-mem* system since the results for both of the systems are similar. We separately test the tool by using *PAF* mappings and *SAM* mappings. Based on the results, we make the following observation.

**Observation 19:** *Racon's memory usage is independent of the number of threads for both PAF mode and SAM mode. However, the memory usage of PAF mode is 1.86x higher than the memory usage of SAM mode, on average (cf. Figure 3-4b).*

The memory usage of Racon depends on the number of mappings received from the fourth step since Racon performs polishing by using these mappings. Racon's memory usage is higher for the PAF mode because the number of mappings stored in the PAF files is greater than the number of mappings stored in the SAM files (*i.e.*, 1.4x). However, using

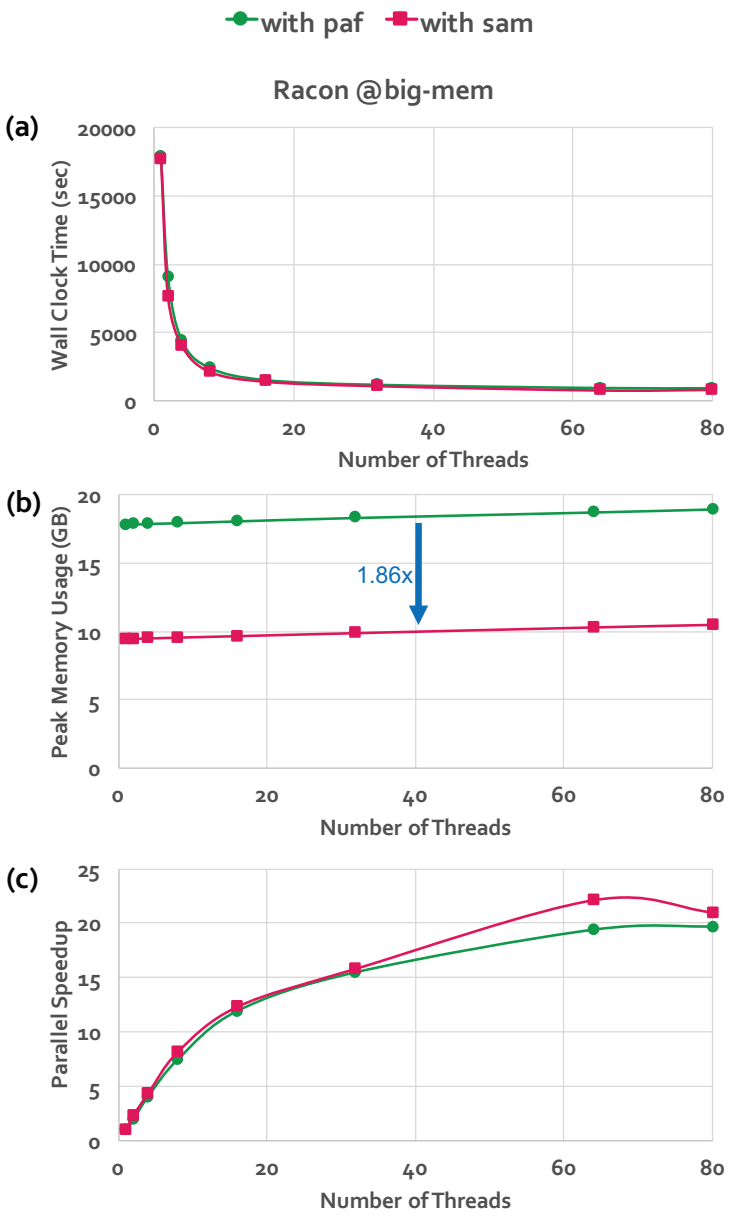



**Figure 3-4: Scalability results of Racon. Wall clock time (a), peak memory usage (b), and parallel speedup (c) results obtained on the *big-mem* system.**

*PAF* mappings or *SAM* mappings do not significantly affect the speed (see Figure 3-4a) and the parallel speedup (see Figure 3-4c) of Racon.

Figure 3-5 shows the scalability results for Nanopolish. We test the tool by separately using a 25kb and a 50kb segment length to assess the scalability of the tool with respect to the segment length, in addition to the scalability with respect to the number of threads. We measure the performance metrics. We only show the results for the *big-mem* system

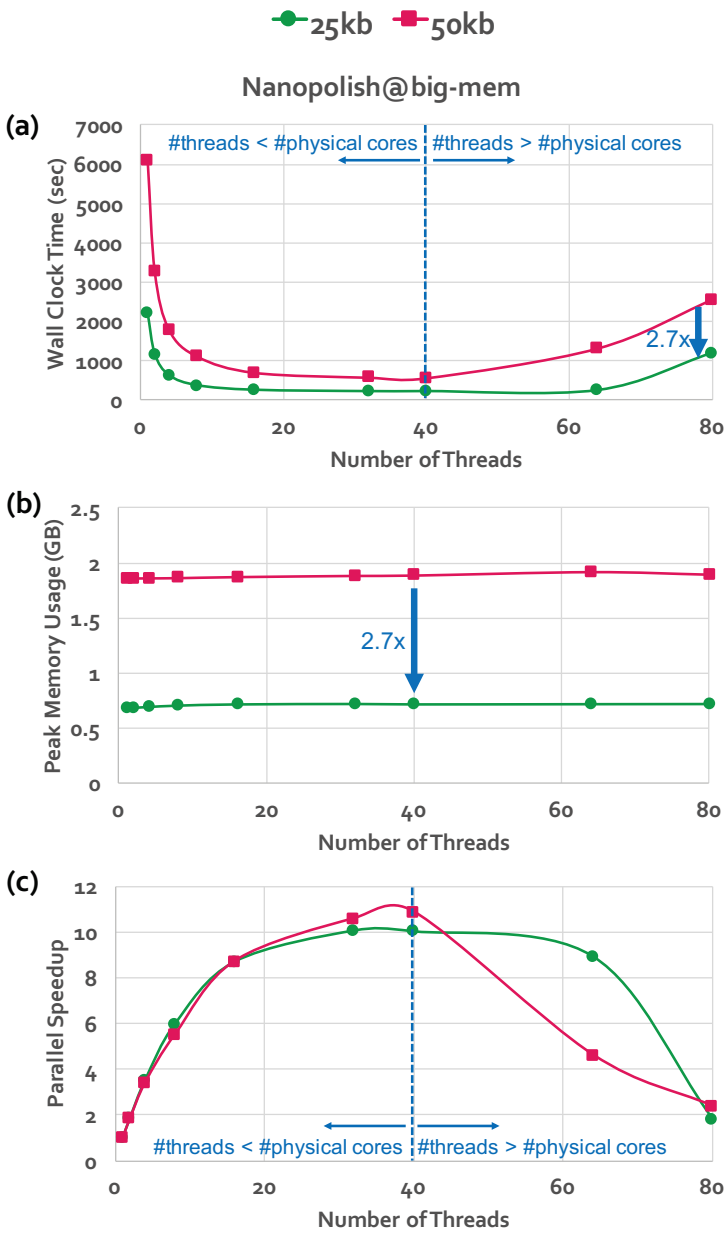



**Figure 3-5: Scalability results of Nanopolish. Wall clock time (a), peak memory usage (b), and parallel speedup (c) results obtained on the *big-mem* system.**

since the results for both of the systems are similar. Based on the results, we make the following observation.

**Observation 20:** *Nanopolish's memory usage is independent of the number of threads. However, its memory usage in dependent on the segment length (cf. Figure 3-5b).*

The memory usage of Nanopolish is not affected by the number of threads. However, it is dependent on the segment length. Nanopolish uses more memory for longer segments. When the segment length is doubled from 25kb to 50kb, the increase in the memory usage (*i.e.*, 2.7x) is greater than 2.0x. This is because the memory usage of Nanopolish depends both on the length of the segment and the number of read mappings that map to this segment. For both of the segments, the memory usage also affects the speed. The Nanopolish run for the 25kb-segment is 2.7x faster than the run for the 50kb-segment (see Figure 3-5a).

**Observation 21:** *Nanopolish's performance greatly degrades when the number of threads exceeds the number of physical cores (cf. Figure 3-5c).*

Hyper-threading causes a slowdown for Nanopolish because of the CPU-intensive workload of Nanopolish and the resulting high contention in the shared resources between the threads executing on the same core, as we discuss in Observation 5.

**Summary.** Based on the observations we make about tools for the optional last two steps of the pipeline, we conclude that further polishing can significantly increase the accuracy of the assemblies. Since BWA-MEM and Nanopolish are more resource-intensive than Minimap and Racon, pipelines with Minimap and Racon can provide a significant speedup compared to the pipelines with BWA-MEM and Nanopolish, while resulting with high-quality consensus sequences.

## 3.4 Recommendations

### 3.4.1 Recommendations for Tool Users

Based on the results we have collected and observations we have made for each step of the genome assembly pipeline using nanopore sequence data and the associated tools, we make the following major recommendations for the current and future tool users.

- ONT's basecalling tools, Metrichor, Nanonet, and Scrappie, are the best choices for the basecalling step in terms of both accuracy and performance. Among these tools,

Scrappie is the newest, fastest and most accurate basecaller. Thus, we recommend using Scrappie for the basecalling step (See analysis in Section 3.3.1).

- For the read-to-read overlap finding step, Minimap is faster than GraphMap, and it requires low memory. Also, it has similar accuracy to GraphMap. Thus, we recommend Minimap for the read-to-read overlap finding step (See analysis in Section 3.3.2).

- For the assembly step, if execution time is not an important concern, we recommend using Canu since it produces much more accurate assemblies. However, for a fast initial analysis, we recommend using Miniasm since it is fast and its accuracy can be increased with an additional polishing step. If Miniasm is used for assembly, we definitely recommend further polishing to increase the accuracy of the final assembly (See analysis in Section 3.3.3). Even though polishing takes a similar amount of time if we use Miniasm or Canu, the accuracy improvements are much *smaller* for a genome assembled using Canu. We hope that future work can improve the performance of polishing when the assembled genome already has high accuracy, to reduce the execution time of the overall assembly pipeline.

- For the polishing step, we recommend using Racon since it is much faster than Nanopolish. Racon also produces highly-accurate assemblies (See analysis in Section 3.3.4).

- In the future, laptops may become a popular platform for running genome assembly tools, as the portability of a laptop makes it a good fit for in-field analysis. Compared to the desktop and server platforms that we use to test our pipelines, a laptop has even greater memory constraints and lower computational power, and we must factor in limited battery life when evaluating the tools. Based on the scalability studies we perform using our desktop and server platforms, we would likely recommend using Minimap followed by Miniasm for the assembly step, and Minimap followed by Racon for the polishing step, when performing assembly on a laptop. These three

tools use relatively low amounts of memory, and execute quickly, which we expect would make the tools a good fit for the various constraints of a laptop. Despite their low memory usage and fast execution, our recommended pipeline can produce high-quality assemblies that are suitable for fast initial in-field analyses. We leave it to future work to quantitatively study the genome assembly pipeline using nanopore sequence data on laptops and other mobile devices.

### 3.4.2 Recommendations for Tool Developers

Based on our analyses, we make the following recommendations for the tool developers.

- The choice of language to implement the tool plays a crucial role regarding the overall performance of the tool. For example, although the basecallers Scrappie and Nanonet belong to the same family (*i.e.*, they both use the more accurate RNNs for basecalling), Scrappie is significantly faster than Nanonet since Scrappie is implemented in C whereas Nanonet is implemented in Python (See analysis in Section 3.3.1).

- Memory usage is an important factor that greatly affects the performance and the usability of the tool. While developing new tools or improving the current ones, the developers should be aware of the memory hierarchy. Data structure choices that can minimize the memory requirements and cache-efficient algorithms have a positive impact on the overall performance of the tools. Keeping memory usage in check with the number of threads can enable not only a usable (*i.e.*, runnable on machines with relatively small memories) tool but also a fast one. For example, we find that GraphMap cannot even run with a single-thread in our *desktop* machine due to excessively high memory usage (See analyses in Sections 3.3.1– 3.3.4).

- Scalability of the tool with the number of cores/threads is an important requirement. It is important to make the tool efficiently parallelized to decrease the overall runtime. Design choices should be made wisely while considering the possible overheads that parallelization can add. For example, we find that the parallel speedup of Minimap

reduces when the number of threads reaches a high number due to a large increase in the overhead of synchronization between threads (See analyses in Sections 3.3.1– 3.3.4).

- Since parallelizing the tool can increase the memory usage, dividing the input data into batches, or limiting the memory usage of each thread, or dividing the computation instead of dividing the dataset between simultaneous threads can prevent large increases in memory usage, while providing performance benefits from parallelization. For example, in Nanonet, all of the threads share the computation of each read, and thus memory usage is not affected by the amount of thread parallelism. As a result, Nanonet's usability is not limited to machines with relatively larger memories (See analyses in Sections 3.3.1– 3.3.4).

## 3.5 Summary

We analyze the multiple steps and the associated state-of-the-art tools in the genome assembly pipeline using nanopore sequence data in terms of accuracy, speed, memory efficiency and scalability. We make four major conclusions based on our experimental analyses of the whole pipeline. First, the basecalling tools with higher accuracy and performance, like Scrappie, can overcome the major drawback of nanopore sequencing technology, *i.e.*, high error rates. Second, the read-to-read overlap finding tools, Minimap and GraphMap, perform similarly in terms of accuracy. However, Minimap performs better than GraphMap in terms of speed and memory usage by storing only minimizers instead of all $k$-mers, and GraphMap is not scalable when running on machines with relatively small memories. Third, the fast but less accurate assembler Miniasm can be used for a very fast initial assembly, and further polishing can be applied on top of it to increase the accuracy of the final assembly. Fourth, a state-of-the-art polishing tool, Racon, generates high-quality consensus sequences while providing a significant speedup over another polishing tool, Nanopolish.

We hope and believe that our observations and analyses will guide researchers and practitioners to make conscious and effective choices while deciding between different tools for each step of the genome assembly pipeline using long reads. We also hope that the bottlenecks or the effects of design choices we have found and exposed can help developers in building new tools or improving the current ones. Based on our analysis and recommendations, we also show that we need high-performance, memory-efficient, low-power, and scalable designs for genome sequence analysis in order to exploit the advantages that genome sequencing provides.

# Chapter 4

# GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis

Read mapping is one of the first key steps in genome sequence analysis. For both short and long reads, *multiple* steps of read mapping must account for the sequencing errors, and for the differences caused by genetic mutations and variations. These errors and differences take the form of base insertions, deletions, and/or substitutions [224, 311, 287, 317, 216, 302]. As a result, read mapping must perform *approximate* (or *fuzzy*) *string matching* (ASM). Several algorithms exist for ASM, but state-of-the-art read mapping tools typically make use of an expensive dynamic programming based algorithm [287, 179, 225] that scales quadratically in both execution time and required storage. This ASM

algorithm has been shown to be the major bottleneck in read mapping [18, 301, 95, 16, 119, 217, 131]. Unfortunately, as sequencing technologies advance, the growth in the rate that sequencing devices generate reads is far outpacing the corresponding growth in computational power [52, 16], placing greater pressure on the ASM bottleneck. Beyond read mapping, ASM is a key technique for other bioinformatics problems such as whole genome alignment (WGA) [73, 172, 43, 125, 275, 44, 75, 300, 187, 203, 182] and multiple sequence alignment (MSA) [271, 49, 246, 123, 188, 227, 177, 226, 80], where two or more whole genomes, or regions of multiple genomes (from the same or different species), are compared to determine their similarity for predicting evolutionary relationships or finding common regions (e.g., genes). Thus, there is a pressing need to develop techniques for genome sequence analysis that provide fast and efficient ASM.

In this work, we propose *GenASM*, an ASM acceleration framework for genome sequence analysis. Our goal is to design a fast, efficient, and flexible framework for both short and long reads, which can be used to accelerate *multiple steps* of the genome sequence analysis pipeline. To avoid implementing more complex hardware for the dynamic programming based algorithm [86, 158, 301, 117, 35, 155, 267, 53], we base GenASM upon the *Bitap* algorithm [34, 317]. Bitap uses only fast and simple bitwise operations to perform approximate string matching, making it amenable to efficient hardware acceleration. To our knowledge, GenASM is the first work that enhances and accelerates Bitap.

## 4.1 Approximate String Matching (ASM)

The goal of approximate string matching [224] is to detect the differences and similarities between two sequences. Given a query read sequence $Q$=[$q_1 q_2 \ldots q_m$], a reference text sequence $T$=[$t_1 t_2 \ldots t_n$] (where $m = |Q|$, $n = |T|$, $n \geq m$), and an edit distance threshold $E$, the approximate string matching problem is to identify a set of approximate matches of $Q$ in $T$ (allowing for at most $E$ differences). The differences between two sequences of the same species can result from sequencing errors [94, 26] and/or genetic variations [88, 13].

Reads are prone to sequencing errors, which account for about 0.1% of the length of short reads [106, 256, 108] and 10–15% of the length of long reads [151, 312, 30, 304].

The differences, known as *edits*, can be classified as *substitutions*, *deletions*, or *insertions* in one or both sequences [179]. Figure 4-1 shows each possible kind of edit. In ASM, to detect a deleted character or an inserted character, we need to examine all possible *prefixes* (i.e., substrings that include the first character of the string) or *suffixes* (i.e., substrings that include the last character of the string) of the two input sequences, and keep track of the pairs of prefixes or suffixes that provide the minimum number of edits.


**Figure 4-1: Three types of errors (i.e., edits).**

Approximate string matching is needed not only to determine the minimum number of edits between two genomic sequences, but also to provide the location and type of each edit. As two sequences could have a large number of different possible arrangements of the edit operations and matches (and hence different *alignments*), the approximate string matching algorithm usually involves a traceback step. The alignment score is the sum of all edit penalties and match scores along the alignment, as defined by a user-specified scoring function. This step finds the *optimal alignment* as the combination of edit operations to build up the highest alignment score.

Approximate string matching is typically implemented as a dynamic programming based algorithm. Existing implementations, such as Levenshtein distance [179], Smith-Waterman [287], and Needleman-Wunsch [225], have quadratic time and space complexity (i.e., $O(m \times n)$ between two sequences with lengths $m$ and $n$). Therefore, it is desirable to find lower-complexity algorithms for ASM.

## 4.2 Bitap Algorithm

One candidate to replace dynamic programming based algorithms for ASM is the *Bitap* algorithm [34, 317]. Bitap tackles the problem of computing the minimum edit distance between a reference text (e.g., reference genome) and a query pattern (e.g., read) with a maximum of $k$ many errors. When $k$ is 0, the algorithm finds the exact matches.

Algorithm 1 shows the *Bitap* algorithm and Figure 4-2 shows an example for the execution of the algorithm. The algorithm starts with a pre-processing procedure (Line 4 in Algorithm 1; ⓪ in Figure 4-2) that converts the query pattern into $m$-sized pattern bitmasks, *PM*. We generate one pattern bitmask for each character in the alphabet. Since 0 means match in the Bitap algorithm, we set $PM[a][i] = 0$ when $pattern[i] = a$, where $a$ is a character from the alphabet (e.g., A, C, G, T). These pattern bitmasks help us to represent the query pattern in a binary format. After the bitmasks are prepared for each character, every bit of all status bitvectors ($R[d]$, where $d$ is in range $[0, k]$) is initialized to 1 (Lines 5–6 in Algorithm 1; ⓪ in Figure 4-2). Each $R[d]$ bitvector at text iteration $i$ holds the partial match information between $text[i : (n-1)]$ (Line 8) and the query with maximum of $d$ errors. Since at the beginning of the execution there are no matches, we initialize all status bitvectors with 1s. The status bitvectors of the previous iteration with edit distance $d$ is kept in $oldR[d]$ (Lines 10–11) to take partial matches into consideration in the next iterations.

The algorithm examines each text character one by one, one per iteration. At each text iteration (①–⑤), the pattern bitmask of the current text character (*PM*) is retrieved (Line 12). After the status bitvector for exact match is computed ($R[0]$; Line 13), the status bitvectors for each distance ($R[d]$; $d = 1...k$) are computed using the rules in Lines 15–19. For a distance $d$, three intermediate bitvectors for the error cases (one each for deletion, insertion, substitution; D/I/S in Figure 4-2) are calculated by using $oldR[d-1]$ or $R[d-1]$, since a new error is being added (i.e., the distance is increasing by 1), while the intermediate bitvector for the match case (M) is calculated using $oldR[d]$. For a deletion (Line 15), we are

**Algorithm 1** Bitap Algorithm

**Inputs:** `text` (reference), `pattern` (query), `k` (edit distance threshold)
**Outputs:** `startLoc` (matching location), `editDist` (minimum edit distance)

```
1: n ← length of reference text
2: m ← length of query pattern
3: procedure PRE-PROCESSING
4:     PM ←generatePatternBitmaskACGT(pattern)        ▷ pre-process the pattern
5:     for d in 0:k do
6:         R[d] ← 111..111                            ▷ initialize R bitvectors to 1s
7: procedure EDIT DISTANCE CALCULATION
8:     for i in (n-1):-1:0 do                         ▷ iterate over each text character
9:         curChar ← text[i]
10:        for d in 0:k do
11:            oldR[d] ← R[d]                         ▷ copy previous iterations' bitvectors as oldR
12:        curPM ← PM[curChar]                        ▷ retrieve the pattern bitmask
13:        R[0] ← (oldR[0]<<1) | curPM                ▷ status bitvector for exact match
14:        for d in 1:k do                            ▷ iterate over each edit distance
15:            deletion (D) ← oldR[d-1]
16:            substitution (S) ← (oldR[d-1]<<1)
17:            insertion (I) ← (R[d-1]<<1)
18:            match (M) ← (oldR[d]<<1) | curPM
19:            R[d] ← D & S & I & M                   ▷ status bitvector for d errors
20:        if MSB of R[d] == 0, where 0 ≤ d ≤ k then  ▷ check if MSB is 0
21:            startLoc ← i                           ▷ matching location
22:            editDist ← d                           ▷ found minimum edit distance
```

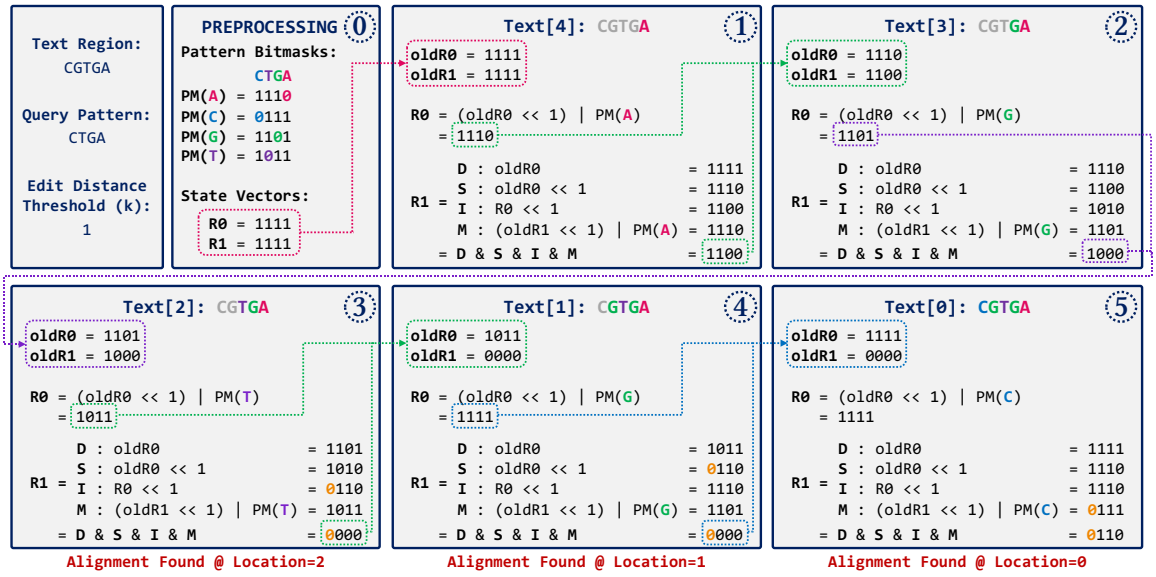



**Figure 4-2: Example for the Bitap algorithm.**

looking for a string match if the current pattern character is missing, so we copy the partial match information of the previous character ($oldR[d-1]$; consuming a text character) *without* any shifting (*not* consuming a pattern character) to serve as the deletion bitvector (labeled as $D$ of $R1$ bitvectors in ①–⑤). For a substitution (Line 16), we are looking for a string match if the current pattern character and the current text character do not match, so we take the partial match information of the previous character ($oldR[d-1]$; consuming a text character) and shift it left by one (consuming a pattern character) before saving it as the substitution bitvector (labeled as $S$ of $R1$ bitvectors in ①–⑤). For an insertion (Line 17), we are looking for a string match if the current text character is missing, so we copy the partial match information of the *current* character ($R[d-1]$; *not* consuming a text character) and shift it left by one (consuming a pattern character) before saving it as the insertion bitvector (labeled as $I$ of $R1$ bitvectors in ①–⑤). For a match (Line 18), we are looking for a string match only if the current pattern character matches the current text character, so we take the partial match information of the previous character ($oldR[d]$; consuming a text character but *not* increasing the edit distance), shift it left by one (consuming a pattern character), and perform an OR operation with the pattern bitmask of the current text character (*curPM*; comparing the text character and the pattern character) before saving the result as the match bitvector (labeled as $R0$ bitvectors and $M$ of $R1$ bitvectors in ①–⑤).

After computing all four intermediate bitvectors, in order to take all possible partial matches into consideration, we perform an AND operation (Line 19) with these four bitvectors to preserve all 0s that exist in any of them (i.e., all potential locations for a string match with an edit distance of $d$ up to this point). We save the ANDed result as the $R[d]$ status bitvector for the current iteration. This process is repeated for each potential edit distance value from 0 to $k$. If the most significant bit of the $R[d]$ bitvector becomes 0 (Lines 20–22), then there is a match starting at position $i$ of the text with an edit distance $d$ (as shown in ③–⑤). The traversal of the text then continues until all possible text positions are examined.

## 4.3 Motivation and Goals

Although the Bitap algorithm is highly suitable for hardware acceleration due to the simple nature of its bitwise operations, we find that it has five limitations that hinder its applicability and efficient hardware acceleration for genome analysis. In this section, we discuss each of these limitations. In order to overcome these limitations and design an effective and efficient accelerator, we find that we need to both (1) modify and extend the Bitap algorithm and (2) develop specialized hardware that can exploit the new opportunities that our algorithmic modifications provide.

### 4.3.1 Limitations of Bitap on Existing Systems

**No Support for Long Reads.** In state-of-the-art implementations of Bitap, the query length is limited by the word size of the machine running the algorithm. This is due to (1) the fact that the bitvector length must be equal to the query length, and (2) the need to perform bitwise operations on the bitvectors. By limiting the bitvector length to a word, each bitwise operation can be done using a single CPU instruction. Unfortunately, the lack of multi-word queries prevents these implementations from working for long reads, whose lengths are on the order of thousands to millions of base pairs (which require thousands of bits to store).

**Data Dependency Between Iterations.** As we show in Section 4.2, the computed bitvectors at each text iteration (i.e., R[d]) of the Bitap algorithm depend on the bitvectors computed in the previous text iteration (i.e., oldR[d-1] and oldR[d]; Lines 11, 13, 15, 16, and 18 of Algorithm 1). Furthermore, for each text character, there is an inner loop that iterates for the maximum edit distance number of iterations (Line 14). The bitvectors computed in each of these inner iterations (i.e., R[d]) are also dependent on the previous inner iteration's computed bitvectors (i.e., R[d-1]; Line 17). This two-level data dependency forces the consecutive iterations to take place sequentially.

**No Support for Traceback.** Although the baseline Bitap algorithm can find possible matching locations of each query read within the reference text, this covers only the first step of approximate string matching required for genome sequence analysis. Since there could be multiple different alignments between the read and the reference, the traceback operation [215, 110, 109, 207, 310, 22, 89, 287, 311, 302] is needed to find the *optimal alignment*, which is the alignment with the minimum edit distance (or with the highest score based on a user-defined scoring function). However, Bitap does not include any such support for optimal alignment identification.

**Limited Compute Parallelism.** Even after we solve the algorithmic limitations of Bitap, we find that we cannot extract significant performance benefits with just algorithmic enhancements alone. For example, while Bitap iterates over each character of the input text sequentially (Line 8), we can enable *text-level parallelism* to improve its performance (Section 4.5). However, the achievable level of parallelism is limited by the number of compute units in existing systems. For example, our studies show that Bitap is bottlenecked by computation on CPUs, since the working set fits within the private caches but the limited number of cores prevents the further speedup of the algorithm.

**Limited Memory Bandwidth.** We would expect that a GPU, which has thousands of compute units, can overcome the limited compute parallelism issues that CPUs experience. However, we find that a GPU implementation of the Bitap algorithm suffers from the limited amount of memory bandwidth available for each GPU thread. Even when we run a CUDA implementation of the baseline Bitap algorithm [184], whose bandwidth requirements are significantly lower than our modified algorithm, the limited memory bandwidth bottlenecks the algorithm's performance. We find that the bottleneck is exacerbated after the number of threads per block reaches 32, as Bitap becomes shared cache-bound (i.e., on-GPU L2 cache-bound). The small number of registers becomes insufficient to hold the intermediate data required for Bitap execution. Furthermore, when the working set of a thread does not fit within the private memory of the thread, destructive interference between threads while accessing the shared memory creates bottlenecks in the algorithm on GPUs. We

expect these issues to worsen when we implement traceback, which requires significantly higher bandwidth than Bitap.

### 4.3.2 Our Goal

Our goal in this work is to overcome these limitations and use Bitap in a fast, efficient, and flexible ASM framework for both short and long reads. We find that this goal cannot be achieved by modifying only the algorithm or only the hardware. We design *GenASM*, the first ASM acceleration framework for genome sequence analysis. Through careful modification and co-design of the enhanced Bitap algorithm and hardware, GenASM aims to successfully replace the expensive dynamic programming based algorithm used for ASM in genomics with the efficient bitwise-operation-based Bitap algorithm, which can accelerate *multiple steps* of genome sequence analysis.

## 4.4 GenASM: A High-Level Overview

In GenASM, we *co-design* our modified Bitap algorithm for distance calculation (DC) and our new Bitap-compatible traceback (TB) algorithm with an area- and power-efficient hardware accelerator. GenASM consists of two components, as shown in Figure 4-3: (1) GenASM-DC (Section 4.5), which for each read generates the bitvectors and performs the minimum edit distance calculation (DC); and (2) GenASM-TB (Section 4.6), which uses the bitvectors to perform traceback (TB) and find the optimal alignment. GenASM is a flexible framework that can be used for different use cases (Section 4.8).

GenASM execution starts when the host CPU issues a task to GenASM with the reference and the query sequences' locations (❶ in Figure 4-3). GenASM-DC reads the corresponding reference text region and the query pattern from the memory. GenASM-DC then writes these to its dedicated SRAM, which we call DC-SRAM (❷). After that, GenASM-DC divides the reference text (e.g., reference genome) and query pattern (e.g., read) into multiple overlapping windows (❸), and for each *sub-text* (i.e., the portion of the reference

text in one window) and *sub-pattern* (i.e., the portion of the query pattern in one window), GenASM-DC searches for the sub-pattern within the sub-text and generates the bitvectors (❹). Each processing element (PE) of GenASM-DC writes the generated bitvectors to its own dedicated SRAM, which we call TB-SRAM (❺). Once GenASM-DC completes its search for the current window, GenASM-TB starts reading the stored bitvectors from TB-SRAMs (❻) and generates the window's traceback output (❼). Once GenASM-TB generates this output, GenASM computes the next window and repeats Steps ❸–❼ until all windows are completed.

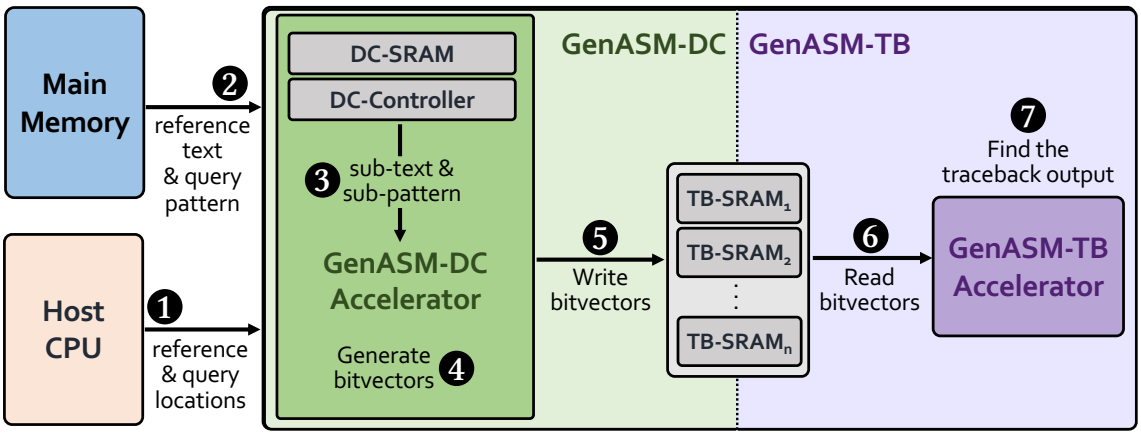



**Figure 4-3: Overview of GenASM.**

Our hardware accelerators are designed to maximize parallelism and minimize memory footprint. Our modified GenASM-DC algorithm is highly parallelizable, and performs only simple and regular bitwise operations, so we implement the GenASM-DC accelerator as a systolic array based accelerator. GenASM-TB accelerator requires simple logic operations to perform the TB-SRAM accesses and the required control flow to complete the traceback operation. Both of our hardware accelerators are highly efficient in terms of area and power. We discuss them in detail in Section 4.7.

## 4.5 GenASM-DC Algorithm

We modify the baseline Bitap algorithm (Section 4.2) to (1) enable efficient alignment of long reads, (2) remove the data dependency between the iterations, and (3) provide parallelism for the large number of iterations.

**Long Read Support.** The GenASM-DC algorithm overcomes the word-length limit of Bitap (Section 4.3.1) by storing the bitvectors in multiple words when the query is longer than the word size. Although this modification leads to additional computation when performing shifts, it helps GenASM to support both short and long reads. When shifting word $i$ of a multi-word bitvector, the bit shifted out (MSB) of word $i-1$ needs to be stored separately before performing the shift on word $i-1$. Then, that saved bit needs to be loaded as the least significant bit (LSB) of word $i$ when the shift occurs. This causes the complexity of the algorithm to be $\lceil \frac{m}{w} \rceil \times n \times k$, where $m$ is the query length, $w$ is the word size, $n$ is the text length, and $k$ is the edit distance.

**Loop Dependency Removal.** In order to solve the two-level data dependency limitation of the baseline Bitap algorithm (Section 4.3.1), GenASM-DC performs loop unrolling and enables computing non-neighbor (i.e., independent) bitvectors in parallel. Figure 4-4 shows an example for unrolling with four threads for text characters T0–T3 and status bitvectors R0–R7. For the iteration where $R[d]$ represents T2–R2 (i.e., the target cell shaded in dark red), $R[d-1]$ refers to T2–R1, $oldR[d-1]$ refers to T1–R1, and $oldR[d]$ refers to T1–R2 (i.e., cells T2–R2 is dependent on, shaded in light red). Based on this example, T2–R2 depends on T1–R2, T2–R1, and T1–R1, but it does not depend on T3–R1, T1–R3, or T0–R4. Thus, these independent bitvectors can be computed in parallel without waiting for one another.

**Text-Level Parallelism.** In addition to the parallelism enabled by removing the loop dependencies, we enable GenASM-DC algorithm to exploit text-level parallelism. This parallelism is enabled by dividing the text into overlapping sub-texts and searching the query in each of these sub-texts in parallel. The overlap ensures that we do not miss any

possible match that may fall around the edges of a sub-text. To guarantee this, the overlap needs to be of length $m + k$, where $m$ is the query length and $k$ is the edit distance threshold.

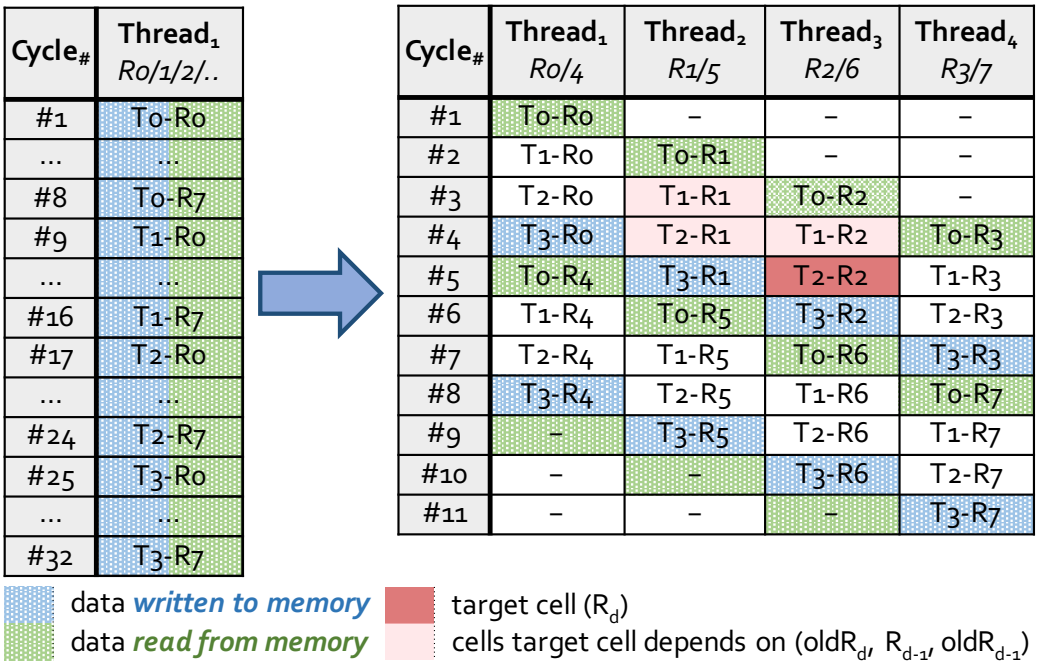



**Figure 4-4: Loop unrolling in GenASM-DC.**

## 4.6 GenASM-TB Algorithm

After finding the matching location of the text and the edit distance with GenASM-DC, our new traceback [215, 110, 109, 207, 310, 22, 89, 287, 311, 302] algorithm, GenASM-TB, finds the sequence of matches, substitutions, insertions and deletions, along with their positions (i.e., CIGAR string) for the matched region (i.e., the text region that starts from the location reported by GenASM-DC and has a length of $m + k$), and reports the optimal alignment. Traceback execution (1) starts from the first character of the matched region between the reference text and query pattern, (2) examines each character and decides which of the four operations should be picked in each iteration, and (3) ends when we reach the last character of the matched region. GenASM-TB uses the intermediate bitvectors generated and saved in each iteration of the GenASM-DC algorithm (i.e., match,

substitution, deletion and insertion bitvectors generated in Lines 15–18 in Algorithm 1). After a value 0 is found at the MSB of one of the $R[d]$ bitvectors (i.e., a string match is found with $d$ errors), GenASM-TB walks through the bitvectors back to the LSB, following a chain of 0s (which indicate matches at each location) and reverting the bitwise operations. At each position, based on which of the four bitvectors holds a value 0 in each iteration (starting with an MSB with a 0 and ending with an LSB with a 0), the sequence of matches, substitutions, insertions and deletions (i.e., traceback output) is found for each position of the corresponding alignment found by GenASM-DC. Unlike GenASM-DC, GenASM-TB has an irregular control flow within the stored intermediate bitvectors, which depends on the text and the pattern.

Algorithm 2 shows the *GenASM-TB* algorithm and Figure 4-5 shows an example for the execution of the algorithm for each of the alignments found in ③–⑤ of Figure 4-2. In Figure 4-5, $<x, y, z>$ stands for `patternI`, `textI` and `curError`, respectively (Lines 6–8 in Algorithm 2). `patternI` represents the position of a 0 currently being processed within a given bitvector (i.e., pattern index), `textI` represents the outer loop iteration index (i.e., text index; $i$ in Algorithm 1), and `curError` represents the inner loop iteration index (i.e., number of remaining errors; $d$ in Algorithm 1).

When we find a 0 at `match[textI][curError][patternI]` (i.e., a *match (M)* is found for the current position; Line 17), one character each from both text and query is consumed, but the number of remaining errors stays the same. Thus, the pointer moves to the next text character (as the text character is consumed), and the 0 currently being processed (highlighted with orange color in Figure 4-5) is right-shifted by one (as the query character is also consumed). In other words, `textI` is incremented (Line 28), `patternI` is decremented (Line 30), but `curError` remains the same. Thus, $<x, y, z>$ becomes $<x - 1, y + 1, z>$ after we find a match. For example, in Figure 4-5a, for Text[0], we have $<3, 0, 1>$ for the indices, and after the match is found, at the next position (Text[1]), we have $<2, 1, 1>$.

**Algorithm 2** GenASM-TB Algorithm

**Inputs:** `text` (reference), `n`, `pattern` (query), `m`, `W` (window size), `O` (overlap size)
**Output:** `CIGAR` (complete traceback output)

```
1: <curPattern,curText> ← <0,0>          ▷ start positions of sub-pattern and sub-text
2: while (curPattern < m) & (curText < n) do
3:     sub-pattern ← pattern[curPattern:(curPattern+W)]
4:     sub-text ← text[curText:(curText+W)]
5:     intermediate bitvectors ← GenASM-DC(sub-pattern,sub-text,W)
6:     patternI ← W-1                    ▷ pattern index (position of 0 being processed)
7:     textI ← 0                                                    ▷ text index
8:     curError ← editDist from GenASM-DC          ▷ number of remaining errors
9:     <patternConsumed,textConsumed> ← <0,0>
10:    prev ← ""                                   ▷ output of previous TB iteration
11:    while textConsumed<(W-O) & patternConsumed<(W-O) do
12:        status ← 0
13:        if ins[textI][curError][patternI]=0 & prev='I' then
14:            status ← 3; add "I" to CIGAR;                  ▷ insertion-extend
15:        else if del[textI][curError][patternI]=0 & prev='D' then
16:            status ← 4; add "D" to CIGAR;                  ▷ deletion-extend
17:        else if match[textI][curError][patternI]=0 then
18:            status ← 1; add "M" to CIGAR; prev ← "M"               ▷ match
19:        else if subs[textI][curError][patternI]=0 then
20:            status ← 2; add "S" to CIGAR; prev ← "S"           ▷ substitution
21:        else if ins[textI][curError][patternI]=0 then
22:            status ← 3; add "I" to CIGAR; prev ← "I"       ▷ insertion-open
23:        else if del[textI][curError][patternI]=0 then
24:            status ← 4; add "D" to CIGAR; prev ← "D"       ▷ deletion-open
25:        if (status > 1) then
26:            curError--                                              ▷ S, D, or I
27:        if (status > 0) && (status != 3) then
28:            textI++; textConsumed++                                 ▷ M, S, or D
29:        if (status > 0) && (status != 4) then
30:            patternI--; patternConsumed++                           ▷ M, S, or I
31:    curPattern ← curPattern+patternConsumed
32:    curText ← curText+textConsumed
```

When we find a 0 at `subs[textI][curError][patternI]` (i.e., a *substitution (S)* is found for the current position; Line 19), one character each from both text and query is consumed, and the number of remaining errors is decremented (Line 26). Thus, $<x, y, z>$ becomes $<x-1, y+1, z-1>$ after we find a substitution (e.g., Text[1] in Figure 4-5b).

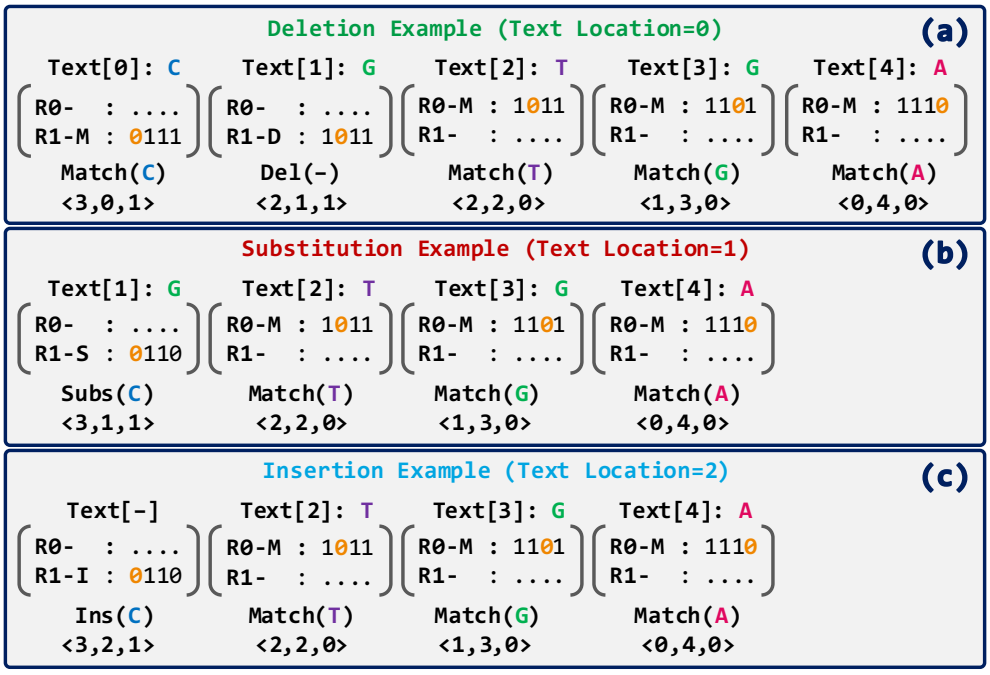



**Figure 4-5: Traceback example with GenASM-TB algorithm.**

When we find a 0 at `ins[textI][curError][patternI]` (i.e., an *insertion (I)* is found for the current position; Lines 13 and 21), the inserted character does not appear in the text, and only a character from the pattern is consumed. The 0 currently being processed is right-shifted by one, but the text pointer remains the same, and the number of remaining errors is decremented. Thus, $<x, y, z>$ becomes $<x - 1, y, z - 1>$ after we find an insertion (e.g., Text[−] in Figure 4-5c).

When we find a 0 at `del[textI][curError][patternI]` (i.e., a *deletion (D)* is found for the current position; Lines 15 and 23), the deleted character does not appear in the pattern, and only a character from the text is consumed. The 0 currently being processed is not right-shifted, but the pointer moves to the next text character, and the number of remaining errors is also decremented. Thus, $<x, y, z>$ becomes $<x, y + 1, z - 1>$ after we find an insertion (e.g., Text[1] in Figure 4-5a).

**Divide-and-Conquer Approach.** Since GenASM-DC stores all of the intermediate bitvectors, in the worst case, the length of the text region that the query pattern maps to can be $m + k$, assuming all of the errors are deletions from the pattern. Since we need to

store all of the bitvectors for $m + k$ characters, and compute $4 \times k$ many bitvectors within each text iteration (each $m$ bits long), for long reads with high error rates, the memory requirement becomes ~80GB, when m is 10,000 and k is 1,500.

In order to decrease the memory footprint of the algorithm, we follow two key ideas. First, we apply a divide-and-conquer approach (similar to the tiling approach of Darwin's alignment accelerator, GACT [301]). Instead of storing all of the bitvectors for $m + k$ text characters, we divide the text and pattern into overlapping windows (i.e., sub-text and sub-pattern; Lines 3–4 in Algorithm 2) and perform the traceback computation for each window. After all of the windows' partial traceback outputs are generated, we merge them to find the complete traceback output. This approach helps us to decrease the memory footprint from $((m + k) \times 4 \times k \times m)$ bits to $(W \times 4 \times W \times W)$ bits, where $W$ is the window size. This divide-and-conquer approach also helps us to reduce the complexity of the bitvector generation step (Section 4.5) from $\lceil \frac{m}{w} \rceil \times n \times k$ to $\lceil \frac{W}{w} \rceil \times W \times W$. Second, instead of storing all 4 bitvectors (i.e., match, substitution, insertion, deletion) separately, we only need to store bitvectors for match, insertion, and deletion, as the substitution bitvector can be obtained easily by left-shifting the deletion bitvector by 1 (Line 16 in Algorithm 1). This modification helps us to decrease the required write bandwidth and the memory footprint to $(W \times 3 \times W \times W)$ bits.

GenASM-TB restricts the number of consumed characters from the text or the pattern to `W-O` (Line 11 in Algorithm 2) to ensure that consecutive windows share $O$ characters (i.e., overlap size), and thus, the traceback output can be generated accurately. The sub-text and the sub-pattern corresponding to each window are found using the number of consumed text characters (`textConsumed`) and the number of consumed pattern characters (`patternConsumed`) in the previous window (Lines 31–32 in Algorithm 2).

**Partial Support for Complex Scoring Schemes.** We extend the GenASM-TB algorithm to provide partial support (Section 4.10.2) for non-unit costs for different edits and the affine gap penalty model [109, 207, 310, 22]. By changing the order in which different traceback cases are checked in Lines 13–24 in Algorithm 2, we can support different types

of scoring schemes. For example, in order to mimic the behavior of the affine gap penalty model, we check whether the traceback output that has been chosen for the previous position (i.e., `prev`) is an insertion or a deletion. If the previous edit is a gap (insertion or deletion), and there is a 0 at the current position of the insertion or deletion bitvector (Lines 13 and 15 in Algorithm 2), then we prioritize extending this previously opened gap, and choose insertion-extend or deletion-extend as the current position's traceback output, depending on the type of the previous gap. As another example, in order to mimic the behavior of non-unit costs for different edits, we can simply sort three error cases (substitution, insertion-open, deletion-open) from the lowest penalty to the highest penalty. If substitutions have a lower penalty than gap openings, the order shown in Algorithm 2 should remain the same. However, if substitutions have a greater penalty than gap openings, we should check for the substitution case after checking the insertion-open and deletion-open cases (i.e., Lines 19–20 should come after Line 24 in Algorithm 2).

## 4.7 GenASM Hardware Design

**GenASM-DC Hardware.** We implement GenASM-DC as a linear cyclic systolic array [171, 170] based accelerator. The accelerator is optimized to reduce both the memory bandwidth and the memory footprint. Feedback logic enabling cyclic systolic behavior allows us to fix the required number of memory ports [170] and to reduce memory footprint.

A GenASM-DC accelerator consists of a processing block (PB; Figure 4-6a) along with a control and memory management logic. A PB consists of multiple processing elements (PEs). Each PE contains a single processing core (PC; Figure 4-6b) and flip-flop-based storage logic. The PC is the primary compute unit, and implements Lines 15–19 of Algorithm 1 to perform the approximate string matching for a $w$-bit query pattern. The number of PEs in a PB is based on compute, area, memory bandwidth and power requirements. This block also implements the logic to load data from outside of the array (i.e., DC-SRAM; Figure 4-6a) or internally for cyclic operations.

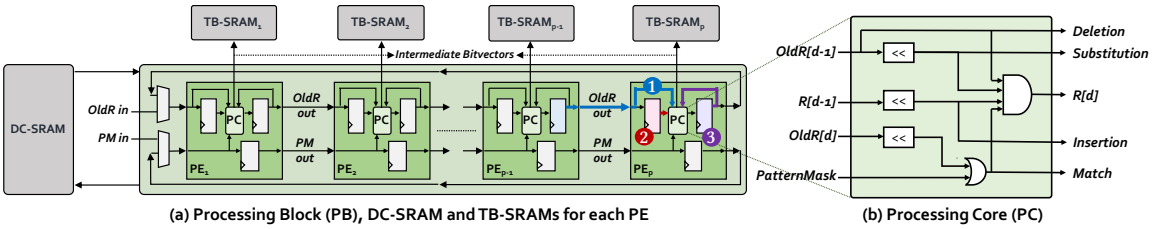



**Figure 4-6: Hardware design of GenASM-DC.**

GenASM-DC uses two types of SRAM buffers (Figure 4-6a): (1) DC-SRAM, which stores the reference text, the pattern bitmasks for the query read, and the intermediate data generated from PEs (i.e., *oldR* values and MSBs required for shifts; Section 4.5); and (2) TB-SRAM, which stores the intermediate bitvectors from GenASM-DC for later use by GenASM-TB. For a 64-PE configuration with 64 bits of processing per PE, and for the case where we have a long (10Kbp) read[1] with a high error rate (15%) and a corresponding text region of 11.5Kbp, GenASM-DC requires a total of 8KB DC-SRAM storage. For each PE, we have a dedicated TB-SRAM, which stores the match, insertion and deletion bitvectors generated by the corresponding PE. For the same configuration of GenASM-DC, each PE requires a total of 1.5KB TB-SRAM storage, with a single R/W port. In each cycle, 192 bits of data (24B) is written to each TB-SRAM by each PE.

When each thread (i.e., each column) in Figure 4-4 is mapped to a PE, GenASM-DC coordinates the data dependencies across DC iterations, with the help of two flip-flops in each PE. For example, T2–R2 in Figure 4-4 is generated by $PE_x$ in $Cycle_y$, and is mapped to $R[d]$. In order to generate T2–R2, T2–R1 (which maps to $R[d-1]$) needs to be generated by $PE_{x-1}$ in $Cycle_{y-1}$ (❶ in Figure 4-6), T1–R1 (which maps to $oldR[d-1]$) needs to be generated by $PE_{x-1}$ in $Cycle_{y-2}$ (❷), and T1–R2 (which maps to $oldR[d]$) needs to be generated by $PE_x$ in $Cycle_{y-1}$ (❸), where $x$ is the PE index and $y$ is the cycle index. With this dependency-aware mapping, regardless of the number of instantiated PEs, we can successfully limit DC-SRAM traffic for a single PB to only one read and one write per cycle.

[1] Although we use 10Kbp-long reads in our analysis (Section 4.9), GenASM does *not* have any limitation on the length of reads as a result of our divide-and-conquer approach (Section 4.6).

**GenASM-TB Hardware.** After GenASM-DC finishes writing all of the intermediate bitvectors to TB-SRAMs, GenASM-TB reads them by following an irregular control flow, which depends on the text and the pattern to find the optimal alignment (by implementing Algorithm 2).

In our GenASM configuration, where we have 64 PEs and 64 bits per PE in a GenASM-DC accelerator, and the window size ($W$) is 64 (Section 4.6), we have one 1.5KB TB-SRAM (which fits our 24B/cycle $\times$ 64 cycles/window output storage requirement) for each of the 64 PEs. As Figure 4-7 shows, a single GenASM-TB accelerator is connected to all of these 64 TB-SRAMs (96KB, in total). In each GenASM-TB cycle, we read from only one TB-SRAM. `curError` provides the index of the TB-SRAM that we read from; `textI` provides the starting index within this TB-SRAM, which we read the next set of bitvectors from; and `patternI` provides the position of the 0 being processed (Algorithm 2).

We implement the GenASM-TB hardware using very simple logic (Figure 4-7), which ❶ reads the bitvectors from one of the TB-SRAMs using the computed address, ❷ performs the required bitwise comparisons to find the CIGAR character for the current position, and ❸ computes the next TB-SRAM address to read the new set of bitvectors. After GenASM-TB finds the complete CIGAR string, it writes the output to main memory and completes its execution.

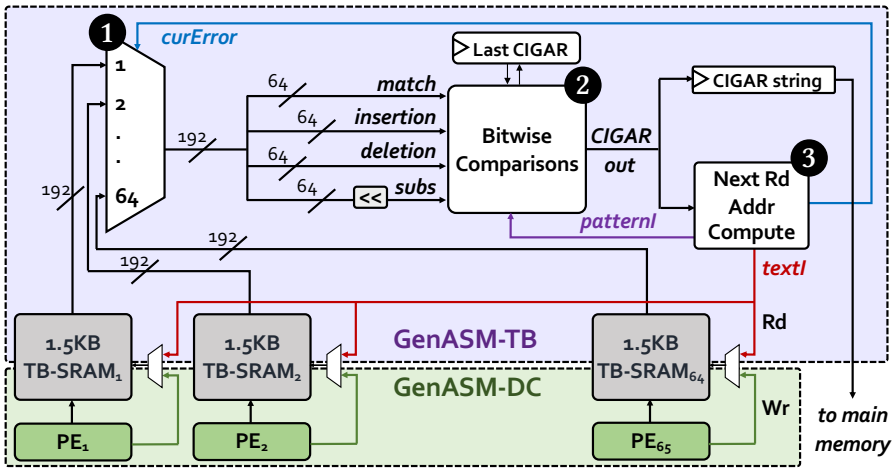



**Figure 4-7: Hardware design of GenASM-TB.**

**Overall System.** We design our system to take advantage of modern 3D-stacked memory systems [104, 167], such as the Hybrid Memory Cube (HMC) [133] or High-Bandwidth Memory (HBM) [154, 178]. Such memories are made up of multiple layers of DRAM arrays that are stacked vertically in a single package. These layers are connected via high-bandwidth links called *through-silicon vias* (TSVs) that provide lower-latency and more energy-efficient data access to the layers than the external DRAM I/O pins [68, 178]. Memories such as HMC and HBM include a dedicated *logic layer* that connects to the TSVs and allows processing elements to be implemented in memory to exploit the efficient data access. Due to thermal and area constraints, only simple processing elements that execute low-complexity operations (e.g., bitwise logic, simple arithmetic, simple cores) can be included in the logic layer [37, 77, 97, 11, 10, 128, 127, 213, 38, 248, 164].

We decide to implement GenASM in the logic layer of 3D-stacked memory, for two reasons. First, we can exploit the natural subdivision within 3D-stacked memory (e.g., vaults in HMC [133], pseudo-channels in HBM [154]) to efficiently enable parallelism across multiple GenASM accelerators. This subdivision allows accelerators to work in parallel without interfering with each other. Second, we can reduce the power consumed for DRAM accesses by reducing off-chip data movement across the memory channel [213]. Both of our hardware accelerators are highly efficient in terms of area and power (Section 4.10.1), and can fit within the logic layer's constraints.

To illustrate how GenASM takes advantage of 3D-stacked memory, we discuss an example implementation of GenASM inside the logic layer of a 16GB HMC with 32 vaults [133]. Within each vault, the logic layer contains a GenASM-DC accelerator, its associated DC-SRAM (8KB), a GenASM-TB accelerator, and TB-SRAMs (64×1.5KB). Since we have small SRAM buffers for both DC and TB to exploit locality, GenASM accesses the memory and utilizes the memory bandwidth only to read the reference and the query sequences. One GenASM accelerator at each vault requires 105–142 MB/s bandwidth, thus the total bandwidth requirement of all 32 GenASM accelerators is 3.3–4.4 GB/s (which is much less than peak bandwidth provided by modern 3D-stacked memories).

## 4.8 GenASM Framework

We demonstrate the efficiency and flexibility of the GenASM acceleration framework by describing three use cases of approximate string matching in genome sequence analysis: (1) read alignment step of short and long read mapping, (2) pre-alignment filtering for short reads, and (3) edit distance calculation between any two sequences. We believe the GenASM framework can be useful for many other use cases, and we discuss some of them briefly in Section 4.11.

**Read Alignment of Short and Long Reads.** As we explain in Section 2.4, read alignment is the last step of short and long read mapping. In read alignment, all of the remaining candidate mapping regions of the reference genome and the query reads are aligned, in order to identify the mapping that yields either the lowest total number of errors (if using edit distance based scoring) or the highest score (if using a user-defined scoring function). Thus, read alignment can be a use case for approximate string matching, since errors (i.e., substitutions, insertions, deletions) should be taken into account when aligning the sequences. As part of read alignment, we also need to generate the traceback output for the best alignment between the reference region and the read.

For read alignment, the whole GenASM pipeline, as explained in Section 4.4, should be executed, including the traceback step. In general, read alignment requires more complex scoring schemes, where different types of edits have non-unit costs. Thus, GenASM-TB should be configured based on the given cost of each type of edit (Section 4.6). As GenASM framework can work with arbitrary length sequences, we can use it to accelerate both short read and long read alignment.

**Pre-Alignment Filtering for Short Reads.** In the pre-alignment filtering step of short read mapping, the candidate mapping locations, reported by the seeding step, are further filtered by using different mechanisms. Although the regions of the reference at these candidate mapping locations share common seeds with query reads, they are not necessarily *similar* sequences. To avoid examining dissimilar sequences at the downstream

computationally-expensive read alignment step, a pre-alignment filter estimates the edit distance between every read and the regions of the reference at each read's candidate mapping locations, and uses this estimation to quickly decide whether or not read alignment is needed. If the sequences are dissimilar enough, significant amount of time is saved by avoiding the expensive alignment step [18, 21, 17, 322, 321].

In pre-alignment filtering, since we only need to estimate (approximately) the edit distance and check whether it is above a user-defined threshold, GenASM-DC can be used as a pre-alignment filter. As GenASM-DC is very efficient when we have shorter sequences and a low error threshold (due to the $O(m \times n \times k)$ complexity of the underlying Bitap algorithm, where $m$ is the query length, $n$ is the reference length, and $k$ is the number of allowed errors), GenASM framework can efficiently accelerate the pre-alignment filtering step of especially short read mapping.[2]

**Edit Distance Calculation.** Edit distance, also called Levenshtein distance [179], is the minimum number of edits (i.e., substitutions, insertions and deletions) required to convert one sequence to another. Edit distance calculation is one of the fundamental operations in genomics to measure the similarity or distance between two sequences [289]. As we explain in Section 4.2, the Bitap algorithm, which is the underlying algorithm of GenASM-DC, is originally designed for edit distance calculation. Thus, GenASM framework can accelerate edit distance calculation between any two arbitrary-length genomic sequences.

Although GenASM-DC can find the edit distance by itself and traceback is optional for this use case, DC-TB interaction is required in our accelerator to exploit the efficient divide-and-conquer approach GenASM follows. Thus, GenASM-DC and GenASM-TB work together to find the minimum edit distance in a fast and memory-efficient way, but the traceback output is not generated or reported by default (though it can optionally be enabled).

[2] Although we believe that GenASM can also be used as a pre-alignment filter for long reads, we leave the evaluation of this use case for future work.

## 4.9 Evaluation Methodology

**Area and Power Analysis.** We synthesize and place & route the GenASM-DC and GenASM-TB accelerator datapaths using the Synopsys Design Compiler [4] with a typical 28nm low-power process, with memories generated using an industry-grade SRAM compiler, to analyze the accelerators' area and power. Our synthesis targets post-routing timing closure at 1GHz clock frequency. We then use an in-house cycle-accurate simulator parameterized with the synthesis and memory estimations to drive the performance and power analysis.

We evaluate a 16GB HMC-like 3D-stacked DRAM architecture, with 32 vaults [133] and 256GB/s of internal bandwidth [37, 133], and a clock frequency of 1.25GHz [133]. The amount of available area in the logic layer for GenASM is around 3.5–4.4 $mm^2$ per vault [77, 37]. The power budget of our PIM logic per vault is 312mW [77].

**Performance Model.** We build a spreadsheet-based analytical model for GenASM-DC and GenASM-TB, which considers reference genome (i.e., text) length, query read (i.e., pattern) length, maximum edit distance, window size, hardware design parameters (number of PEs, bit width of each PE) and number of vaults as input parameters and projects compute cycles, DRAM read/write bandwidth, SRAM read/write bandwidth, and memory footprint. We verify the analytically-estimated cycle counts for various PE configurations with the cycle counts collected from our RTL simulations.

**Read Alignment Comparisons.** For the read alignment use case, we compare GenASM with the read alignment steps of two commonly-used state-of-the-art read mappers: Minimap2 [182] and BWA-MEM [180], running on an Intel® Xeon® Gold 6126 CPU [146] operating at 2.60GHz, with 64GB DDR4 memory. Software baselines are run with a single thread and with 12 threads. We measure the execution time and power consumption of the alignment steps in Minimap2 and BWA-MEM. We measure the individual power consumed by each tool using Intel's PCM power utility [143].

We also compare GenASM with a state-of-the-art GPU-accelerated short read alignment tool, GASAL2 [9]. We run GASAL2 on an Nvidia Titan V GPU [233] with 12GB HBM2 memory [154]. To fully utilize the GPU, we configure the number of alignments per batch based on the GPU's number of multiprocessors and the maximum number of threads per multiprocessor, as described in the GASAL2 paper [9]. To better analyze the high parallelism that the GPU provides, we replicate our datasets to obtain datasets with 100K, 1M and 10M reference-read pairs for short reads. We run the datasets with GASAL2, and collect kernel time and average power consumption using *nvprof* [234].

We also compare GenASM with two state-of-the-art hardware-based alignment accelerators, GACT of Darwin [301] and SillaX of GenAx [95]. We synthesize and execute the open-source RTL for GACT [66]. We estimate the performance of SillaX using data reported by the original work [95].

We analyze the alignment accuracy of GenASM by comparing the alignment outputs (i.e., alignment score, edit distance, and CIGAR string) of GenASM with the alignment outputs of BWA-MEM and Minimap2, for short reads and long reads, respectively. We obtain the BWA-MEM and Minimap2 alignments by running the tools with their default settings.

**Pre-Alignment Filtering Comparisons.** We compare GenASM with Shouji [17], which is the state-of-the-art FPGA-based pre-alignment filter for short reads. For execution time and filtering accuracy analyses, we use data reported by the original work [17]. For power analysis, we report the total power consumption of Shouji using the power analysis tool in Xilinx Vivado [320], after synthesizing and implementing the open-source FPGA design of Shouji [269].

**Edit Distance Calculation Comparisons.** We compare GenASM with the state-of-the-art software-based read alignment library, Edlib [289], running on an Intel® Xeon® Gold 6126 CPU [146] operating at 2.60GHz, with 64GB DDR4 memory. Edlib uses the Myers' bitvector algorithm [216] to find the edit distance between two sequences. We use the

default global Needleman-Wunsch (NW) [225] mode of Edlib to perform our comparisons. We measure the power consumed by Edlib using Intel's PCM power utility [143].

We also compare GenASM with ASAP [35], which is the state-of-the-art FPGA-based accelerator for computing the edit distance between two short reads. We estimate the performance of ASAP using data reported by the original work [35].

**Datasets.** For the read alignment use case, we evaluate GenASM using the latest major release of the human genome assembly, GRCh38 [3]. We use the 1–22, X, and Y chromosomes by filtering the unmapped contigs, unlocalized contigs, and mitochondrial genome. Genome characters are encoded into 2-bit patterns (A = 00, C = 01, G = 10, T = 11). With this encoding, the reference genome uses 715 MB of memory.

We generate four sets of long reads (i.e., PacBio and ONT datasets) using PBSIM [236] and three sets of short reads (i.e., Illumina datasets) using Mason [126]. For the PacBio datasets, we use the default error profile for the continuous long reads (CLR) in PBSIM. For the ONT datasets, we modify the settings to match the error profile of ONT reads sequenced using R9.0 chemistry [152]. Both datasets have 240,000 reads of length 10Kbp, each simulated with 10% and 15% error rates. The Illumina datasets have 200,000 reads of length 100bp, 150bp, and 250bp, each simulated with a 5% error rate.

For the pre-alignment filtering use case, we use two datasets that Shouji [17] provides as test cases: reference-read pairs (1) of length 100bp with an edit distance threshold of 5, and (2) of length 250bp with an edit distance threshold of 15.

For the edit distance calculation use case, we use the publicly-available dataset that Edlib [289] provides. The dataset includes two real DNA sequences, which are 100Kbp and 1Mbp in length, and artificially-mutated versions of the original DNA sequences with measures of similarity ranging between 60%–99%. Evaluating this set of sequences with varying values of similarity and length enables us to demonstrate how these parameters affect performance.

## 4.10 Results

### 4.10.1 Area and Power Analysis

Table 4-1 shows the area and power breakdown of each component in GenASM, and the total area overhead and power consumption of (1) a single GenASM accelerator (in 1 vault) and (2) 32 GenASM accelerators (in 32 vaults). Both GenASM-DC and GenASM-TB operate at 1GHz.

The area overhead of one GenASM accelerator is 0.334 $mm^2$, and the power consumption of one GenASM accelerator, including the SRAM power, is 101 mW. When we compare GenASM with a single core of a modern Intel® Xeon® Gold 6126 CPU [146] (which we conservatively estimate to use 10.4 W [146] and 32.2 $mm^2$ [60] per core), we find that GenASM is significantly more efficient in terms of both area and power consumption. As we have one GenASM accelerator per vault, the total area overhead of GenASM in all 32 vaults is 10.69 $mm^2$. Similarly, the total power consumption of 32 GenASM accelerators is 3.23 W.

**Table 4-1: Area and power breakdown of GenASM.**

| Component | Area ($mm^2$) | Power (W) |
|---|---|---|
| GenASM-DC (64 PEs) | 0.049 | 0.033 |
| GenASM-TB | 0.016 | 0.004 |
| DC-SRAM (8 KB) | 0.013 | 0.009 |
| TB-SRAMs (64 x 1.5 KB) | 0.256 | 0.055 |
| **Total – 1 vault (32 vaults)** | **0.334 (10.69)** | **0.101 (3.23)** |

### 4.10.2 Use Case 1: Read Alignment

**Software Baselines (CPU).** Figure 4-8 shows the read alignment throughput (reads/sec) of GenASM and the alignment steps of BWA-MEM and Minimap2, when aligning long noisy PacBio and ONT reads against the human reference genome. When comparing with

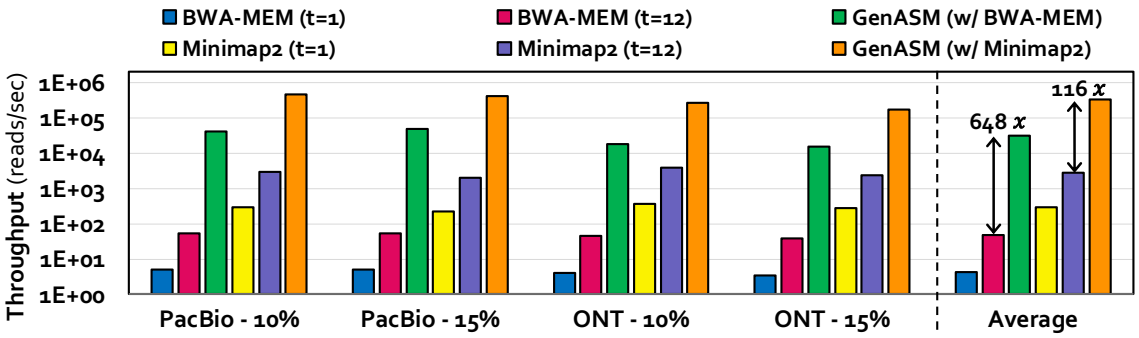



**Figure 4-8: Throughput comparison of GenASM and the alignment steps of BWA-MEM and Minimap2 for long reads.**

BWA-MEM, we run GenASM with the candidate locations reported by BWA-MEM's filtering step. Similarly, when comparing with Minimap2, we run GenASM with the candidate locations reported by Minimap2's filtering step. GenASM's throughput is determined by the throughput of the execution of GenASM-DC and GenASM-TB with window size ($W$) of 64 and overlap size ($O$) of 24.

As Figure 4-8 shows, GenASM provides (1) 7173× and 648× throughput improvement over the alignment step of BWA-MEM for its single-thread and 12-thread execution, respectively, and (2) 1126× and 116× throughput improvement over the alignment step of Minimap2 for its single-thread and 12-thread execution, respectively.

Based on our power analysis with long reads, we find that power consumption of BWA-MEM's alignment step is 58.6 W and 109.5 W, and power consumption of Minimap2's read alignment step is 59.8 W and 118.9 W for their single-thread and 12-thread executions, respectively. GenASM consumes only 3.23W, and thus reduces the power consumption of the alignment steps of BWA-MEM and Minimap2 by 18× and 19× for single-thread execution, and by 34× and 37× for 12-thread execution, respectively.

Figure 4-9 compares the read alignment throughput (reads/sec) of GenASM with that of the alignment steps of BWA-MEM and Minimap2, when aligning short Illumina reads against the human reference genome. GenASM provides (1) 1390× and 111× throughput improvement over the alignment step of BWA-MEM for its single-thread and 12-thread exe-

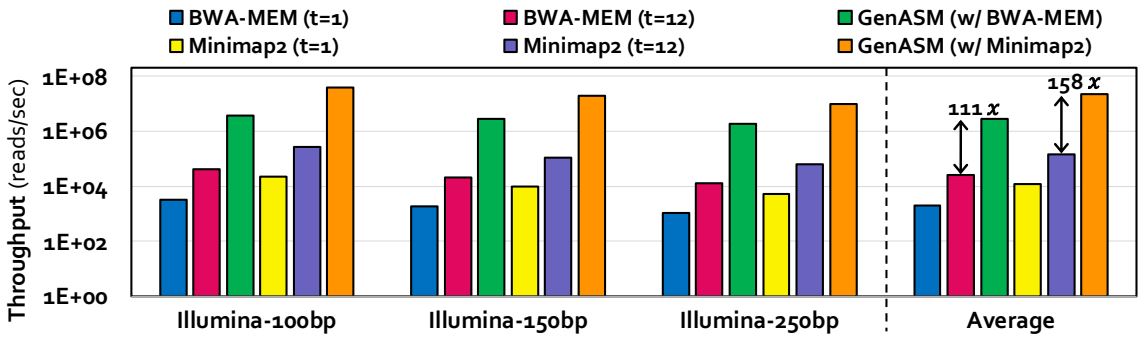



**Figure 4-9: Throughput comparison of GenASM and the alignment steps of BWA-MEM and Minimap2 for short reads.**

cution, respectively, and (2) 1839× and 158× throughput improvement over the alignment step of Minimap2 for its single-thread and 12-thread execution.

Based on our power analysis with short reads, we find that GenASM reduces the power consumption over the alignment steps of BWA-MEM and Minimap2 by 16× and 18× for single-thread execution, and by 33× and 31× for 12-thread execution, respectively.

Figure 4-10 shows the total execution time of the entire BWA-MEM and Minimap2 pipelines, along with the total execution time when the alignment steps of each pipeline are replaced by GenASM, for the three representative input datasets. As Figure 4-10 shows, GenASM provides (1) 2.4× and 1.9× speedup for Illumina reads (250bp); (2) 6.5× and 3.4× speedup for PacBio reads (15%); and (3) 4.9× and 2.1× speedup for ONT reads (15%), over the entire pipeline executions of BWA-MEM and Minimap2, respectively.

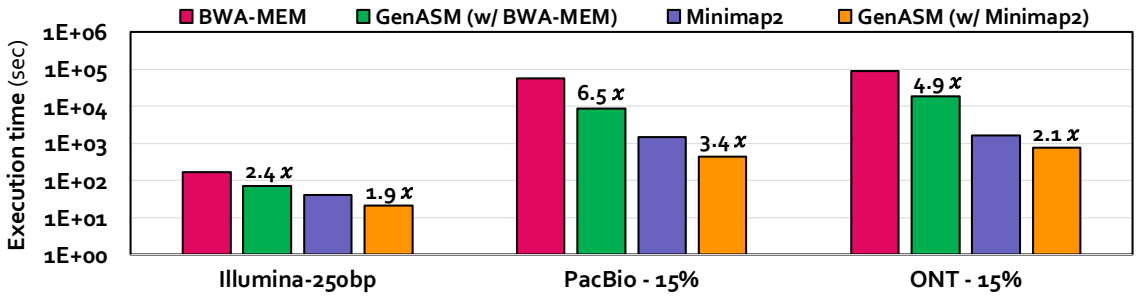



**Figure 4-10: Total execution time of the entire BWA-MEM and Minimap2 pipelines with and without GenASM.**

**Software Baselines (GPU).** We compare GenASM with the state-of-the-art GPU aligner, GASAL2 [9], using three datasets of varying size (100K, 1M, and 10M reference-

read pairs). Based on our analysis, we make three findings. First, for 100bp Illumina reads, GenASM provides 9.9×, 9.2×, and 8.5× speedup over GASAL2, while reducing the power consumption by 15.6×, 17.3× and 17.6× for 100K, 1M, and 10M datasets, respectively. Second, for 150bp Illumina reads, GenASM provides 15.8×, 13.1×, and 13.4× speedup over GASAL2, while reducing the power consumption by 15.4×, 18.0×, and 18.7× for 100K, 1M, and 10M datasets, respectively. Third, for 250bp Illumina reads, GenASM provides 21.5×, 20.6×, and 21.1× speedup over GASAL2, while reducing the power consumption by 16.8×, 20.2×, and 20.6× for 100K, 1M, and 10M datasets, respectively. We conclude that GenASM provides significant performance benefits and energy efficiency over GPU aligners for short reads.

**Hardware Baselines.** We compare GenASM with two state-of-the-art hardware accelerators for read alignment: GACT (from Darwin [301]) and SillaX (from GenAx [95]).

Darwin is a hardware accelerator designed for *long* read alignment [301]. Darwin contains components that accelerate both the filtering (D-SOFT) and alignment (GACT) steps of read mapping. The open-source RTL code available for the GACT accelerator of Darwin allows us to estimate the throughput, area and power consumption of GACT and compare it with GenASM for read alignment. In Darwin, GACT logic and the associated 128KB SRAM are responsible for filling the dynamic programming matrix, generating the traceback pointers and finding the maximum score. Thus, we believe that it is fair to compare the power consumption and the area of the GACT logic and GenASM logic, along with their associated SRAMs.

In order to have an iso-bandwidth comparison with Darwin's GACT, we compare only a single array of GACT and a single set of GenASM-DC and GenASM-TB, because (1) GenASM utilizes the high memory bandwidth that PIM provides only to parallelize many sets of GenASM-DC and GenASM-TB, and a single set of GenASM-DC and GenASM-TB does *not* require high bandwidth, and (2) all internal data of both GenASM and Darwin is provided by local SRAMs. We synthesize both designs (i.e., GenASM and GACT) at an

iso-PVT (process, voltage, temperature) corner, with the same number of PEs, and with their optimum parameters.

As Figure 4-11 shows, for a single GACT array with 64 PEs at 1GHz, the throughput of GACT decreases from 55,556 to 6,289 alignments per second when the sequence length increases from 1Kbp to 10Kbp, while consuming 277.7 mW of power. In comparison, for a single GenASM accelerator at 1GHz (with a 64-PE configuration), the throughput decreases from 236,686 to 23,669 alignments per second when the sequence length increases from 1Kbp to 10Kbp, while consuming 101 mW of power. This shows that, on average, GenASM provides 3.9× better throughput than GACT, while consuming 2.7× less power. Thus, GenASM provides 10.5× better throughput per unit power for long reads when compared to GACT.

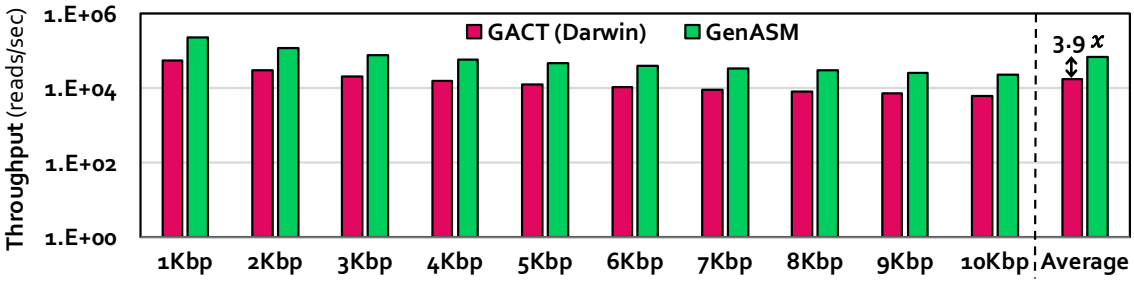



**Figure 4-11: Throughput comparison of GenASM and GACT from Darwin for long reads.**

As Figure 4-12 shows, we also compare the throughput of GenASM and GACT for short read alignment (i.e., 100–300bp reads). We find that GenASM performs 7.4× better than GACT when aligning short reads, on average. Thus, GenASM provides 20.0× better throughput per unit power for short reads when compared to GACT.

We compare the required area for the GACT logic with 128KB of SRAM and the required area for the GenASM logic (GenASM-DC and GenASM-TB) with 8KB of DC-SRAM and 96KB of TB-SRAMs, at 28nm. We find that GenASM requires 1.7× less area than GACT. Thus, GenASM provides 6.6× and 12.6× better throughput per unit area for long reads and for short reads, respectively, when compared to GACT.

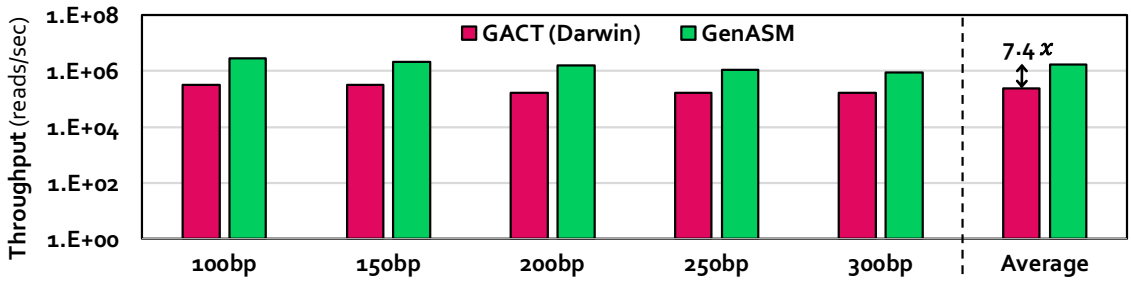



**Figure 4-12: Throughput comparison of GenASM and GACT from Darwin for short reads.**

The main difference between GenASM and GACT is the underlying algorithms. GenASM uses our modified Bitap algorithm, which requires only simple and fast bitwise operations. On the other hand, GACT uses the complex and computationally more expensive dynamic programming based algorithm for alignment. This is the main reason why GenASM is more efficient than GACT of Darwin.

GenAx is a hardware accelerator designed for *short* read alignment [95]. Similar to Darwin, GenAx accelerates both the filtering and alignment steps of read mapping. Unlike GenAx, whose design is optimized only for short reads, GenASM is more robust and works with *both* short and long reads. While we are unable to reimplement GenAx, the throughput analysis of SillaX (the alignment accelerator of GenAx) provided by the original work [95] allows us to provide a performance comparison between GenASM and SillaX for short read alignment.

We compare SillaX with GenASM at their optimal operating frequencies (2GHz for SillaX, 1GHz for GenASM), and find that GenASM provides 1.9× higher throughput for short reads (101bp) than SillaX (whose approximate throughput is 50M alignments per second). Using the area and power numbers reported for the computation logic of SillaX, we find that GenASM requires 63% less logic area (2.08 $mm^2$ vs. 5.64 $mm^2$) and 82% less logic power (1.18 W vs. 6.6 W).

In order to compare the total area of SillaX and GenASM, we perform a CACTI-based analysis [315] for the SillaX SRAM (2.02 MB). We find that the SillaX SRAM consumes an area of 3.47 $mm^2$, resulting in a total area of 9.11 $mm^2$ for SillaX. Although GenASM

(10.69 $mm^2$) requires 17% more total area than SillaX, we find that GenASM provides 1.6× better throughput per unit area for short reads than SillaX.

**Accuracy Analysis.** We compare the traceback outputs of GenASM and (1) BWA-MEM for short reads, (2) Minimap2 for long reads, to assess the accuracy and correctness of GenASM-TB. We find that the optimum ($W$, $O$) setting (i.e., window size and overlap size) for the GenASM-TB algorithm in terms of performance and accuracy is $W = 64$ and $O = 24$. With this setting, GenASM completes the alignment of all reads in each dataset, and increasing the window size does *not* change the alignment output.

For short reads, we use the default scoring setting of BWA-MEM (i.e., match=+1, substitution=-4, gap opening=-6, and gap extension=-1). For 96.6% of the short reads, GenASM finds an alignment whose score is equal to the score of the alignment reported by BWA-MEM. This fraction increases to 99.7% when we consider scores that are within ±4.5% of the scores reported by BWA-MEM.

For long reads, we use the default scoring setting of Minimap2 (i.e., match=+2, substitution=-4, gap opening=-4, and gap extension=-2). For 99.6% of the long reads with a 10% error rate, GenASM finds an alignment whose score is within ±0.4% of the score of the alignment reported by Minimap2. For 99.7% of the long reads with a 15% error rate, GenASM finds an alignment whose score is within ±0.7% of the score of the alignment reported by Minimap2.

There are two reasons for the difference between the alignment scores reported by GenASM and the scores reported by the baseline tools. First, GenASM performs traceback for the alignment with the minimum edit distance. However, the baseline can report an alignment that has a higher number of edits but a lower score than the alignment reported by GenASM, when more complex scoring schemes are used. Second, during the TB stage, GenASM follows a fixed order at each iteration when picking between substitutions, insertions, or deletions (based on the penalty of each error type). While we pick the error type with the lowest possible cost at the current iteration, another error type with a higher

initial cost may lead to a better (i.e., lower-cost) alignment in later iterations, which cannot be known beforehand.[3]

Although GenASM is optimized for unit-cost based scoring (i.e., edit distance) and currently provides only partial support for more complex scoring schemes, we show that GenASM framework can still serve as a fast, memory- and power-efficient, and quite accurate alternative for read alignment.

### 4.10.3 Use Case 2: Pre-Alignment Filtering

We compare GenASM with the state-of-the-art FPGA-based pre-alignment filter for short reads, Shouji [17], using two datasets provided in [17]. When we compare Shouji (with maximum filtering units) and GenASM for the dataset with 100bp sequences, we find that GenASM provides 3.7× speedup over Shouji, while reducing power consumption by 1.7×. When we perform the same analysis with 250bp sequences, we find that GenASM does not provide speedup over Shouji, but reduces power consumption by 1.6×.

In pre-alignment filtering for short reads, only GenASM-DC is executed (Section 4.8). The complexity of GenASM-DC is $O(n \times m \times k)$ whereas the complexity of Shouji is $O(m \times k)$, where $n$ is the text length, $m$ is the read length, and $k$ is the edit distance threshold. Going from the 100bp dataset to the 250bp dataset, all these three parameters increase linearly. Thus, the speedup of GenASM over Shouji for pre-alignment filtering decreases for datasets with longer reads.

To analyze filtering accuracy, we use Edlib [289] to generate the ground truth edit distance value for each sequence pair in the datasets (similar to Shouji). We evaluate the accuracy of GenASM as a pre-alignment filter by computing its false accept rate and false reject rate (as defined in [17]).

The false accept rate [17] is the ratio of the number of dissimilar sequences that are falsely accepted by the filter (as similar) and the total number of dissimilar sequences that

[3] We can add support for different orderings by adding more configurability to the GenASM-TB accelerator, which we leave for future work.

are rejected by the ground truth. The goal is to minimize the false accept rate to maximize the number of dissimilar sequences that are eliminated by the filter. For the 100bp dataset with an edit distance threshold of 5, Shouji has a 4% false accept rate, whereas GenASM has a false accept rate of only 0.02%. For the 250bp dataset with an edit distance threshold of 15, Shouji has a 17% false accept rate, whereas GenASM has a false accept rate of only 0.002%. Thus, GenASM provides a very low rate of falsely-accepted dissimilar sequences, and significantly improves the accuracy of pre-alignment filtering compared to Shouji.

While Shouji approximates the edit distance, GenASM calculates the actual distance. Although calculation requires more computation than approximation, a computed distance results in a near-zero (0.002%) false accept rate.[4] Thus, GenASM filters more false-positive locations out, leaving fewer candidate locations for the expensive alignment step to process. This greatly reduces the combined execution time of filtering and alignment. Thus, even though GenASM does not provide any speedup over Shouji when filtering the 250bp sequences, its lower false accept rate makes it a better option for this step of the pipeline with greater overall benefits.

The false reject rate [17] is the ratio of the number of similar sequences that are rejected by the filter (as dissimilar) and the total number of similar sequences that are accepted by the ground truth. The false reject rate should always be equal to 0%. We observe that GenASM always provides a 0% false reject rate, and thus does not filter out similar sequence pairs, as does Shouji.

### 4.10.4 Use Case 3: Edit Distance Calculation

We compare GenASM with the state-of-the-art edit distance calculation library, Edlib [289]. Figure 4-13 compares the execution time of Edlib (with and without finding the traceback

[4]The reason for the non-zero false accept rate of GenASM is that when there is a deletion in the first character of the query, GenASM does *not* count this as an edit, and skips this extra character of the text when computing the edit distance. Since GenASM reports an edit distance that is one lower than the edit distance reported by the ground truth, if GenASM's reported edit distance is below the threshold but the ground truth's is not, GenASM leads to a false accept.

output) and GenASM when finding the edit distance between two sequences of length 100Kbp, and also two sequences of length 1Mbp, which have similarity ranging from 60% to 99% (Section 4.9). Since Edlib is a single-thread edit distance calculation tool, for a fair comparison, we compare the throughput of only one GenASM accelerator (i.e., in one vault) with a single-thread execution of the Edlib tool.

As Figure 4-13 shows, when performing edit distance calculation between two 100Kbp sequences, GenASM provides 22–716× and 146–1458× speedup over Edlib execution without and with traceback, respectively. GenASM has the same execution time for both of the cases. When the sequence length increases from 100Kbp to 1Mbp, the execution time of GenASM increases linearly (since $W$ is constant, but $m + k$ increases linearly). However, due to its quadratic complexity, Edlib cannot scale linearly. Thus, for the edit distance calculation of 1Mbp sequences, GenASM provides 262–5413× and 627–12501× speedup over Edlib execution without and with traceback, respectively.

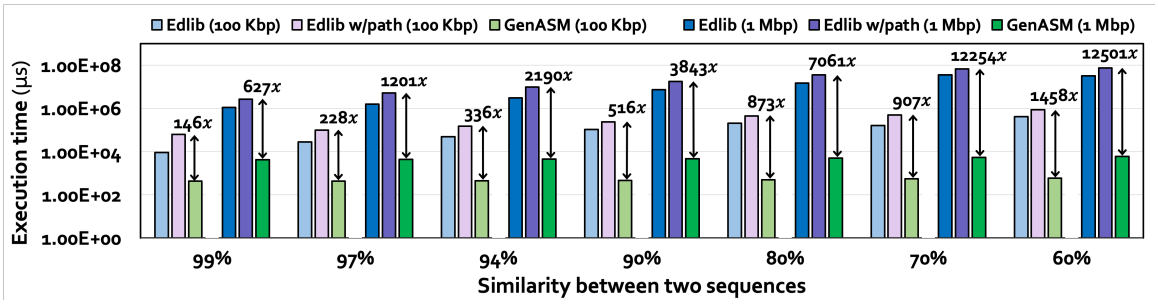



**Figure 4-13: Execution time comparison of GenASM and Edlib for edit distance calculation.**

Although both the GenASM algorithm and Edlib's underlying Myers' algorithm [216] use bitwise operations only for edit distance calculation and exploit bit-level parallelism, the main advantages of the GenASM algorithm come from (1) the divide-and-conquer approach we follow for efficient support for longer sequences, and (2) our efficient co-design of the GenASM algorithm with the GenASM hardware accelerator.

Based on our power analysis, we find that power consumption of Edlib is 55.3 W and 58.8 W when finding the edit distance between two 100Kbp sequences and two 1Mbp

sequences, respectively. Thus, GenASM reduces power consumption by 548× and 582× over Edlib, respectively.

We also compare GenASM with ASAP [35], the state-of-the-art FPGA-based accelerator for edit distance calculation. While we are unable to reimplement ASAP, the execution time and power consumption analysis of ASAP provided in [35] allows us to provide a comparison between GenASM and ASAP. ASAP is optimized for shorter sequences and reports execution time only for sequences of length 64bp–320bp [35]. Based on [35], the execution time of one ASAP accelerator increases from 6.8 µs to 18.8 µs when the sequence length increases from 64bp to 320bp, while consuming 6.8 W of power. In comparison, we report that the execution time of one GenASM accelerator increases from 0.017 µs to 2.025 µs when the sequence length increases from 64bp to 320bp, while consuming 0.101 W of power. This shows that GenASM provides 9.3–400× speedup over ASAP, while consuming 67× less power.

### 4.10.5 Sources of Improvement in GenASM

GenASM's performance improvements come from our algorithm/hardware co-design, i.e., both from our modified algorithm and our co-designed architecture for this algorithm. The sources of the large improvements in GenASM are (1) the very simple computations it performs; (2) the divide-and-conquer approach we follow, which makes our design efficient for both short and long reads despite their different error profiles; and (3) the very high degree of parallelism obtained with the help of specialized compute units, dedicated SRAMs for both GenASM-DC and GenASM-TB, and the vault-level parallelism provided by processing in the logic layer of 3D-stacked memory.

**Algorithm-Level.** Our divide-and-conquer approach allows us to decrease the execution time of GenASM-DC from $(\frac{m\times(m+k)\times k}{P\times w})$ cycles to $((\frac{W\times W\times min(W,k)}{P\times w})\times\frac{m+k}{W-O})$ cycles, where $m$ is the pattern size, $k$ is the edit distance threshold, $P$ is the number of PEs that GenASM-DC has (i.e., 64), $w$ is the number of bits processed by each PE (i.e., 64), $W$ is the window size (i.e., 64), and $O$ is the overlap size between windows (i.e., 24). Although the

total GenASM-TB execution time does *not* change ($(m + k)$ cycles vs. $((W - O) \times \frac{m+k}{W-O})$ cycles), our divide-and-conquer approach helps us decrease the GenASM-DC execution time by 3662× for long reads, and by 1.6 – 3.9× for short reads.

**Hardware-Level.** GenASM-DC's systolic-array-based design removes the data dependency limitation of the underlying Bitap algorithm, and provides 64× parallelism by performing 64 iterations of the GenASM-DC algorithm in parallel. Our hardware accelerator for GenASM-TB makes use of specialized per-PE TB-SRAMs, which eliminates the otherwise very high memory bandwidth consumption of traceback and enables efficient execution.

**Technology-Level.** With the help of 3D-stacked memory's vault-level parallelism, we can obtain 32× parallelism by performing 32 alignments in parallel in different vaults.

## 4.11 Other Use Cases of GenASM

We have quantitatively evaluated three use cases of approximate string matching for genome sequence analysis (Section 4.10). We discuss four other potential use cases of GenASM, whose evaluation we leave for future work.

**Read-to-Read Overlap Finding Step of de Novo Assembly.** *De novo* assembly [51] is an alternate genome sequencing approach that assembles an entire DNA sequence without the use of a reference genome. The first step of *de novo* assembly is to find read-to-read overlaps since the reference genome does not exist [280]. Pairwise read alignment (i.e., read-to-read alignment) is the last step of read-to-read overlap finding [251, 182]. As sequencing devices can introduce errors to the reads, read alignment in overlap finding also needs to take these errors into account. GenASM can be used for the pairwise read alignment step of overlap finding.

**Hash-Table Based Indexing.** In the indexing step of read mapping, the reference genome is indexed and stored as a hash table, whose keys are all possible fixed-length substrings (i.e., seeds) and whose values are the locations of these seeds in the reference

genome. This index structure is queried in the seeding step to find the candidate matching locations of query reads. As we need to find the locations of each seed in the reference text to form the index structure, GenASM can be used to generate the hash-table based index.

**Whole Genome Alignment.** Whole genome alignment [75, 246] is the method of aligning two genomes (from the same or different species) for predicting evolutionary or familial relationships between these genomes. In whole genome alignment, we need to align two very long sequences. Since GenASM can operate on arbitrary-length sequences as a result of our divide-and-conquer approach, whole genome alignment can be accelerated using the GenASM framework.

**Generic Text Search.** Although GenASM-DC is optimized for genomic sequences (i.e., DNA sequences), which are composed of only 4 characters (i.e., A, C, G and T), GenASM-DC can be extended to support larger alphabets, thus enabling generic text search. When generating the pattern bitmasks during the pre-processing step, the only change that is required is to generate bitmasks for the entire alphabet, instead of for only four characters. There is no change required to the edit distance calculation step.

As special cases of general text search, the alphabet can be defined as RNA bases (i.e., A, C, G, U) for RNA sequences or as amino acids (i.e., A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, V) for protein sequences. This enables GenASM to be used for RNA sequence alignment or protein sequence alignment [121, 287, 225, 189, 23, 24, 160, 225, 227, 123, 294, 79, 335].

## 4.12 Related Work

To our knowledge, this is the first approximate string matching acceleration framework that enhances and accelerates the Bitap algorithm, and demonstrates the effectiveness of the framework for multiple use cases in genome sequence analysis. Many previous works have attempted to improve (in software or in hardware) the performance of a *single* step

of the genome sequence analysis pipeline. Recent acceleration works tend to follow one of two key directions [16].

The first approach is to build pre-alignment filters that use heuristics to first check the differences between two genomic sequences before using the computationally-expensive approximate string matching algorithms. Examples of such filters are the Adjacency Filter [322] that is implemented for standard CPUs, SHD [321] that uses SIMD-capable CPUs, and GRIM-Filter [164] that is built in 3D-stacked memory. Many works also exploit the large amounts of parallelism offered by FPGA architectures for pre-alignment filtering, such as GateKeeper [18], MAGNET [19], Shouji [17], and SneakySnake [21]. A recent work, GenCache [217], proposes an in-cache accelerator to improve the filtering (i.e., seeding) mechanism of GenAx (for short reads) by using in-cache operations [8] and software modifications.

The second approach is to use hardware accelerators for the computationally-expensive read alignment step. Examples of such hardware accelerators are RADAR [130], FindeR [334], and AligneR [333], which make use of ReRAM based designs for faster FM-index search, or RAPID [117] and BioSEAL [158], which target dynamic programming acceleration with processing-in-memory. Other read alignment acceleration works include SIMD-capable CPUs [65], multicore CPUs [101, 190], and specialized hardware accelerators such as GPUs (e.g., GSWABE [190], CUDASW++ 3.0 [191]), FPGAs (e.g., FPGASW [86], ASAP [35]), or ASICs (e.g., Darwin [301] and GenAx [95]).

In contrast to GenASM, all of these prior works focus on accelerating only a single use case in genome sequence analysis, whereas GenASM is capable of accelerating at least three different use cases (i.e., read alignment, pre-alignment filtering, edit distance calculation) where approximate string matching is required.

## 4.13 Summary

We propose GenASM, an approximate string matching (ASM) acceleration framework for genome sequence analysis built upon our modified and enhanced Bitap algorithm. GenASM performs bitvector-based ASM, which can accelerate multiple steps of genome sequence analysis. We co-design our highly-parallel, scalable and memory-efficient algorithms with low-power and area-efficient hardware accelerators. We evaluate GenASM for three different use cases of ASM in genome sequence analysis for both short and long reads: read alignment, pre-alignment filtering, and edit distance calculation. We show that GenASM is significantly faster and more power- and area-efficient than state-of-the-art software and hardware tools for each of these use cases.

# Chapter 5

# BitMAc: FPGA-Based Near-Memory Acceleration of Bitvector-Based Sequence Alignment

Modern data-intensive applications demand high computation capabilities with strict power constraints. Unfortunately, such applications suffer from a significant waste of both execution cycles and energy in current computing systems due to the costly data movement between the computation units and the memory units [285, 213, 37, 284, 235, 214, 103, 107, 118, 87, 286]. GenASM-based sequence alignment (Chapter 4) is an example for such applications.

Recent FPGAs couple a reconfigurable fabric with high-bandwidth memory (HBM) to enable more efficient data movement and improve overall performance and energy efficiency. This trend is an example of a paradigm shift to *near-memory computing*. In this work, we propose BitMAc, where we leverage such an FPGA with high-bandwidth

memory (HBM) for presenting an FPGA-based prototype for our GenASM accelerators. In BitMAc, we map GenASM on an FPGA with a state-of-the-art 3D-stacked memory (HBM2), where HBM2 offers high memory bandwidth and FPGA resources offer high parallelism by instantiating multiple copies of the GenASM accelerators. We exploit intra-level parallelism by instantiating multiple processing elements (PEs) for the DC execution, and inter-level parallelism by running multiple independent GenASM executions in parallel. We show that due to the simplicity of the GenASM algorithms, BitMAc is a low-cost and scalable solution for bitvector-based sequence alignment, with its high energy-efficiency and low resource requirements.

## 5.1 Near-Memory Computing with Modern FPGAs

FPGAs are one of the most commonly used form of reconfigurable hardware engines today, and their computational capabilities are greatly increasing every generation due to increased number of transistors on the FPGA chips. Besides these increasing computational capabilities, modern FPGAs provide (1) an advanced technology node of 7-14nm FinFET [142, 319] that offers higher performance, (2) different types of on-chip memories (e.g., M20Ks for Intel/Altera chips or UltraRAM (URAM) for Xilinx chips that offer large on-chip memory next to the logic, and (3) the integration of high-bandwidth memory (HBM) on the same package with an FPGA that allows us to implement our accelerator logic much closer to the memory with an order of magnitude more bandwidth than traditional DDR4-based FPGA boards [284, 285]. Thus, modern FPGA architectures can deliver unprecedented levels of integration and compute capability due to these new advances and features, which provide an opportunity to largely alleviate the memory bottleneck of real-world data-intensive applications.

One example of such modern FPGAs is Intel's Stratix 10 MX device [145]. Intel Stratix 10 MX integrates 3D-stacked High-Bandwidth DRAM Memory (HBM2) alongside a high-performance monolithic 14 nm FPGA fabric die. The fabric die contains 2,100K logic

elements (LEs), 94.5 Mbits of embedded eSRAM blocks (each with 47.25 Mbit), over 134 Mbits of embedded M20K memory blocks (each with 20 Kbit), and hard memory controllers.

In Figure 5-1, we show a high-level schematic of the Intel Stratix 10 MX device. The FPGA fabric die is connected to two HBM stacks, each of which has 8GB of capacity and 8 independent channels or 16 independent pseudo-channels [154]. Since there are two HBM stacks per each Stratix 10 MX device, in total we have 16GB of memory capacity and 32 independent pseudo-channels. Each HBM2 channel supports a 128-bit DDR data bus, thus, providing two independent 64-bit data bus for each pseudo-channel. For each pseudo-channel, the accesses happen with the burst length of 4, thus at each access, a pseudo-channel reads/writes 32B of data.

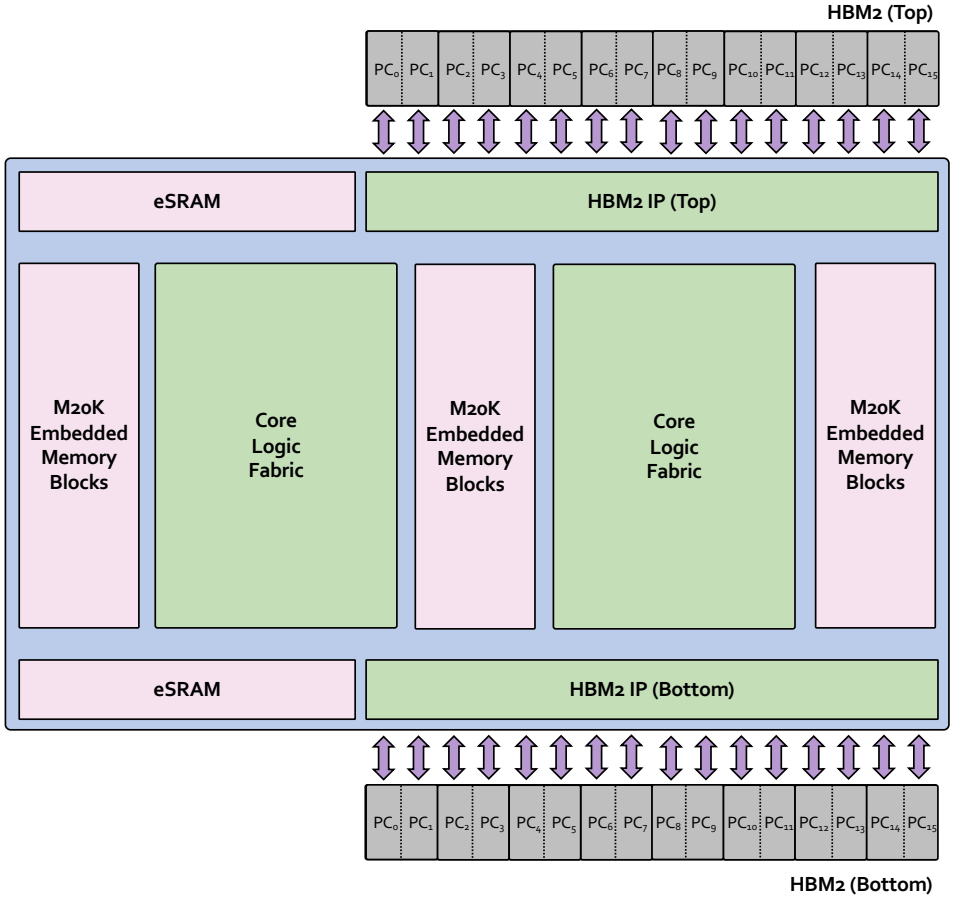



**Figure 5-1: High-level schematic of the Intel Stratix 10 MX device.**

The HBM2 IP includes a soft logic adaptor implemented in core logic to efficiently interface user logic to the HBM2 controller. The user interface to the HBM2 controller is maintained through the AXI4 protocol [31]. The HBM2 IP provides 16 AXI interfaces for each HBM2 controller, with one AXI interface available per HBM2 pseudo-channel. Thus, each AXI interface supports a 256-bit wide write data and 256-bit wide read data interface to/from the HBM2 controller.

## 5.2 BitMAc Implementation

In BitMAc, we map our GenASM accelerators and their associated SRAMs (See Section 4.7) to the Intel Stratix 10 MX device. As we show in Figure 5-2, we map the DC and TB datapaths along with the HBM2 interface to the FPGA core logic fabric. Due to their larger total capacity, we map TB-SRAMs to the M20Ks. On the other hand, since DC-SRAM is required only to store the input text and pattern, we store them into registers and thus, map them to the core logic fabric as well.

### 5.2.1 Mapping TB-SRAMs to M20Ks

In our GenASM design, we set the window size (W) as 64 and at each cycle, we store 3 out of the 4 intermediate bitvectors (i.e., deletion, insertion, and match bitvectors since substitution can be obtained using the deletion bitvector). However, for a better mapping, here in our BitMAc design, to match the width of the data port of M20Ks (40 bits) and to decrease the amount of data written to M20Ks, we set the window size as 60 and we store only 2 out of the 4 intermediate bitvectors (i.e., deletion and match) since substitution bitvector can be obtained using the deletion bitvector and if none of these 3 bitvectors contain the 0-of-interest at the position-of-interest, it means that insertion bitvector has it, thus it is not required to explicitly store the fourth bitvector. These changes enable us to decrease the amount of data generated by each processing element of the DC accelerator

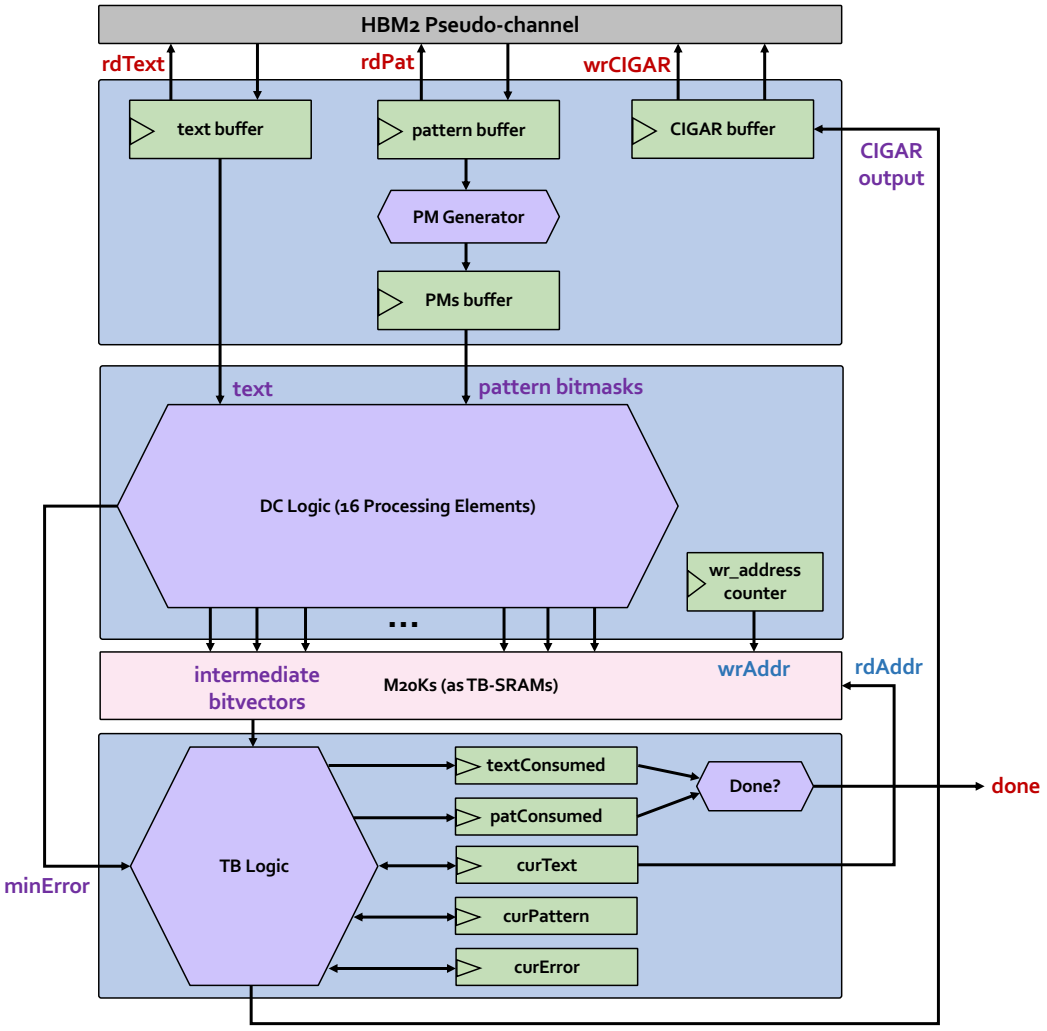



**Figure 5-2: The overview of a BitMAc accelerator attached to one pseudo-channel of an Intel Stratix 10 MX device.**

to be 120 bits (instead of the original 192 bits), which is 3 times of the width of an M20K data bus.

Since we have 6847 many M20Ks in our FPGA and since we need 3 M20K ports for each processing element, within our budget, we can instantiate 2282 many processing elements (PEs). To be able to instantiate multiple BitMAc accelerators to exploit inter-level parallelism, we set the number of PEs as 16, based on our empirical analysis. This enables us to set the error threshold ($k$) for each window of the BitMAc execution as 15 since for

each error value (0...$k$), we have one dedicated PE (Figure 4-4). With this configuration, we can instantiate maximum of 142 BitMAc accelerators on our FPGA.

### 5.2.2 Mapping DC and TB Datapaths to the FPGA Logic

We map the GenASM-DC and GenASM-TB datapaths to the FPGA logic without any modifications. We also map our FSM design, which controls the memory accesses, DC and TB executions to the FPGA logic. Different than our GenASM design, we also implement the memory components required to store the inputs and outputs of the system as registers, instead of the DC-SRAM in the GenASM design. The reason of this design choice is to be able to dedicate all of the M20Ks solely for storing the intermediate bitvectors.

### 5.2.3 Mapping of the Main Memory to the HBM2 Stacks

In order to exploit the high-bandwidth that the HBM2 stacks provide, we store our inputs (text-pattern pairs) and output (CIGAR string) in the HBM2 memory. Since we have 32 pseudo-channels in total, we can attach 4 BitMAc accelerators to each of these pseudo-channels not to exceed our 142 accelerator limit (See Section 5.2.1).

Since at each burst access of a pseudo-channel, we can read/write 32B of data, we can read one pair of a 60bp-length text (120 bits with a 2-bit implementation) and a 60bp-length pattern (120 bits with a 2-bit implementation) at each burst. For a better memory alignment, we can also set each of these sequences as 64bp (256 bits in total). Similarly, for the CIGAR output, we can have maximum of 256 bits. Thus, in memory, we can reserve 64B memory space for each pair of text and pattern (16B text + 16B pattern + 32B CIGAR output). This also enables us to (1) have a single memory address counter for each pseudo-channel, and (2) have 1 memory access for reading and 1 memory access for writing data from/to HBM2 for each window's execution.

## 5.3 Evaluation

### 5.3.1 Methodology

We implement our DC and TB accelerator datapaths using SystemVerilog and incorporate the M20Ks and the HBM2 interface for both top and bottom HBM2 stacks using M20K IP and HBM2 IP of Intel Quartus Prime [144], respectively. The complete BitMAc design has 4 BitMAc accelerators connected to each pseudo-channel (128 in total), where each BitMAc accelerator contains a DC accelerator with 16 PEs, a TB accelerator, an FSM, and 13.2KB of M20Ks. We synthesize and place & route the complete BitMAc design clocked at 200 MHz, and report the resource utilization and power analysis results based on the compilation flow that Intel Quartus Prime provides. We also modify the spreadsheet-based analytical model of GenASM based on our new BitMAc design, and report our performance results based on this analytical model.

We perform our BitMAc evaluation for the read alignment use case (See Section 4.8). Similar to GenASM, we compare BitMAc with (1) the read alignment steps of Minimap2 [182] and BWA-MEM [180] as the CPU-based baselines, and (2) GASAL2 [9] as the GPU-based baseline. We also compare BitMAc with GenASM to show the comparison for the FPGA- and ASIC-based implementations, respectively. You can refer to Section 4.9 for more details on the baselines and the datasets used.

### 5.3.2 Power Analysis

Table 5-1 shows the dynamic and total on-chip power dissipation of different configurations of our BitMAc design on an Intel Stratix 10 MX device. We show that the total power dissipation of a single BitMAc accelerator is 6 W. We also show that for 32 BitMAc accelerators (one BitMAc accelerator per pseudo-channel), the total power dissipation is 17.2 W, and for 128 BitMAc accelerators (four BitMAc accelerators per pseudo-channel; our complete design), the total power dissipation is 48.9 W.

**Table 5-1: On-chip power dissipation of the BitMAc design.**

| Component | Dynamic On-Chip Power Dissipation | Total On-Chip Power Dissipation |
|---|---|---|
| DC Logic (16 PEs) | 128.57 mW | |
| TB Logic | 10.24 mW | |
| FSM Logic | 3.15 mW | |
| M20Ks | 211.61 mW | |
| Other | 15.72 mW | |
| **Total – 1 BitMAc Accelerator** | **369.29 mW (0.4 W)** | **6043.24 mW (6.0 W)** |
| **Total – 32 BitMAc Accelerators (1 per each pseudo-channel)** | **11569.92 mW (11.6 W)** | **17234.67 mW (17.2 W)** |
| **Total – 128 BitMAc Accelerators (4 per each pseudo-channel)** | **43042.90 mW (43 W)** | **48935.65 mW (48.9 W)** |

When we look at the power dissipation analysis by block type for our complete BitMAc design, we find that 59% of the total dissipation accounts for the M20Ks, 9% accounts for the combination cells, 7% accounts for the register cells, 13% accounts for the clock network, and 12% accounts for the static power dissipation. Thus, M20Ks are the main contributors to the total on-chip power dissipation.

### 5.3.3 Performance Analysis

**CPU-based Baselines.** Figure 5-3 shows the read alignment throughput (reads/sec) of BitMAc and the alignment steps of BWA-MEM and Minimap2, when aligning long noisy PacBio and ONT reads against the human reference genome. As Figure 5-3 shows, BitMAc provides 761× and 136× throughput improvement over the alignment steps of BWA-MEM and Minimap2 for their 12-thread execution, respectively, while reducing the power consumption by 1.9× and 2.0× for 12-thread execution.

Figure 5-4 compares the read alignment throughput (reads/sec) of BitMAc with that of the alignment steps of BWA-MEM and Minimap2, when aligning short Illumina reads against the human reference genome. We find that BitMAc provides 92× and 130× throughput improvement over the alignment steps of BWA-MEM and Minimap2 for their

12-thread execution, respectively, while reducing the power consumption by 2.2× and 2.0× for 12-thread execution.

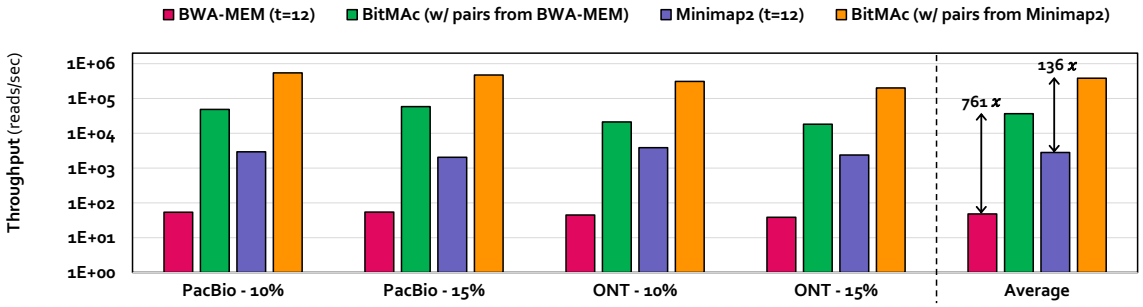



**Figure 5-3: Throughput comparison of BitMAc and the alignment steps of BWA-MEM and Minimap2 for long reads.**

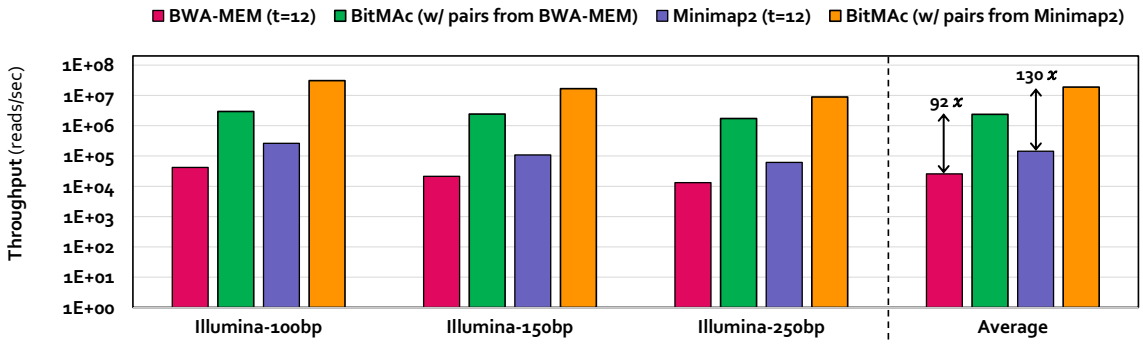



**Figure 5-4: Throughput comparison of BitMAc and the alignment steps of BWA-MEM and Minimap2 for short reads.**

**GPU-based Baseline.** We compare BitMAc with the state-of-the-art GPU aligner, GASAL2 [9], using three datasets of varying size (100K, 1M, and 10M reference-read pairs). Based on our analysis, we make three findings. First, for 100bp Illumina reads, BitMAc provides 7.9×, 7.3×, and 6.8× speedup over GASAL2, while reducing the power consumption by 2.9%, 12.3% and 13.7% for 100K, 1M, and 10M datasets, respectively. Second, for 150bp Illumina reads, BitMAc provides 13.2×, 10.9×, and 11.1× speedup over GASAL2, while reducing the power consumption by 1.5%, 15.6% and 19.1% for 100K, 1M, and 10M datasets, respectively. Third, for 250bp Illumina reads, BitMAc provides 19.4×, 18.7×, and 19.0× speedup over GASAL2, while reducing the power consumption by 9.9%, 25.2% and 26.6% for 100K, 1M, and 10M datasets, respectively.

### 5.3.4 FPGA Resource Utilization

We list the resource utilization of different configurations of our BitMAc design on an Intel Stratix 10 MX device in Table 5-2. We make four key observations. First, we find that for the complete BitMAc design (128 BitMAc accelerators), the DC logic corresponds to 73% of the logic utilization, the TB logic corresponds to 23% of the logic utilization, the FSM logic corresponds to 1% of the logic utilization, and the memory interface logic corresponds to 3% of the logic utilization. Second, we find that there is a linear scaling of the FPGA resources with the number of instances we include in our design. Third, we observe that the logic utilization (64%) is lower than the on-chip memory utilization (90%). Fourth, due to high amount of data that needs to be stored for the TB execution on M20Ks, we are bottlenecked by the amount of on-chip memory (i.e., M20Ks) we have. Because of this limitation, we also cannot saturate the high bandwidth that multiple HBM2 stacks on the FPGA provide. Thus, in order to scale further and fully exploit the high-bandwidth that HBM2 stacks provide, we need (1) algorithm-level modifications to decrease the amount of data that need to be stored in M20Ks, and (2) newer FPGA chips that provide a higher amount of on-chip memory capacity.

**Table 5-2: FPGA resource utilization of the BitMAc design.**

| Configuration | Logic Utilization | M20K | eSRAM | DSP |
|---|---|---|---|---|
| 1 BitMAc Accelerator | 0.5% | 0.7% | 0% | 0% |
| 32 BitMAc Accelerators (1 per each pseudo-channel) | 17.7% | 22.4% | 0% | 0% |
| 128 BitMAc Accelerators (4 per each pseudo-channel) | 64.3% | 89.7% | 0% | 0% |

### 5.3.5 FPGA-based BitMAc vs. ASIC-based GenASM

We also compare BitMAc with GenASM (Chapter 4) to show the comparison for the FPGA- and ASIC-based implementations, respectively. For GenASM, we consider the

configuration explained in Section 4.9 and for BitMAc, we consider the configuration explained in Section 5.3.1. GenASM operates at 1GHz while having 64 PEs per accelerator and 32 accelerators in total in its complete design. On the other hand, BitMAc operates at 200MHz while having 16 PEs per accelerator and 128 accelerators in total in its complete design.

Based on our analysis, we find that BitMAc provides 1.18× throughput improvement over GenASM for both short and long reads with the help of our algorithmic and hardware design changes for a more efficient FPGA mapping. However, due to the cost of reconfigurability of the FPGAs and dedicating the whole chip for the BitMAc design, a single FPGA-based BitMAc accelerator increases the power consumption by 4× compared to a single ASIC-based GenASM accelerator when we only take the dynamic on-chip power dissipation into consideration. On the other hand, when we take the total on-chip power dissipation into consideration, BitMAc increases the power consumption by 59×.

## 5.4 Summary

We propose BitMAc, where we leverage a modern FPGA with high-bandwidth memory (HBM) for presenting an FPGA-based prototype for our GenASM accelerators. In BitMAc, we map GenASM on Stratix 10 MX FPGA with a state-of-the-art 3D-stacked memory (HBM2), where HBM2 offers high memory bandwidth and FPGA resources offer high parallelism by instantiating multiple copies of the GenASM accelerators.

After re-modifying the GenASM algorithms for a better mapping to existing FPGA resources, we show that BitMAc provides 64% logic utilization and 90% on-chip memory utilization, while having 48.9 W of total power consumption. We compare BitMAc with state-of-the-art CPU-based and GPU-based read alignment tools. For long reads, BitMAc provides 761× and 136× speedup over the alignment steps of the state-of-the-art read mappers, BWA-MEM and Minimap2, respectively, while reducing power consumption by 1.9× and 2.0×. For short reads, BitMAc provides 92× and 130× speedup over the alignment

steps of BWA-MEM and Minimap2, respectively, while reducing power consumption by 2.2× and 2.0×. We also show that BitMAc provides significant speedup compared to the GPU-based baseline, GASAL2. Thus, BitMAc is a low-cost and scalable FPGA-based solution for bitvector-based sequence alignment.

# Chapter 6

# SeGraM: A Hardware Acceleration Framework for Sequence-to-Graph Mapping

An emerging problem with using a single reference genome for an entire species is the DNA variation that exists from organism to organism within a population (known as *genetic diversity*), which results from DNA mutations over time. The use of a single reference genome can bias the mapping process and downstream analysis towards the DNA composition and variations present in the reference organism, because (1) the organism whose DNA is being constructed may have a different set of variations, and (2) the reference organism's variations might be uncommon among most organisms in the population [247]. As a result, the reconstructed DNA may not be a faithful reproduction of the original sequence. Combined with errors that can be introduced during genome sequencing (with error rates as high as 5–10% for long reads with thousands of base pairs [151, 312, 30, 304]), reference bias can lead to significant inaccuracies during mapping. This can create issues for a wide range of genomic studies, from identifying mutations that lead to cancer, to

tracking mutating variants of viruses such as SARS-CoV-2 [59], where the details of the variation from the reference are critical to our understanding [27].

An emerging technique to overcome reference bias is the use of graph-based representations of a species' genome, known as *genome graphs* [251, 247]. A genome graph represents the reference genome *and* known genetic variations in the population as a graph-based data structure. A node represents one or more base pairs, and edges connect the base pairs in a node to all of the possible base pairs that come next in the sequence, with multiple outgoing edges from a node capture genetic variation. Genome graphs are growing in popularity for a number of applications, such as variation calling [99], genome assembly [58, 251, 326, 283], error correction [270], and multiple sequence alignment [246, 177].

For genome sequence analysis, instead of mapping an organism's reads to the linear DNA sequence of a single reference organism (known as sequence-to-sequence mapping), *sequence-to-graph mapping* captures the inherent genetic diversity among a population and can result in significantly more accurate read mapping [99]. Like sequence-to-sequence mapping, sequence-to-graph mapping follows the *seed-and-extend strategy* [262]. The first time the graph is constructed, its nodes are indexed for fast lookup. During mapping, this pre-processed index is used in the *seeding* step, which aims to find potential seed matches between the query read and a region of the graph. After optionally clustering or *filtering* the potential matches, *alignment* is performed between all of the remaining seed locations of the graph and the query read.

Due to its nascent nature, only a few software tools exist for graph-based genome sequence analysis [228, 262, 99, 261, 163]. Given the additional complexities and overheads of processing a genome graph instead of a linear reference genome, graph-based analysis exacerbates analysis bottlenecks such as read-to-reference mapping. **Our goal** is to design high-performance, scalable, power- and area-efficient hardware accelerators that alleviate bottlenecks in both the seeding and alignment steps of sequence-to-graph mapping,

with support for both short (e.g., Illumina) and long (e.g., PacBio, ONT) read sequencing technologies.

In this work, we propose SeGraM, a hardware acceleration framework for sequence-to-graph mapping and alignment. For seeding, we base SeGraM on a memory-efficient minimizer-based seeding algorithm, and for alignment, we develop a new bitvector-based, highly-parallel sequence-to-graph alignment algorithm. We *co-design* both of our algorithms with high-performance, area- and power-efficient hardware accelerators. SeGraM consists of two components: (1) MinSeed, which provides hardware support to execute our minimizer-based seeding algorithm efficiently, and (2) BitAlign, which provides hardware support to execute our bitvector-based sequence-to-graph alignment algorithm effectively. To our knowledge, SeGraM is the *first* hardware acceleration framework for sequence-to-graph mapping, MinSeed is the *first* hardware accelerator for minimizer-based seeding, and BitAlign is the *first* hardware accelerator for sequence-to-graph alignment.

## 6.1 Minimizer-Based Indexing & Seeding

In many applications of string comparisons for sequence analysis, the first step is to find the set of *seeds* to represent each sequence. Seeds are chosen from the set of *k-mers*, which are exact matching subsequences of length $k$ between the query sequence and reference [265]. Considering ultra-long reads being produced by recent sequencing machines, the size of the k-mer set can be enormous depending on $k$, making it hard to store and process.

One possible approach for reducing storage requirements is to apply fixed k-mer sampling methods [192]. Another approach divides the sequence into windows with a predefined size and selects a k-mer from each window according to a scoring mechanism as representatives. These unique k-mers, called *minimizers* [265, 272], ensure that two different sequences are represented with the same seed if they contain a long enough

common subsequence. Since minimizer-based seeding eliminates some possible k-mers, sensitivity and speed can vary depending on the window size and scoring [150].

## 6.2 Sequence-to-Graph Alignment

The goal of aligning a sequence to a graph is finding the path on the graph that yields the sequence's highest alignment score [149]. Sequence-to-graph alignment algorithm with quadratic time complexity was first formulated by Navarro [224] and it traverses the DP matrix row by row instead of the original column-wise fashion of linear alignment. After calculating the terms on a row, the algorithm makes searches on the graph and propagates the values to the next row. There are many efforts for optimizing or accelerating the dynamic programming for linear alignment. However, obtaining efficient solutions for sequence-to-graph alignment demands attention with the growing trend in genome graphs.

## 6.3 Motivation and Goal

As shown in GenASM (Chapter 4) and other prior works [18, 301, 95, 166, 21, 111, 96, 36, 217, 164, 175, 157, 158], sequence-to-sequence mapping is one of the major bottlenecks of the genome sequence analysis pipeline and need to be accelerated using specialized hardware. Since a graph-based representation of the genome is more complex than the linear representation, sequence-to-graph mapping places greater pressure on this bottleneck. Thus, there is a pressing need to develop techniques that provide fast, efficient, and low-cost sequence-to-graph mapping, which support both short reads (e.g., Illumina reads) and long reads (e.g., PacBio and ONT reads).

Even though there are several hardware accelerators designed to alleviate bottlenecks in several steps of linear read mapping (e.g., pre-alignment filtering [164, 21], sequence-to-sequence alignment [301, 95, 279]), none of these designs can be employed directly for the

sequence-to-graph mapping problem. This is because linear sequence mapping is a special case of sequence-to-graph mapping, where all nodes have only one edge, and hence the corresponding accelerators are limited to this special case, but unsuitable for the general problem, where we also need to consider multiple edges that a node can have (i.e., hops). However, with the growing importance and usage of genome graphs, it is crucial to have efficient designs for sequence-to-graph mapping, which are tuned to work with both short and long reads.

In this work, our goal is to design a high-performance, memory-efficient, and low-power hardware acceleration framework for sequence-to-graph mapping and alignment. To this end, we propose SeGraM, the first hardware acceleration framework for sequence-to-graph mapping and alignment. For an efficient and general-purpose acceleration, SeGraM aims to accelerate both seeding and sequence-to-graph alignment steps of the sequence-to-graph mapping pipeline, which are optimized for both short and long reads. We base SeGraM upon a minimizer-based seeding algorithm and we propose a novel bitvector-based algorithm to perform approximate string matching between a read and a graph-based reference. To our knowledge, SeGraM proposes the first hardware accelerator for minimizer-based seeding and the first hardware accelerator for sequence-to-graph alignment.

## 6.4 Overview of SeGraM

In SeGraM, we *co-design* our minimizer-based seeding algorithm and bitvector-based sequence-to-graph alignment algorithm with highly parallel, low power, and area efficient accelerators. SeGraM consists of two main components: (1) MinSeed (MS), which finds the minimizers for a given query read and fetches the candidate seed locations for the selected minimizers; and (2) BitAlign (BA), which for each candidate seed, aligns the query read for the subgraph surrounding the seed and finds the optimal alignment.

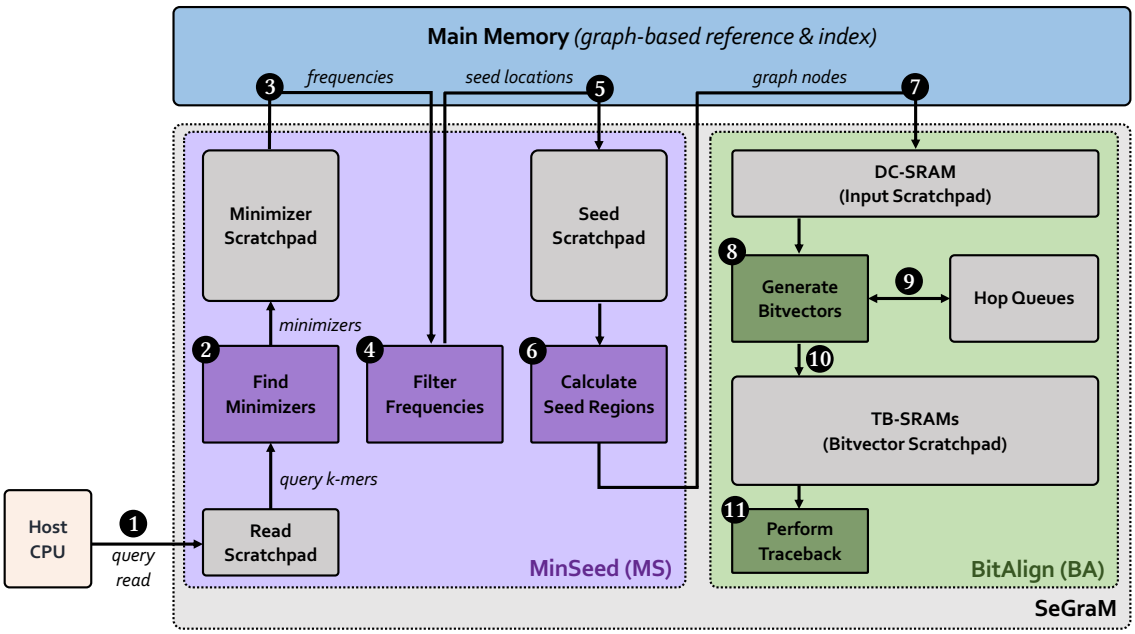



**Figure 6-1: Overview of SeGraM.**

Before SeGraM execution starts, as the pre-processing steps, each chromosome's graph structure is generated, then each graph's nodes are indexed, and both the resulting graph and hash table index are pre-loaded to the main memory.

SeGraM execution starts when the query read is streamed from the host CPU and MinSeed writes it to the *read scratchpad* (❶). Using all of the *k-mers* of the query read, MinSeed finds the minimizers and writes them to the *minimizer scratchpad* (❷). For each minimizer, MinSeed fetches its frequency from the hash table at the main memory (❸) and filters out the minimizers whose frequency is above an user-defined threshold (❹). Next, MinSeed fetches the seed locations of the remaining minimizers from the main memory, and writes them to the *seed scratchpad* (❺). Finally, MinSeed calculates the candidate reference region for each seed (❻), fetches the graph nodes from the memory for each candidate region and writes them to the *input scratchpad* (❼). Once MinSeed sends the subgraph corresponding to the candidate reference region along with the query read, BitAlign execution starts with generating the bitvectors (❽) required for the distance calculation step of the alignment (i.e., seed-extension). While generating these bitvectors,

BitAlign writes them to the *hop queues* in order to handle the hops required for the graph alignment (❾), and also, to the *bitvector scratchpad* (❿). Once BitAlign finishes generating and writing all the bitvectors, it starts reading them back from the *bitvector scratchpad*, performs the traceback operation, and finds the optimal alignment between the subgraph and the query read (⓫).

## 6.5 Pre-Processing for SeGraM

SeGraM mechanism requires two pre-processing steps before starting its execution: (1) generating the graph-based reference using a linear reference genome (i.e., as a FASTA file [140]) and its associated variations (i.e., as VCF file(s) [240]), and topologically sorting this graph [156]; and (2) indexing the nodes of this generated graph and generating the hash table-based index.

**Graph-based reference generation.** As the first pre-processing step, we generate the graph-based reference from the input FASTA file and VCF file(s) using the vg toolkit's [99] `vg construct` command. We generate one graph for each chromosome, and we set the maximum sequence length of each node as 16Kbp. For the alignment step of sequence-to-graph mapping, we need to make sure the nodes of our graphs are topologically sorted. Thus, as our next step, we sort our graphs using the `vg ids -s` command. Then, we convert our VG-formatted graphs to GFA-formatted [2] graphs using the `vg view` command since GFA is easier to work with for the later steps of the pre-processing.

As we show in Figure 6-2, in order to store the graph-based reference, we generate three separate table structures by using the GFA-formatted graphs: (1) the nodes table, which stores the nodes of the graphs as the <node id> as the key and the <node's sequence's length, starting address at the sequences table, node's number of outgoing edges, starting address at the edges table> as the value; (2) the sequences table, which stores the associated sequences of each node; and (3) the edges table, which stores the associated outgoing nodes (by node ID) of each node.

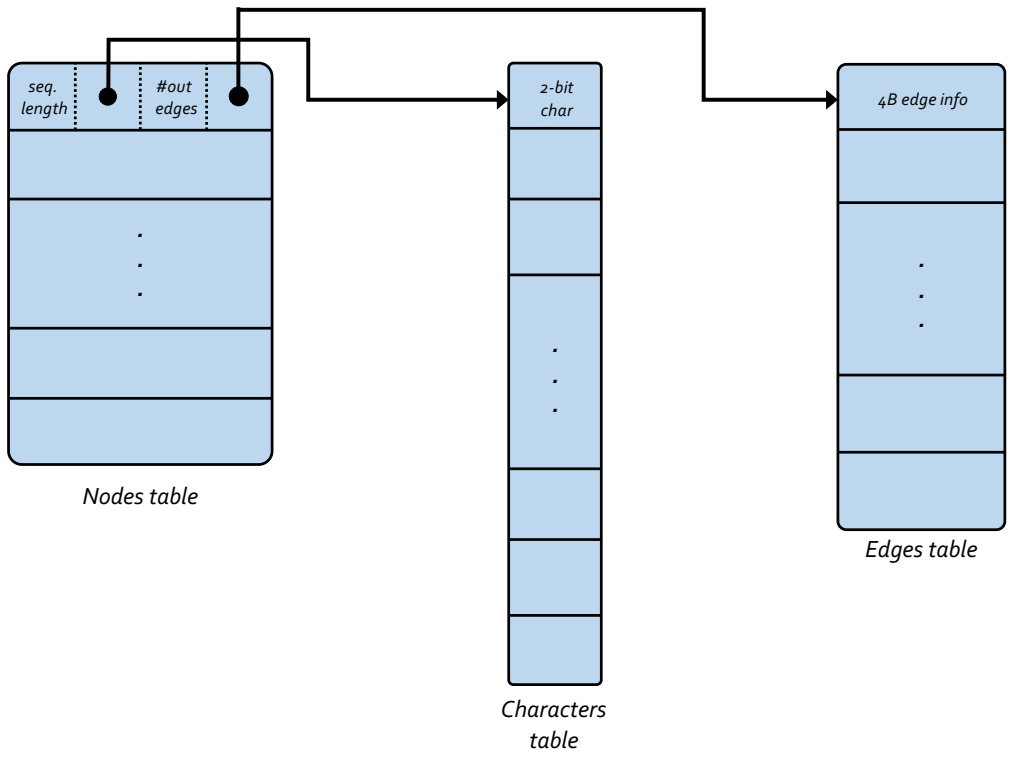



**Figure 6-2: Memory layout of the graph-based reference structure.**

We use the stats (i.e., number of nodes, number of edges, and total sequence length) for each chromosome's associated graph to determine the data sizes of our graph structure. Based on our analysis, we find that each node in the nodes table requires 32B per entry and the total size of the nodes table is $\#nodes * 32B$. Since we can store characters in the sequences table using a 2-bit representation (since we only have 4 possible characters: A (00), C (01), G (10), and T (11)), the total size is $total\ sequence\ length * 2bit$. We also find that each entry in the edges table requires 4B, thus the total size of the edges table is $\#edges * 4B$. In total, it takes 1.4 GB to store the graph structures for 24 chromosomes of the human genome.

**Hash table-based index generation.** As the second pre-processing step, we generate the hash table-based index for each of the generated graphs (one for each chromosome). Different than the traditional linear read mapping, in sequence-to-graph mapping's indexing step, the nodes of the graph structure are indexed and stored in the hash table. As we explain in Section 6.6, since SeGraM performs minimizer-based seeding, we also use

minimizers [265, 181, 182] as the keys and their exact matching locations in the graphs' nodes as the values of the index while generating the index.

As we show in Figure 6-3, in order to store the hash table-based index, we use a three-level structure. In the first-level of the hash table, similar to Minimap2 [182], we have *buckets* to decrease the memory footprint of the index. Each entry in this first-level of the index stores <starting address of the minimizers in the corresponding bucket in the second-level, number of minimizers in the corresponding bucket>. In the second-level of the hash table, we have the *minimizers*. Each entry in this second-level of the index stores <hash value of the corresponding minimizer, starting address of the seed locations of the corresponding minimizer in the second-level, number of locations of the corresponding minimizer>. Minimizers are sorted based on their hash values in the second-level of the index. Finally, in the third-level of the hash table, we have the *seed locations*. Each entry in this third-level of the index stores <node ID of the corresponding seed location, relative offset of the corresponding seed location within the node>. Locations are grouped based on their corresponding minimizers and sorted within each group based on their values.

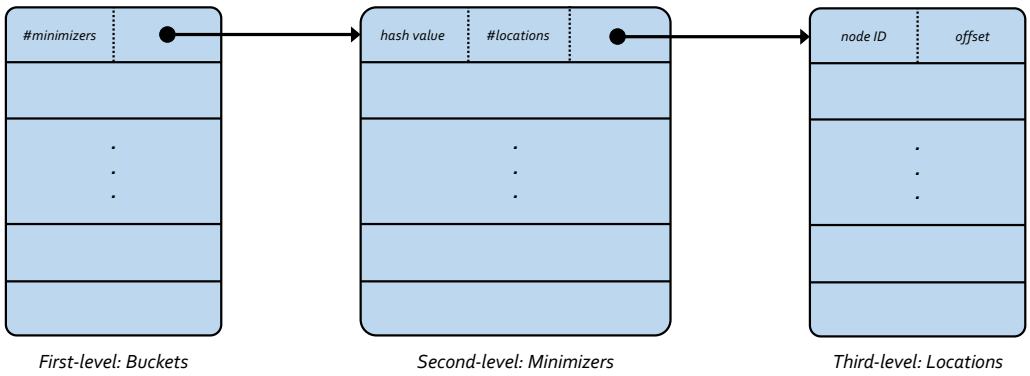



**Figure 6-3: Memory layout of the hash table-based index structure.**

We use the stats (i.e., number of distinct minimizers, total number of locations, maximum number of minimizers per bucket, and maximum number of locations per minimizer) for each graph to determine the data sizes of our index structure. Based on our empirical analysis, we find that $2^{24}$ is the optimum number of buckets in terms of memory footprint and the number of keys per each bucket. We also find that each bucket entry requires 4B

of data, thus the total size of the first-level of the index is $2^{24} * 4B$. Each minimizer in the second-level of the index requires 12B of data, thus the total size of the second-level of the index is *#distinct minimizers* $*$ *12B*. Each location in the third-level of the index requires 8B of data, thus the total size of the third-level of the index is *#total number of locations* $*$ *8B*. In total, it takes 9.8 GB to store the hash table-based index for 24 chromosomes of the human genome.

## 6.6 MinSeed Algorithm

We base our seeding algorithm, MinSeed, upon Minimap2's minimizer-based seeding algorithm (i.e., `mm_sketch`). A *minimizer* [265, 181, 182] is the smallest *k-mer* in a window of *w* consecutive k-mers. The goals of using minimizers, or <*w*,*k*>-minimizers, instead of the full set of k-mers, are to decrease the storage requirements of the index by storing fewer number of k-mers, and speeding up the queries that are made to this index. In Figure 6-4, we show an example of how <5,3>-minimizer of a sequence is selected among the full set of k-mers. After finding the 5 adjacent 3-mers, we sort them and select the smallest. In this example, sorting is done simply by the lexicographical order.

| Position | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Sequence | A | G | T | A | G | C | A |
| Full set of k-mers with minimizer in **red** | A | G | T | | | | |
| | | G | T | A | | | |
| | | | T | A | G | | |
| | | | | **A** | **G** | **C** | |
| | | | | | G | C | A |

**Figure 6-4: Example of finding the minimizers of a given sequence.**

MinSeed algorithm starts with computing the minimizers of a given query read. Even though using two loops such that the outer loop iterates over the query read while the inner loop finds the minimum k-mer within each window is an easy solution for minimizer

computation, using a queue that caches the previous minimum k-mers can avoid the inner loop and provide a $O(m)$ complexity algorithm, where $m$ is the length of the query read [181, 182, 150].

After finding the minimizers, MinSeed queries the hash table-based index stored in the memory to fetch the frequencies (i.e., *#locations*) of each minimizer. If the frequency of a minimizer is above the user-defined threshold, then it is discarded. If the minimizer is not discarded, then all the seed locations for that minimizer is fetched from the index in the memory.

After fetching all the seed locations, using the node ID and relative offset of the seed locations along with the relative offset of the corresponding minimizer within the query read, the rightmost and leftmost positions (i.e., right-extension and left-extension) of each seed are calculated. As we show in Figure 6-5, to find the leftmost position of the seed region ($x$), we need the start position of the minimizer within the query read ($a$), the start position of the seed ($c$), and the error rate ($E$). Similarly, to find the rightmost position of the seed region ($y$), we need the end position of the minimizer within the query read ($b$), the end position of the seed ($d$), the query read length ($m$), and the error rate ($E$). Finally, for each seed, the subgraph surrounded by these positions is fetched from the memory and provided as the output of the MinSeed algorithm.

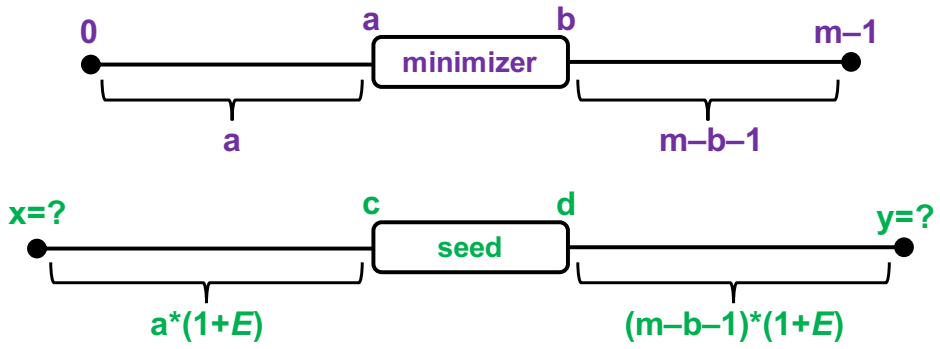



**Figure 6-5: Calculations for finding the candidate seed region.**

## 6.7 BitAlign Algorithm

After finding the subgraphs, our new sequence-to-graph alignment algorithm, BitAlign, performs distance calculation and traceback operations between the query read and the subgraph in order to find the *optimal alignment*. In order to provide an efficient, hardware-friendly, and low-cost solution, we generalize the bitvector-based sequence alignment algorithm, GenASM (Sections 4.5 and 4.6), for sequence-to-graph alignment and exploit the bit-parallelism the algorithm provides.

The major difference between sequence-to-sequence alignment and sequence-to-graph alignment is, in sequence-to-sequence alignment, we are only interested in the neighbor (i.e., previous/adjacent) text character whereas in sequence-to-graph alignment, due to possible incoming/outgoing edges from non-neighbor characters, we have to incorporate those edges as well. More formally, in the original GenASM algorithm, edges were implicit: the character at position `i < n-1` had a single *successor* at position `i+1`, and iteration `i` requires the results of `i+1`. In a graph however, edges must be stored explicitly: every node `i` has a set of successors $\{j_0, j_1, \ldots\}$, and iteration `i` now requires the results of all $j_x$. By topologically sorting the nodes, we ensure that $\forall x.i < j_x$, and thus the results of all successors are available in iteration `i`.

Thus, we modify the GenASM algorithm, such that while generating the bitvectors during the distance calculation step, we not only consider the previous text character's bitvectors (*oldR[d] bitvectors*), but also all bitvectors that are corresponding to the outgoing edges (or *hops*). For example, as we show in Figure 6-6, when generating the bitvectors for the dark blue-shaded node, we need both light blue-shaded nodes' bitvectors. On the other hand, when generating the bitvectors for the dark red-shaded node, we only need the light red-shaded node's bitvectors.

In order to generalize the GenASM algorithm for sequence-to-graph alignment, in BitAlign, we (1) first linearize the input subgraph (assuming it is topologically sorted), (2) store the (*R[d] bitvectors*) for all the text iterations (not just for the previous one;

*oldR[d]*), and (3) update how intermediate bitvectors (i.e., match, substitution, deletion, and insertion) are calculated in order to incorporate the hops as well. We show the new BitAlign algorithm in Algorithm 3. When calculating the deletion (*D*), substitution (*S*), and match (*M*) bitvectors, we take the hops into consideration, whereas when calculating the insertion (*I*) bitvector, we do *not* need to.

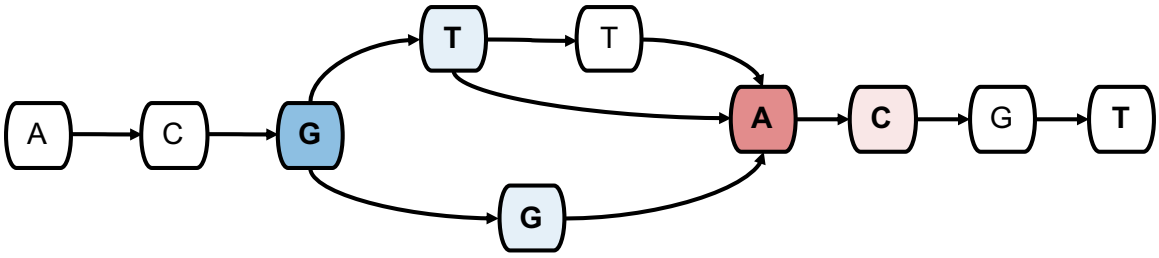



**Figure 6-6: Example of the dependency between different nodes of a graph when generating bitvectors.**

**Algorithm 3** BitAlign Algorithm

**Inputs:** `graph-nodes` (reference), `pattern` (query), `k` (edit distance threshold)
**Outputs:** `editDist` (minimum edit distance), `CIGARstr` (traceback output)

```
1: n ← length of linearized reference subgraph
2: m ← length of query pattern
3: PM ←genPatternBitmasks(pattern)                    ▷ pre-process the pattern
4: allR[n][d] ← 111.111                    ▷ init R[d] bitvectors for all characters
5: for i in (n-1):-1:0 do                              ▷ iterate over each graph node
6:     curChar ← graph-nodes[i].char
7:     curPM ← PM[curChar]                          ▷ retrieve the pattern bitmask
8:     R0 ← 111...111                          ▷ status bitvector for exact match
9:     for j in graph-nodes[i].successors do
10:        R0 ← ((R[j][0]<<1) | curPM) & R0
11:    allR[i][0] ← R0
12:    for d in 1:k  do
13:        I ← (allR[i][d-1]<<1)                                        ▷ insertion
14:        Rd ← I                                  ▷ status bitvector for d errors
15:        for j in graph-nodes[i].successors do
16:            D ← allR[j][d-1]                                         ▷ deletion
17:            S ← allR[j][d-1]<<1                                  ▷ substitution
18:            M ← (allR[j][d]<<1) | curPM                              ▷ match
19:            Rd ← D & S & M & Rd
20:        allR[i][d] ← Rd
21: <editDist, CIGAR> ← traceback(allR, graph-nodes, pattern)
```

For traceback in the style of GenASM, we need $3(k+1)$ bitvectors to be stored per edge in the graph. Since the number of edges in the graph can only be bounded very loosely, the potential memory footprint increases significantly, which is expensive to implement in hardware. We solve this problem by storing only $k+1$ bitvectors per *node*, from which the $3(k+1)$ bitvectors per *edge* can be regenerated on-demand during traceback. While this incurs a minor computational overhead, this modification helps us to decrease the memory footprint of the algorithm by $3\times$ when the graph is a path, and additional edges incur no memory overhead. Since memory is the main area and power cost of the alignment hardware, this tradeoff is very favorable.

## 6.8 SeGraM Hardware Design

In SeGraM, we *co-design* our new MinSeed algorithm for seeding and new BitAlign algorithm for sequence-to-graph alignment with specialized custom accelerators.

### 6.8.1 MinSeed Hardware

A MinSeed accelerator consists of: (1) three computation modules responsible for finding the minimizers from a query read, filtering the frequencies of minimizers if above a threshold, and finding the associated regions of every seed location by calculating the rightmost and leftmost positions; (2) three scratchpads for storing the query read, its minimizers, and seed locations; and (3) the memory interface, which handles the frequency, seed location, and subgraph accesses.

As we show in Figure 6-7, MinSeed accelerator gets the query read as the input and finds the subgraphs to align this query as the output. The computation modules are implemented with a simple logic since we require basic logical operations (e.g., comparisons, simple arithmetic operations, scratchpad R/W operations).

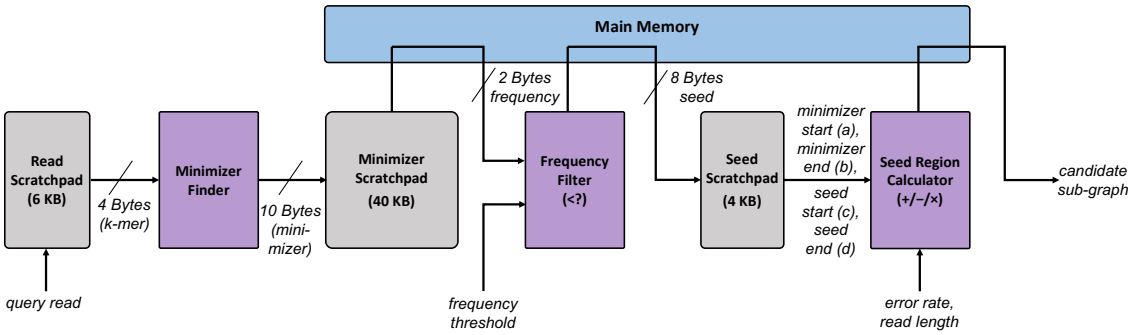



**Figure 6-7: Hardware design of MinSeed.**

### 6.8.2 BitAlign Hardware

We implement the distance calculation (DC) hardware of BitAlign as a linear cyclic systolic array-based accelerator. Since we need to incorporate the hops as well, in our new design, we use *hop queue registers* in order to feed the bitvectors of non-neighbor characters/nodes.

As we show in Figure 6-8, the generated $R[d]$ bitvector from each processing element (PE) is fed to the tail of the hop queue register of the current PE. Each hop queue register then provides the stored bitvectors as the $oldR[d]$ bitvectors to the same PE (required for the match bitvector's calculation) and as the $oldR[d-1]$ bitvectors to the next PE (required for the deletion and substitution bitvectors' calculation) in the next cycle.

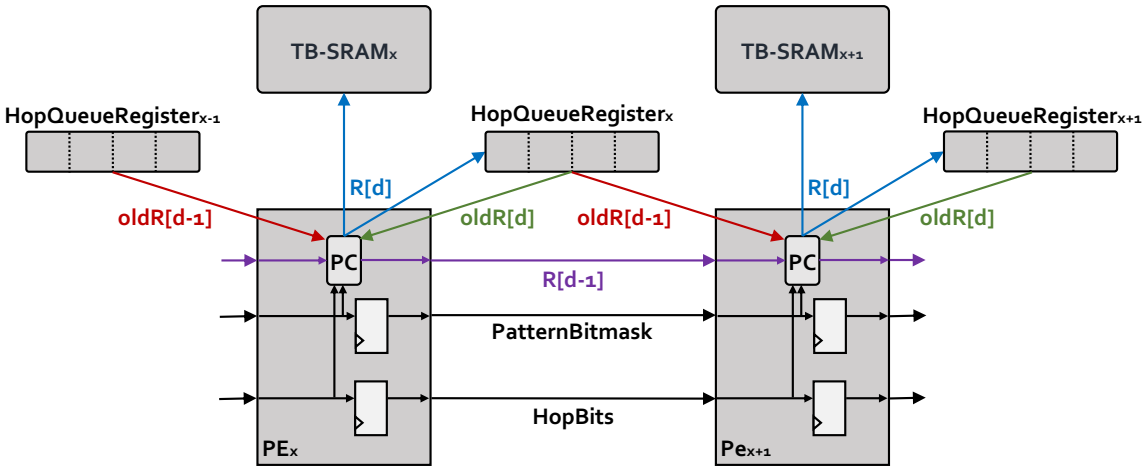
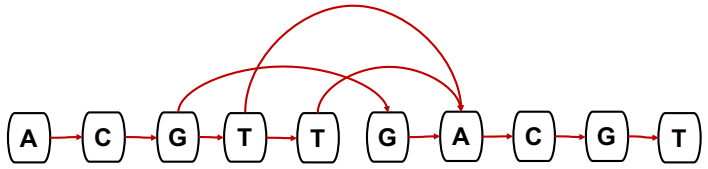



**Figure 6-8: Processing element (PE) design of BitAlign.**

We implement the successor relation as an adjacency matrix called *hopBits* (Figure 6-9). Based on the *hopBits* of the current text character, either the actual hop bitvector or all 1s bitvector is used when calculating the match, deletion, and substitution bitvectors of the current PE. $R[d-1]$ bitvector (required for the insertion bitvector's calculation) is directly provided by the previous PE (i.e., not through the hop queue registers). In order to decrease the size of each hop queue register, based on our empirical analysis, we limit the hop length to 12. Thus, each hop queue register contains 12 elements.




| NodeID | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| HopBits | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 |
| | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |
| | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 5 |
| | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 |
| | 0 | 0 | 0 | **1** | **1** | **1** | 0 | 0 | 0 | 0 | 7 |
| | 0 | 0 | 0 | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 8 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | 0 | 0 | 9 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | 0 | 10 |

**Figure 6-9: Linearized input subgraph and the generated hopBits.**

As we explain in Section 6.7, in order to decrease the memory footprint of the stored bitvectors required for the traceback (TB) execution, we only store the ANDed version of the intermediate bitvectors ($R[d]$) and re-compute the intermediate bitvectors (i.e., match, substitution, deletion, and insertion) during the TB execution. Thus, each element of the queue register has a length equal to the window size ($W$), instead of $3 * W$. Similarly, the size of each TB-SRAM for each PE decreases with this design choice. Another required change to the BitAlign hardware due to this design choice is for the TB accelerator design.

After reading the $R[d]$ bitvectors from the TB-SRAMs and before performing the bitwise comparisons to find the CIGAR character for the current traceback iteration, we need an additional step that re-generates the intermediate bitvectors using the $R[d]$ bitvectors, required for the TB execution.

Besides the TB-SRAMs, BitAlign also requires DC-SRAM to store the linearized reference graph, associated hopBits for each node, and the pattern bitmasks for the query read. For a 128-PE configuration with 128 bits of processing per PE, BitAlign requires a total of 24KB DC-SRAM storage. Also, each PE requires a total of 2KB TB-SRAM storage, with a single R/W port (128KB, in total). In each cycle, 128 bits of data (16B) is written to each TB-SRAM and to each hop queue register by each PE.

### 6.8.3 Overall System Design

Figure 6-10 shows the overall design of SeGraM. SeGraM is connected to a host system. The host transfers a single query read to SeGraM, which is buffered before being processed. We employ double buffering technique to hide the transfer latency. Our acceleration platform consists of four HBM2E stacks [154], each with 8 channels. Next to each HBM2E stack, we place one SeGraM module. A channel is exposed to SeGraM as a 256-bit wide port. The theoretical bandwidth available per 3D stack of HBM2E is 307 GB/s [154]. Thus, our complete accelerator design can leverage a peak theoretical bandwidth of 1.2 TB/s. By placing SeGraM in the same package as the four HBM2E stacks, we mimic the configuration of current commercial devices such as GPUs [231, 232] and FPGA boards [5, 1]. This in-package configuration allows SeGraM to have high-bandwidth memory access without limitations in area and thermal dissipation of other 3D-stacked memory technologies [133], where accelerators can be placed in the logic layer of the 3D stack. We replicate the content of each HBM2E stack (i.e., graph-based reference and hash tablebased index) among the 4 independent stacks. Within each stack, we distribute the graph and index structures of the 24 chromosomes based on their sizes among the 8 independent channels.

The main computation pipeline of SeGraM consists of MinSeed and BitAlign modules. A single SeGraM consists of 8 MinSeed modules that exploit data-level parallelism when performing seeding. Each MinSeed module has exclusive access to one HBM2E channel. This ensures full bandwidth exploitation without contention, and allows us to optimally balance bandwidth and MinSeed's compute throughput. The MinSeed module is responsible for finding the minimizers of a given query read and their associated seed locations that are fed to our BitAlign module. Each MinSeed module is connected to a single BitAlign module. The BitAlign module is responsible for performing alignment between each of the seeds reported by the MinSeed module and the query read.

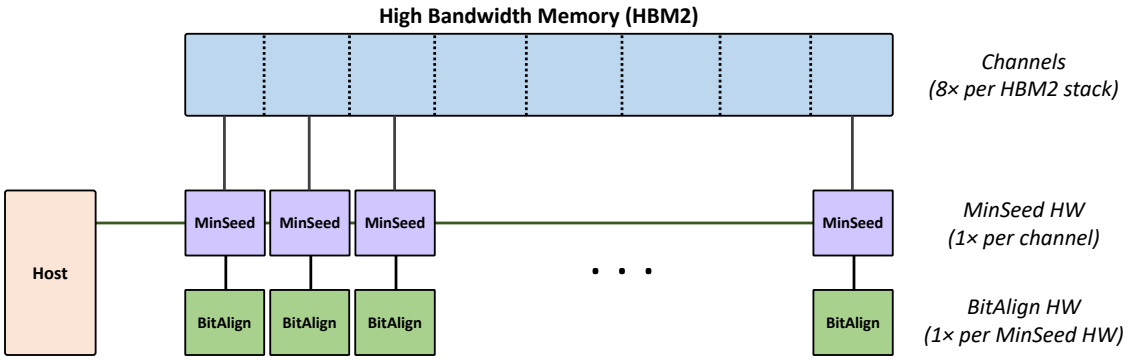



**Figure 6-10: Overall system design of SeGraM.**

We design the SeGraM mechanism in a pipelined fashion, such that we can hide the latency of our MinSeed accelerator when performing seeding while running sequence-to-graph alignment with our BitAlign accelerator. Thus, while BitAlign is running, MinSeed can fetch the next set of minimizers, frequencies, and seeds from the main memory, and write them to their associated scratchpads. In order to enable this, similar to the read scratchpad, we employ double buffering technique for the minimizer and seed scratchpads, not to overwrite the existing minimizers and seeds in those scratchpads that are being executed with the BitAlign accelerator, in parallel.

## 6.9 SeGraM as a Framework

With the help of the flexibility and decomposability of the SeGraM framework, we can run each accelerator (i.e., MinSeed and BitAlign) separately or we can run them together for end-to-end execution. Thus, we describe three use cases of SeGraM: (1) end-to-end sequence-to-graph mapping, (2) sequence-to-graph alignment, and (3) seeding.

**End-to-end sequence-to-graph mapping.** For sequence-to-graph mapping, the whole SeGraM design should be executed, since both seeding and alignment steps are required, as we explain in Section 2.6. Thus, for this use case, both MinSeed and BitAlign should be executed. Also, with the help of the inherited divide-and-conquer approach from the GenASM algorithms, we can use SeGraM for performing sequence-to-graph mapping for both short and long reads.

**Sequence-to-graph alignment.** Since as an input, BitAlign requires the subgraph as the reference and the query read as the pattern, it can also act as a sequence-to-graph aligner, without the need of an initial seeding tool/accelerator. Also, since sequence-to-sequence alignment is a special and simpler variant of sequence-to-graph alignment, BitAlign can also be used for that use case, when the linear input text is represented as a graph, where the nodes only have a single edge.

**Seeding.** Similarly, MinSeed only can be used as the seeding module for both graph-based mapping and linear traditional mapping. MinSeed is orthogonal to be coupled with any alignment tool or accelerator.

## 6.10 Evaluation Methodology

**Performance, Area and Power Analysis.** We synthesize and place & route the MinSeed and BitAlign accelerator datapaths using the Synopsys Design Compiler [4] with a typical 28nm low-power process. Our synthesis targets post-routing timing closure at 1GHz clock frequency. We use CACTI [315, 282] to estimate the area overhead and

power consumption of the scratchpad in MinSeed and BitAlign. We then use an in-house cycle-accurate simulator and a spreadsheet-based analytical model parameterized with the synthesis and memory estimations to drive the performance analysis.

**Baseline Tools.** We compare SeGraM with two state-of-the-art sequence-to-graph mappers: vg [99] and GraphAligner [262], running on an Intel® Xeon® Gold 6126 CPU [146] operating at 2.60GHz, with 64GB DDR4 memory. We run both tools with 12 threads. We measure the execution time and power consumption of the baseline tools. We measure the individual power consumed by each tool using Intel's PCM power utility [143]. We also compare BitAlign with a state-of-the-art sequence-to-graph aligner, PaSGAL [149], and also with three state-of-the-art sequence-to-sequence hardware aligners: Darwin [301], GenAx [95], and GenASM [279]. For all these four baselines, we use the numbers reported by their respective papers.

**Datasets.** We evaluate SeGraM using the latest major release of the human genome assembly, GRCh38 [3], as the starting reference genome. For the variations, we use 7 VCF files for HG001-007 from the GIAB project (v3.3.2) [138].

As the read datasets, we generate four sets of long reads (i.e., PacBio and ONT datasets) using PBSIM2 [237] and three sets of short reads (i.e., Illumina datasets) using Mason [126]. For the PacBio and ONT datasets, we have reads of length 10Kbp, each simulated with 5% and 10% error rates. The Illumina datasets have reads of length 100bp, 150bp, and 250bp, each simulated with a 1% error rate.

## 6.11 Results

### 6.11.1 Area and Power Analysis

Table 6-1 shows the area and power breakdown of the compute (i.e., logic) units and the memory components (i.e., scratchpads) in SeGraM, and the total area overhead and power consumption of (1) a single SeGraM accelerator (attached to a single channel), (2) 8

**Table 6-1: Area and power breakdown of SeGraM.**

| Component | Area ($mm^2$) | Power (mW) |
|---|---|---|
| MinSeed – Logic | 0.017 | 10.8 |
| Read Scratchpad (6 KB) | 0.009 | 1.9 |
| Minimizer Scratchpad (40 KB) | 0.061 | 6.9 |
| Seed Scratchpad (4 KB) | 0.006 | 2.5 |
| BitAlign – DC Logic with HopQueueRegisters (64 PEs) | 0.393 | 378.0 |
| BitAlign – TB Logic | 0.020 | 2.7 |
| Input Scratchpad (DC-SRAM; 24 KB) | 0.034 | 8.4 |
| Bitvector Scratchpad (TB-SRAMs; 128 KB) | 0.233 | 115.1 |
| **Total – 1 x SeGraM** | **0.773** | **526.3 (0.5 W)** |
| **Total – 8 x SeGraM** | **6.184** | **4210.4 (4.2 W)** |
| **Total – 32 x SeGraM** | **24.736** | **16841.6 (16.8 W)** |

SeGraM accelerators (in a single stack with 8 channels), and (3) 32 SeGraM accelerators (in 4 stacks). Our accelerators operate at 1GHz.

The area overhead of one SeGraM accelerator is 0.773 $mm^2$, and the power consumption of one SeGraM accelerator is 526 mW. We find that the main contributors for the area overhead and power consumption are (1) hopQueueRegisters since they constitute more than 60% of the area and power of BitAlign-DC logic, and (2) the bitvector scratchpads (TB-SRAMs). As we have one SeGraM accelerator per channel, the total area overhead of SeGraM attached to all 32 channels is 24.7 $mm^2$. Similarly, the total power consumption of 32 SeGraM accelerators is 16.8 W.

### 6.11.2 Analysis of SeGraM

We compare end-to-end execution of SeGraM with two state-of-the-art sequence-to-graph mapping tools, GraphAligner and vg. We compare both of the tools with SeGraM for both long and short reads. We measure the execution time and power consumption of the baseline tools for their seeding, filtering/chaining, and alignment steps only (i.e., we do not include the pre-processing steps, which are executed only once).

**Long Read Analysis.** Figure 6-11 shows the read mapping throughput (reads/sec) of SeGraM and GraphAligner, when aligning long noisy PacBio and ONT reads against the graph-based representation of the human reference genome. We show that, on average, SeGraM provides 8.8× throughput improvement over GraphAligner's 12-thread execution.

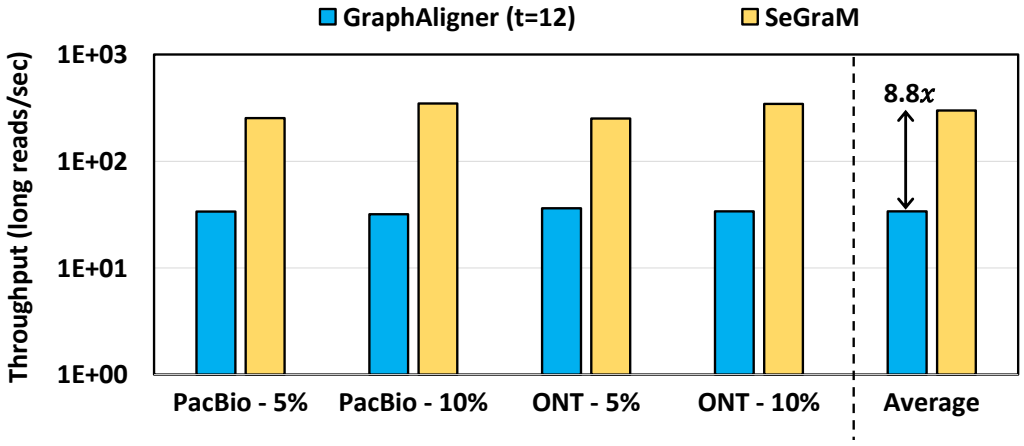



**Figure 6-11: Throughput comparison of SeGraM and GraphAligner for long reads.**

Figure 6-12 shows the read mapping throughput (reads/sec) of SeGraM and vg, when aligning long noisy PacBio and ONT reads against the graph-based representation of the human reference genome. We show that, on average, SeGraM provides 7.3× throughput improvement over vg's 12-thread execution.

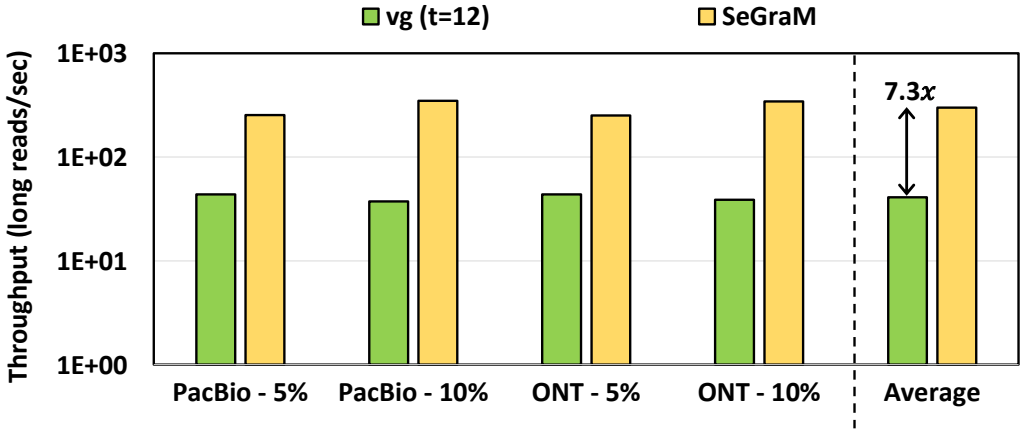



**Figure 6-12: Throughput comparison of SeGraM and vg for long reads.**

Based on our power analysis with long reads, we find that power consumption of GraphAligner is 83 W and power consumption of vg is 109 W for their 12-thread execution. Thus, SeGraM reduces the power consumption of GraphAligner and vg by 4.9× and 6.5× over their 12-thread execution.

**Short Read Analysis.** Figure 6-13 shows the read mapping throughput (reads/sec) of SeGraM and GraphAligner, when aligning short Illumina reads against the graph-based representation of the human reference genome. We show that, on average, SeGraM provides 168× throughput improvement over GraphAligner's 12-thread execution.

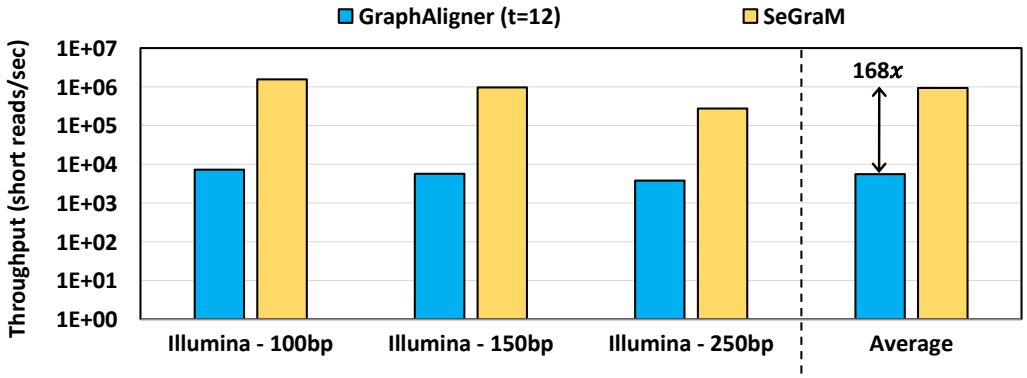



**Figure 6-13: Throughput comparison of SeGraM and GraphAligner for short reads.**

Figure 6-14 shows the read mapping throughput (reads/sec) of SeGraM and vg, when aligning short Illumina reads against the graph-based representation of the human reference

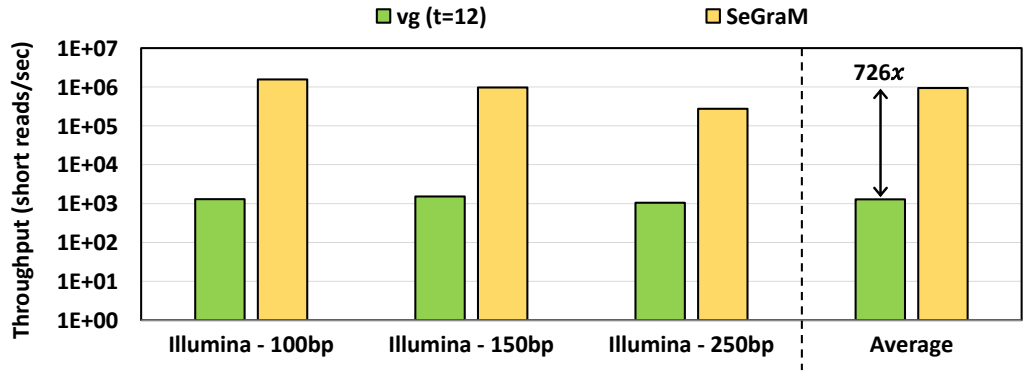



**Figure 6-14: Throughput comparison of SeGraM and vg for short reads.**

genome. We show that, on average, SeGraM provides 726× throughput improvement over vg's 12-thread execution.

Based on our power analysis with short reads, we find that power consumption of GraphAligner is 79 W and power consumption of vg is 83 W for their 12-thread execution. Thus, SeGraM reduces the power consumption of GraphAligner and vg by 4.7× and 4.9× over their 12-thread execution.

**Sources of Improvement.** The sources of large performance improvements in SeGraM are (1) the efficient and hardware-friendly underlying algorithms for both seeding and sequence-to-graph alignment, (2) carefully designed dedicated scratchpads based on the empirical data we collect for different data structures (e.g., graphs, minimizers, seeds), (3) hop queue registers, since they allow us to fetch all the bitvectors for the hops within a single cycle, and (4) our pipelined overall design, where we can hide the execution latency of MinSeed, with the BitAlign execution. Thus, even though we have to increase the sizes of the scratchpads to allow double buffering and add hop queue registers for not increasing the execution time required for sequence-to-graph alignment, these additional area and power overheads help us to provide large increase in throughput for both short and long reads.

### 6.11.3 Analysis of BitAlign

**Sequence-to-Graph Alignment.** As we explain in Section 6.9, BitAlign-only can be used for sequence-to-graph alignment, without the need of a preceding seeding tool/accelerator. We compare BitAlign with the state-of-the-art SIMD-based sequence-to-graph alignment tool, PaSGAL [149]. PaSGAL is composed of three main steps: (1) DP-fwd, where the input graph and query read are aligned using the dynamic programming based graph alignment approach to compute the ending position of the alignment, without running the traceback operation; (2) DP-rev, where graph and query are aligned in the reverse direction to compute the starting position of the alignment, again without running the traceback operation; and (3) Traceback, where using the starting and ending positions of

the alignment, the corresponding section of the score matrix is re-calculated and traceback is performed to find the optimal alignment.

Since the input of BitAlign is the subgraph and the query read, not the complete input graph, we *only* compare BitAlign with the third step of PaSGAL for a fair comparison. As we show in Figure 6-15, based on the results reported in [149] for the LRC-L1, LRC-L2, MHC1-M1, and MHC1-M2 datasets, SeGraM provides 41×–539× speedup over the 48-thread AVX512-supported execution of PaSGAL.

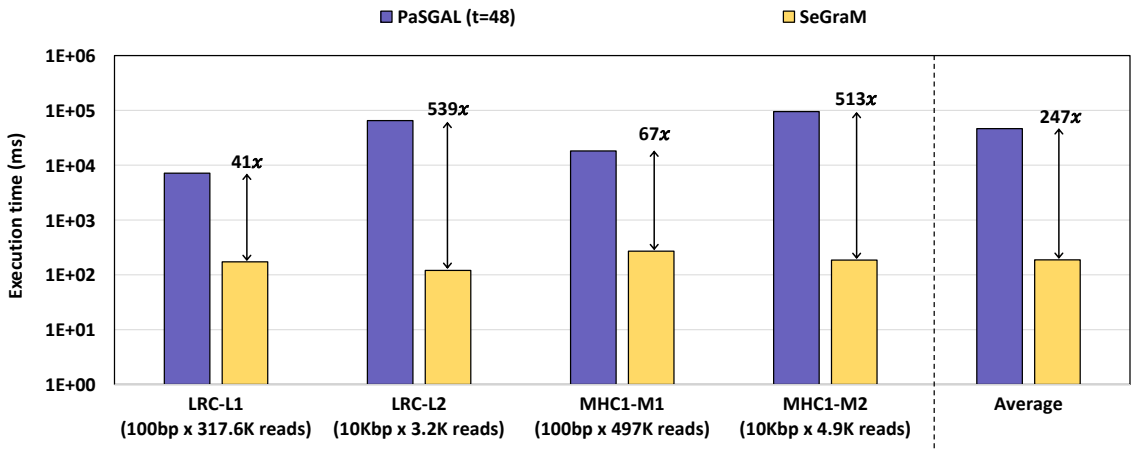



**Figure 6-15: Performance comparison of SeGraM and PaSGAL for sequence-to-graph alignment.**

Compared to PaSGAL, BitAlign shows significant speedup, especially for the long read datasets (i.e., LRC-L2 and MHC1-M2). The reason of this is the inherited divide-and-conquer approach that BitAlign follows. Instead of aligning the full subgraph and the query read, with the help of the windowing approach, BitAlign manages to decrease the complexity of sequence-to-graph alignment, and efficiently aligns both short and long reads.

**Sequence-to-Sequence Alignment.** Even though sequence-to-sequence alignment tools or accelerators cannot be used for sequence-to-graph alignment since they do *not* consider hops and only consider neighbor text characters, sequence-to-graph alignment tools or accelerators can be used for the traditional sequence-to-sequence alignment problem. Thus, BitAlign can be used for both sequence-to-sequence alignment and sequence-to-

graph alignment. The cost of more functionality in BitAlign is the extra hop queue registers. However, with the help of these additional memory components, we do *not* sacrifice any performance.

To show the efficiency of BitAlign for this special use case of sequence-to-graph alignment (i.e., sequence-to-sequence alignment), we compare BitAlign with the state-of-the-art hardware accelerators for sequence-to-sequence alignment: GACT of Darwin [301], SillaX of GenAx [95], and GenASM [279]. GACT is optimized for long reads, SillaX is optimized for short reads, and GenASM is optimized for both short and long reads. We use the optimum configuration of each accelerator reported in their corresponding papers.

Based on our analysis with long reads, we find that, on average, SeGraM provides 4.8× and 1.2× throughput improvement over GACT of Darwin and GenASM, respectively, while having 1.9× and 5.2× higher power consumption, and 1.4× and 2.3× higher area overhead. For short reads, we find that, on average, SeGraM provides 2.4× and 1.3× throughput improvement over SillaX of GenAx and GenASM, respectively.

### 6.11.4 Analysis of MinSeed

As we explain in Section 6.8.3, with the help of our pipelined design, MinSeed execution is not on the critical path of the overall SeGraM execution. However, since MinSeed selects the candidate seed locations and sends them to BitAlign for the final alignment, it plays a critical role on the overall sensitivity of our approach. Since MinSeed does *not* perform any filtering approach to reduce the number of candidate seed regions that are sent for alignment (except discarding the seeds that have higher frequency than the threshold, which is an optimization that baseline tools already implement), SeGraM does not decrease the sensitivity of the overall sequence-to-graph mapping execution.

Even though not having a filtering mechanism increases the number of candidate seed regions that are sent for the expensive alignment step, with our highly-efficient BitAlign accelerator, we alleviate this bottleneck that exists in other existing tools and provide significant improvement over the baseline tools, as we show in Section 6.11.2. For example,

for a long read dataset, GraphAligner decreases the number of seeds extended from 77M to 48K with its filtering/chaining approaches. On the other hand, MinSeed sends 35M seeds to our BitAlign accelerator, but still provides higher throughput than GraphAligner. Similarly, for a short read dataset, GraphAligner decreases the number of seeds extended from 828K to 11K, while MinSeed sends 375K seeds to our BitAlign accelerator, and still provides much higher throughput. Thus, we show that SeGraM provides a highly sensitive and also high-performance solution for sequence-to-graph mapping.

## 6.12 Related Work

To our knowledge, we are the first to propose (1) a hardware acceleration framework for sequence-to-graph mapping (SeGraM), (2) a hardware accelerator for minimizer-based seeding (MinSeed), and (3) a hardware accelerator for sequence-to-graph alignment (BitAlign). No prior work has studied hardware for genome graph processing.

**Software Tools for Sequence-to-Graph Mapping.** Even though genome graphs gain attention recently, there are only a few tools available specialized for sequence-to-graph mapping or alignment. Examples of sequence-to-graph mapping tools are GraphAligner [262], vg [99], and HISAT2 [163]. There are also some works which focus on alignment only, without an indexing and seeding step, such as PaSGAL [149] and abPOA [98]. However, these are all software-based tools.

**Hardware Accelerators for Genome Sequence Analysis.** Existing hardware accelerators for genome sequence analysis focus on accelerating only the traditional read mapping (i.e., sequence-to-sequence) pipeline, and cannot support genome graphs as their inputs. For example, ERT [291], NEST [132], SaVI [173], and MEDAL [131] accelerate the seeding step of sequence-to-sequence mapping. Darwin [301], GenAx [95], GenASM [279], and SeedEx [96] accelerate read alignment with only a single reference genome. These accelerators have no way to track the multiple paths that need to be traversed in a graph, and cannot be easily modified to support multiple path tracking. SeGraM builds upon hardware

components of GenASM, and incorporates new algorithms and hardware, to efficiently support genome graph mapping and alignment, and can also support sequence-to-sequence mapping (as a graph where each node has only one outgoing edge).

There are also processing-in-memory (PIM) based accelerators for genome sequence analysis, such as GRIM-Filter [164], RAPID [117], PIM-Aligner [29], RADAR [130], FindeR [334], and AligneR [333]. However, similar to read alignment accelerators, they are tuned for a single linear reference only, and cannot support genome graphs. Besides these accelerators mostly focusing on the read mapping steps, there are also other PIM-based accelerators that focus on other steps of the pipeline [280], such as PIM-Assembler [28] for genome assembly and Helix [197] for nanopore basecalling.

## 6.13 Summary

Genome graphs are emerging representations for the DNA of a population, and overcome the biases present in traditional genome sequence analysis, where a single reference genome is used. Unfortunately, the additional overheads of sequence-to-graph mapping exacerbate the read mapping bottleneck in the genome sequence analysis pipeline. To alleviate this, we propose SeGraM, the first acceleration framework for sequence-to-graph mapping and alignment. For SeGraM, we co-design algorithms and accelerators for memory-efficient minimizer-based seeding and bitvector-based, highly-parallel sequence-to-graph alignment.

For sequence-to-graph mapping with long reads, we find that SeGraM achieves 8.8× and 7.3× speedup over 12-thread execution of state-of-the-art sequence-to-graph mapping tools (GraphAligner and vg, respectively), while reducing power consumption by 4.9× and 6.5×. For sequence-to-graph mapping with short reads, we find that SeGraM achieves 168× and 726× speedup over 12-thread execution of GraphAligner and vg, respectively, while reducing power consumption by 4.7× and 4.9×. For sequence-to-graph alignment, we show that BitAlign provides 41×–539× speedup over PaSGAL, a state-of-the-art sequence-

to-graph alignment tool. We conclude that SeGraM is a high-performance and efficient hardware acceleration framework, which can accelerate multiple steps of the sequence-to-graph mapping pipeline, and of the traditional read mapping pipeline.

# Chapter 7

# Importance of Accelerating Genome Sequence Analysis

We believe the long-term impact of GenASM, BitMAc, SeGraM, and other works that propose hardware acceleration for genome sequence analysis is three-fold:

**Enabling Portable, Fast, and Efficient Genome Sequence Analysis.** Recent advances have enabled genome sequencing anywhere in the world with cheap, portable sequencing machines (e.g., ONT's MinION). Soon, even smaller sequencing devices can enable sequencing using smartphones. Such readily available sequencing technologies can open up several new applications, such as bringing personalized medicine to rural or remote areas, near-patient testing, and rapid infection diagnosis and outbreak tracing (e.g., COVID-19 [316, 122, 153, 61], Ebola [257, 113], Zika [84]). However, these applications require memory-efficient, low-power, and area-efficient systems to process the generated genome sequence data, as laptops and mobile phones have limited resources (e.g., greater memory constraints, limited battery life). Our approach of co-designing scalable and memory-efficient algorithms with area- and power-efficient hardware accelerators is an important milestone, allowing genome sequence analysis to be performed in highly-resource-constrained environments. Our genomics accelerators can even be implemented

in the sequencing machine itself, eliminating expensive sequencer-to-computer data movement and providing a single embedded solution for portable sequencing and sequence analysis.

**Rapid Genome Sequence Analysis for Pandemics.** Rapid genome sequence analysis plays a critical role during pandemics such as the current COVID-19 (i.e., SARS-CoV-2) crisis in 2020-2021 [85, 129, 114]. Rapid analysis can (1) enable the quick detection of the virus in human DNA samples; (2) enable the rapid identification of the mutations, sources, and transmission modes of the virus; (3) help with the development of new treatments; and (4) help uncover why some people experience more severe symptoms and higher mortality than others. Given the fast pace at which viruses can proliferate and mutate during a pandemic, there is a need to perform large volumes of viral genomic analysis rapidly and widely, as lost time or limited availability can hinder tracking and harm our ability to control spread and mutations. Today, rapid genome sequence analysis is bottlenecked by the limited computational power and memory bandwidth of existing systems. We believe it is more important than ever to overcome these bottlenecks through the development of high-efficiency, low-cost solutions. Beyond the benefits that our genomics accelerators already yield, we hope that our co-design approach sparks further research from both academia and industry on developing even more powerful and efficient solutions for rapid genome sequence analysis of viruses.

**Reducing Genomic Accelerator Costs with Multi-Purpose Frameworks.** While there is a pressing need for genomic sequence analysis hardware, any fixed-function hardware incurs high per-unit costs, as the non-recurring engineering (NRE) costs can be amortized over only the number of platforms that perform the specific function. To significantly lower NRE costs, we design GenASM and BitMAc to provide substantial benefits for generic approximate string matching (a widely-used primitive for any text search or error-aware pattern matching), while still optimizing their designs to maximize benefits for genomic use cases. We believe that such an approach, with flexible frameworks that can serve as general-purpose accelerators but include domain-specific optimizations,

opens a promising pathway for a low-cost acceleration of other tasks. For example, other bioinformatics workloads (e.g., a graph processing acceleration framework for genome assembly, a neural network acceleration framework for nanopore basecalling or variant calling [253]) can take a similar approach, making what would otherwise be high-cost hardware much cheaper, and addressing key cost concerns in the healthcare industry. We hope that our approach of reusing genome sequence analysis frameworks for general-purpose acceleration will inspire future designers to consider NRE and incorporate general-purpose support into their accelerators.

# Chapter 8

# Conclusions and Future Directions

## 8.1 Conclusions

In this dissertation, we characterize the real-system behavior of the genome sequence analysis pipeline and its associated tools, expose the bottlenecks and tradeoffs of the pipeline and tools, and co-design fast and efficient algorithms along with scalable and energy-efficient customized hardware accelerators for the key pipeline bottlenecks to enable faster genome sequence analysis. Our goals are to (1) understand where the current tools and algorithms do not perform well in order to develop better tools and algorithms, and (2) understand the limitations of existing hardware systems when running these tools and algorithms in order to design efficient customized accelerators. Towards this end, we propose four major works.

First, we present the first work that analyzes state-of-the-art tools associated with each step of the genome assembly pipeline using long reads. We analyze the tools in multiple dimensions that are important for both developers and users/practitioners: accuracy, performance, memory usage and scalability. We reveal new bottlenecks and tradeoffs that

different combinations of tools and different underlying systems lead to, based on our extensive experimental analyses. We also provide guidelines for both practitioners, such that they can determine the appropriate tools and tool combinations that can satisfy their goals, and tool developers, such that they can make design choices to improve current and future tools.

Second, we propose GenASM, the first approximate string matching (ASM) acceleration framework for genome sequence analysis. GenASM performs bitvector-based ASM, which can efficiently accelerate multiple steps of genome sequence analysis. We modify the underlying ASM algorithm (Bitap) to significantly increase its parallelism and reduce its memory footprint. Using this modified algorithm, we design the first hardware accelerator for Bitap. Our hardware accelerator consists of specialized systolic-array-based compute units and on-chip SRAMs that are designed to match the rate of computation with memory capacity and bandwidth, resulting in an efficient design whose performance scales linearly as we increase the number of compute units working in parallel. We demonstrate that GenASM provides significant performance and power benefits for three different use cases in genome sequence analysis. First, GenASM accelerates read alignment for both long reads and short reads. For long reads, GenASM outperforms state-of-the-art software and hardware accelerators by 116× and 3.9×, respectively, while reducing power consumption by 37× and 2.7×. For short reads, GenASM outperforms state-of-the-art software and hardware accelerators by 111× and 1.9×. Second, GenASM accelerates pre-alignment filtering for short reads, with 3.7× the performance of a state-of-the-art pre-alignment filter, while reducing power consumption by 1.7× and significantly improving the filtering accuracy. Third, GenASM accelerates edit distance calculation, with 22–12501× and 9.3–400× speedups over the state-of-the-art software library and FPGA-based accelerator, respectively, while reducing power consumption by 548–582× and 67×. We also briefly discuss four other use cases that can benefit from GenASM.

Third, we propose BitMAc, which is an FPGA-based prototype for GenASM. In BitMAc, we map our GenASM algorithms on Stratix 10 MX FPGA with a state-of-the-art 3D-stacked

memory (HBM2), where HBM2 offers high memory bandwidth and FPGA resources offer high parallelism by instantiating multiple copies of the GenASM accelerators. After re-modifying the GenASM algorithms for a better mapping to existing FPGA resources, we show that BitMAc provides 64% logic utilization and 90% on-chip memory utilization, while having 48.9 W of total power consumption. We compare BitMAc with state-of-the-art CPU-based and GPU-based read alignment tools. Compared to the alignment steps of the CPU-based read mappers, (1) for long reads, BitMAc provides 761× and 136× speedup, while reducing power consumption by 1.9× and 2.0×, and (2) for short reads, BitMAc provides 92× and 130× speedup, while reducing power consumption by 2.2× and 2.0×. We also show that BitMAc provides significant speedup compared to the GPU-based baseline, while reducing the power consumption.

Fourth, we propose SeGraM, the first hardware acceleration framework for sequence-to-graph mapping and alignment. Reference genomes are conventionally represented as a linear sequence. However, this linear representation of the reference genome results with ignoring the variations that exist in a population (i.e., genetic diversity) and introducing biases for the downstream analysis. To address these limitations, recently, graph-based representations of the genomes (i.e., *genome graphs*) have gained attention. As shown in many prior works [18, 301, 279, 95, 166, 21, 111, 96, 36, 217, 164, 175, 157, 158], sequence-to-sequence mapping is one of the major bottlenecks of the genome sequence analysis pipeline and need to be accelerated using specialized hardware. Since graph-representation of the genome is much more complex than the linear representation, sequence-to-graph mapping is placing a greater pressure on this bottleneck. Thus, in this work, our goal is to design a high-performance, scalable, power- and area-efficient hardware accelerator for sequence-to-graph mapping that support both short and long reads. We base SeGraM on a memory-efficient minimizer-based seeding algorithm and a bitvector-based, highly-parallel sequence-to-graph alignment algorithm. We *co-design* both of our algorithms with high-performance, area- and power-efficient hardware accelerators. SeGraM consists of two components: (1) MinSeed, which provides hardware support to execute our minimizer-

based seeding algorithm, and (2) BitAlign, which provides hardware support to execute our bitvector-based sequence-to-graph alignment algorithm. For sequence-to-graph mapping with long reads, we find that SeGraM achieves 8.8× and 7.3× speedup over 12-thread execution of state-of-the-art sequence-to-graph mapping tools (GraphAligner and vg, respectively), while reducing power consumption by 4.9× and 6.5×. For sequence-to-graph mapping with short reads, we find that SeGraM achieves 168× and 726× speedup over 12-thread execution of GraphAligner and vg, respectively, while reducing power consumption by 4.7× and 4.9×. For sequence-to-graph alignment, we show that BitAlign provides 41×–539× speedup over PaSGAL, a state-of-the-art sequence-to-graph alignment tool.

Overall, we demonstrate that genome sequence analysis can be accelerated by co-designing scalable and energy-efficient customized accelerators along with efficient algorithms for the key steps of genome sequence analysis.

## 8.2 Future Research Directions

This dissertation opens new avenues for genomics research. In this section, we describe several such promising research directions in which the ideas and approaches in this dissertation can be extended to provide more functionality or to accelerate other key steps of the genome sequence analysis pipeline.

### 8.2.1 Algorithmic Enhancements to GenASM/BitAlign for Broader Functionality

As we explain in Chapter 4, GenASM is the first work that enhances and accelerates the Bitap algorithm for approximate string matching. We modify Bitap to add efficient support for long reads and enable parallelism within each ASM operation. We also propose the first Bitap-compatible traceback algorithm. Later, as we explain in Chapter 6, we further extend the GenASM algorithms and propose BitAlign, a novel bitvector-based

sequence-to-graph alignment algorithm. However, currently, both the GenASM algorithm and the BitAlign algorithm have a limitation, which affects their accuracy: they only support Levenshtein distance (i.e., edit distance) calculation [179], where each error (i.e., substitution, insertion or deletion) has the same cost (i.e., 1). One future work would be to extend both GenASM and BitAlign to fully support[1] different costs for each error type and affine gap penalty model, where gap openings and gap extensions are penalized differently. Once the algorithms are modified with this scoring extension, the hardware accelerators need to be modified to support these algorithmic changes. In order to efficiently support the changes, multiple components of the accelerators may need to be modified or more significantly redesigned.

### 8.2.2 End-to-End Acceleration of the Mapping Pipeline

As we explain in Section 4.8, GenASM provides support for the read alignment and pre-alignment filtering steps of the read mapping (i.e., sequence-to-sequence mapping) pipeline (Section 2.4). Even though GenASM is orthogonal to any indexing and seeding approach, due to Amdahl's Law, it is preferable to accelerate the entire read mapping pipeline rather than its individual steps. Similar to the approaches followed by Illumina's DRAGEN platform [134] and NVIDIA's Parabricks platform [230], in order to obtain larger amounts of speedup, one future work would be extending our GenASM work such that all of the steps of the read mapping pipeline would be accelerated as a complete hardware accelerator design. This end-to-end design would also help us to reduce the high amount of data movement that takes place while moving data between different compute units that perform different steps of the pipeline.

Similarly, as we explain in Section 6.9, SeGraM provides support for the seeding and the sequence-to-graph alignment steps of the sequence-to-graph mapping pipeline (Section 2.6). Thus, another promising future work would be to incorporate the pre-processing steps of

[1] GenASM currently offers partial support for non-unit costs for different edits and the affine gap penalty model (Section 4.6).

sequence-to-graph mapping and also implementing efficient pre-alignment filters as part of our SeGraM design for a more comprehensive and efficient end-to-end design.

### 8.2.3 Bottleneck Analysis and Acceleration of Assembly with Long Reads

With the emergence of long read sequencing technologies (ONT [238] and PacBio [244]), *de novo* assembly becomes a promising way of constructing the original genome. When we analyze the genome assembly pipeline using nanopore (ONT) sequence data (Chapter 3), we show that assembly is one of the most computationally-expensive steps of the pipeline. We also show that there is a tradeoff between accuracy and performance when deciding on the appropriate tool for this step. Thus, we believe that there is a need to design an accelerator for generic graph processing algorithms that includes specialized support for the assembly step of the pipeline, which will provide both high performance and high accuracy.

The expected first step would be to comprehensively analyze the current state-of-the-art long read assembly tools and revealing the bottlenecks in terms of performance, scalability, and accuracy. This would enable to explore possible algorithmic changes and different acceleration mechanisms (e.g., specialized accelerators, in-memory processing engines, and SIMD architecture) to resolve the bottlenecks. Even though many prior works (e.g., [330, 328, 11, 10, 218, 288, 278, 229, 115, 161, 64, 327, 329, 102, 331, 243, 120, 63, 332, 62, 33, 74, 308, 162, 309]) have proposed hardware accelerators for generic graph processing algorithms, efficiently using these accelerators for genome assembly is an important but unexplored research problem. Thus, exploring state-of-the-art graph processing accelerators and analyzing their suitability for the requirements of the assembly step would be beneficial. A related direction is to co-design an assembly algorithm with an efficient hardware accelerator and evaluate this design with real or simulated long read datasets to assess the performance and accuracy of the approach. Furthermore, exploring how to support different steps of the genome sequence analysis pipeline as well as generic graph processing algorithms with the proposed design would be beneficial.

## 8.3 Final Concluding Remarks

In this dissertation, we have demonstrated that *genome sequence analysis can be accelerated by co-designing scalable and energy-efficient customized accelerators along with efficient algorithms for the key steps of genome sequence analysis.* First, we comprehensively analyze the tools in the genome assembly pipeline for long reads in multiple dimensions (i.e., accuracy, performance, memory usage, and scalability), uncovering bottlenecks and tradeoffs that different combinations of tools and different underlying systems lead to. We show that we need high-performance, memory-efficient, low-power, and scalable designs for genome sequence analysis in order to exploit the advantages that genome sequencing provides. Second, we propose GenASM, an acceleration framework that builds upon bitvector-based approximate string matching (ASM) to accelerate multiple steps of the genome sequence analysis pipeline. We co-design our highly-parallel, scalable and memory-efficient algorithms with low-power and area-efficient hardware accelerators. Third, we implement an FPGA-based prototype for GenASM, where state-of-the-art 3D-stacked memory (HBM2) offers high memory bandwidth and FPGA resources offer high parallelism by instantiating multiple copies of the GenASM accelerators. Fourth, we propose SeGraM, the first hardware acceleration framework for sequence-to-graph mapping and alignment. SeGraM enables the efficient mapping of a sequenced genome to a graph-based reference, providing more comprehensive and accurate genome sequence analysis. We conclude and hope that this dissertation inspires future work in co-designing algorithms and hardware together to create powerful frameworks that accelerate other genomics workloads and emerging applications.

# Other Works of the Author

Throughout the course of my Ph.D. study, I have worked on several different topics with many fellow graduate students from Carnegie Mellon University, ETH Zurich, and other institutions. In this chapter, I would like to acknowledge these works.

I have worked on a number of other projects on genomics. In collaboration with Jeremie S. Kim, we propose GRIM-Filter [164], a novel seed location filtering algorithm, which is optimized to exploit 3D-stacked memory systems that integrate computation within a logic layer stacked under memory layers, to perform processing-in-memory (PIM). We also propose AirLift [165], a fast and comprehensive technique for remapping alignments from one genome to another. AirLift greatly reduces the time to perform end-to-end BAM-to-BAM [139] (i.e., binary alignment/map format; binary version of SAM (sequence alignment/map format [239, 141, 183]) remapping on a read set from one reference genome to another while maintaining high accuracy and comprehensiveness that is comparable to fully mapping the read set to the new reference.

In collaboration with Can Firtina, we propose Apollo [91], the first machine learning-based universal technology-independent assembly polishing algorithm. Apollo enables all available reads to contribute to assembly polishing and scales well to polish an assembly of any size (e.g. both small and large genome assemblies) within a single run. Apollo corrects errors in an assembly by using read-to-assembly alignment regardless of the sequencing technology used to generate reads.

In collaboration with Mohammed Alser, we propose a survey on state-of-the-art algorithmic methods and hardware-based acceleration approaches for genome analysis [16]. We cover the algorithmic approaches that exploit the structure of the genome as well as the structure of the underlying hardware, and the hardware-based acceleration approaches that exploit specialized microarchitectures or various execution paradigms (e.g., processing inside or near memory).

I have also worked on a number of non-genomics focused projects. In collaboration with Saugata Ghose, we propose an experimental study, where we rigorously analyze the combined DRAM–workload behavior for 9 different DRAM types and 115 modern applications and multiprogrammed workloads [104].

In collaboration with Gagandeep Singh, we propose a work, where we leverage an FPGA coupled with high-bandwidth memory (HBM) for improving the pre-alignment filtering step of genome analysis and representative kernels from a weather prediction model [284].

# Bibliography


[1] ADM-PCIE-9H7-High-Speed Communications Hub. https://www.alpha-data.com/dcp/products.php?product=adm-pcie-9h7.

[2] GFA: Graphical Fragment Assembly (GFA) Format Specification. https://github.com/GFA-spec/GFA-spec.

[3] NCBI: GRCh38.p13. https://www.ncbi.nlm.nih.gov/assembly/GCA_000001405.28.

[4] Tool from Synopsys, Design Compiler (Version L-2016.03-SP2). https://www.synopsys.com.

[5] Virtex UltraScale+. https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html.

[6] Computational Pan-Genomics: Status, Promises and Challenges. *Briefings in Bioinformatics*, 2018.

[7] 1000 Genomes Project Consortium and others. A Global Reference for Human Genetic Variation. *Nature*, 526(7571):68, 2015.

[8] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute Caches. In *HPCA*, 2017.

[9] Nauman Ahmed, Jonathan Lévy, Shanshan Ren, Hamid Mushtaq, Koen Bertels, and Zaid Al-Ars. GASAL2: A GPU Accelerated Sequence Alignment Library for High-Throughput NGS Data. *BMC Bioinformatics*, 2019.

[10] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *ISCA*, 2015.

[11] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *ISCA*, 2015.

[12] New basecaller now performs 'raw basecalling', for improved sequencing accuracy. https://nanoporetech.com/about-us/news/new-

basecaller-now-performs-raw-basecalling-improved-sequencing-accuracy.

[13] Can Alkan, Bradley P Coe, and Evan E Eichler. Genome Structural Variation Discovery and Genotyping. *Nature Reviews Genetics*, 2011.

[14] Can Alkan, Jeffrey M Kidd, Tomas Marques-Bonet, et al. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature Genetics*, 41(10):1061–1067, 2009.

[15] Can Alkan, Saba Sajjadian, and Evan E Eichler. Limitations of next-generation genome sequence assembly. *Nature Methods*, 8(1):61–65, 2011.

[16] Mohammed Alser, Zülal Bingöl, Damla Senol Cali, Jeremie Kim, Saugata Ghose, Can Alkan, and Onur Mutlu. Accelerating Genome Analysis: A Primer on an Ongoing Journey. *IEEE Micro*, 2020.

[17] Mohammed Alser, Hasan Hassan, Akash Kumar, Onur Mutlu, and Can Alkan. Shouji: A Fast and Efficient Pre-alignment Filter for Sequence Alignment. *Bioinformatics*, 2019.

[18] Mohammed Alser, Hasan Hassan, Hongyi Xin, et al. GateKeeper: A New Hardware Architecture for Accelerating Pre-alignment in DNA Short Read Mapping. *Bioinformatics*, 2017.

[19] Mohammed Alser, Onur Mutlu, and Can Alkan. MAGNET: Understanding and Improving the Accuracy of Genome Pre-alignment Filtering. *IPSI Transactions on Internet Research*, 2017.

[20] Mohammed Alser, Jeremy Rotman, Kodi Taraszka, Huwenbo Shi, Pelin Icer Baykal, Harry Taegyun Yang, Victor Xue, Sergey Knyazev, Benjamin D Singer, Brunilda Balliu, David Koslicki, Pavel Skums, Alex Zelikovsky, Can Alkan, Onur Mutlu, and Serghei Mangul. Technology Dictates Algorithms: Recent Developments in Read Alignment. *Genome Biology*, 2021.

[21] Mohammed Alser, Taha Shahroodi, Juan Gómez-Luna, Can Alkan, and Onur Mutlu. SneakySnake: a fast and accurate universal genome pre-alignment filter for CPUs, GPUs and FPGAs. *Bioinformatics*, 2020.

[22] Stephen F Altschul and Bruce W Erickson. Optimal Sequence Alignment using Affine Gap Costs. *Bulletin of Mathematical Biology*, 1986.

[23] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 1990.

[24] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs. *Nucleic Acids Research*, 1997.

[25] Maria Jesus Alvarez-Cubero, Maria Saiz, Belén Martínez-García, Sara M Sayalero, Carmen Entrala, Jose Antonio Lorente, and Luis Javier Martinez-Gonzalez. Next generation sequencing: an application in forensic sciences? *Annals of Human Biology*, 44(7):581–592, 2017.

[26] Shanika L Amarasinghe, Shian Su, Xueyi Dong, Luke Zappia, Matthew E Ritchie, and Quentin Gouil. Opportunities and Challenges in Long-Read Sequencing Data Analysis. *Genome Biology*, 2020.

[27] J.M. Ambler, S. Mulaudzi, and N. Mulder. GenGraph: a python module for the simple generation and manipulation of genome graphs. *BMC Bioinformatics*, 2019.

[28] Shaahin Angizi, Naima Ahmed Fahmi, Wei Zhang, and Deliang Fan. PIM-Assembler: A Processing-in-Memory Platform for Genome Assembly. In *DAC*, 2020.

[29] Shaahin Angizi, Jiao Sun, Wei Zhang, and Deliang Fan. PIM-Aligner: A Processing-in-MRAM Platform for Biological Sequence Alignment. In *DATE*, 2020.

[30] Simon Ardui, Adam Ameur, Joris R Vermeesch, and Matthew S Hestand. Single molecule real-time (SMRT) sequencing comes of age: Applications and utilities for medical diagnostics. *Nucleic Acids Research*, 2018.

[31] ARM Developer. AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite. `https://developer.arm.com/documentation/ihi0022/e`.

[32] Euan A Ashley. Towards Precision Medicine. *Nature Reviews. Genetics*, 2016.

[33] Osama G Attia, Tyler Johnson, Kevin Townsend, Philip Jones, and Joseph Zambreno. CyGraph: A reconfigurable architecture for parallel breadth-first search. In *IEEE International Parallel & Distributed Processing Symposium Workshops*, 2014.

[34] Ricardo Baeza-Yates and Gaston H Gonnet. A New Approach to Text Searching. *CACM*, 1992.

[35] Subho Sankar Banerjee, Mohamed El-Hadedy, Jong Bin Lim, Zbigniew T. Kalbarczyk, Deming Chen, Steven S. Lumetta, and Ravishankar K. Iyer. ASAP: Accelerated short-read alignment on programmable hardware. *IEEE Transactions on Computers*, 2019.

[36] Zülal Bingöl, Mohammed Alser, Onur Mutlu, Ozcan Ozturk, and Can Alkan. GateKeeper-GPU: Fast and Accurate Pre-Alignment Filtering in Short Read Mapping. *arXiv preprint arXiv:2103.14978*, 2021.

[37] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[38] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T Malladi, Hongzhong Zheng, and Onur Mutlu. CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators. In *ISCA*, 2019.

[39] Claus Børsting and Niels Morling. Next Generation Sequencing and Its Applications in Forensic Genetics. *Forensic Science International: Genetics*, 2015.

[40] Vladimír Boža, Broňa Brejová, and Tomáš Vinař. DeepNano: Deep recurrent neural networks for base calling in MinION nanopore reads. *PloS One*, 12(6), 2017.

[41] Débora YC Brandt, Vitor RC Aguiar, Bárbara D Bitarello, Kelly Nunes, Jérôme Goudet, and Diogo Meyer. Mapping Bias Overestimates Reference Allele Frequencies at the HLA Genes in the 1000 Genomes Project Phase I Data. *G3: Genes, Genomes, Genetics*, 5(5):931–941, 2015.

[42] Daniel Branton, David W Deamer, Andre Marziali, Hagan Bayley, Steven A Benner, Thomas Butler, Massimiliano Di Ventra, Slaven Garaj, Andrew Hibbs, Xiaohua Huang, Stevan B Jovanovich, Predrag S Krstic, Stuart Lindsay, Xinsheng Sean Ling, Carlos H Mastrangelo, Amit Meller, John S Oliver, Yuriy V Pershin, J Michael Ramsey, Robert Riehn, Gautam V Soni, Vincent Tabard-Cossa, Meni Wanunu, Matthew Wiggin, and Jeffery A Schloss. The Potential and Challenges of Nanopore Sequencing. *Nature Biotechnology*, 2008.

[43] Nick Bray, Inna Dubchak, and Lior Pachter. AVID: A Global Alignment Program. *Genome Research*, 2003.

[44] Michael Brudno, Chuong B Do, Gregory M Cooper, Michael F Kim, Eugene Davydov, NISC Comparative Sequencing Program, Eric D Green, Arend Sidow, and Serafim Batzoglou. LAGAN and Multi-LAGAN: Efficient Tools for Large-Scale Multiple Alignment of Genomic DNA. *Genome Research*, 2003.

[45] Stefan Burkhardt and Juha Kärkkäinen. Better filtering with gapped q-grams. *Fundamenta Informaticae*, 56(1-2):51–70, 2003.

[46] Burrows-Wheeler Aligner. `http://bio-bwa.sourceforge.net`.

[47] Canu - A single molecule sequence assembler for genomes large and small. `https://github.com/marbl/canu`.

[48] Canu tutorial. `http://canu.readthedocs.io/en/latest/tutorial.html`.

[49] Humberto Carrillo and David Lipman. The Multiple Sequence Alignment Problem in Biology. *SIAP*, 1988.

[50] Mark Chaisson, Pavel Pevzner, and Haixu Tang. Fragment Assembly with Short Reads. *Bioinformatics*, 2004.

[51] Mark JP Chaisson, Richard K Wilson, and Evan E Eichler. Genetic Variation and the De Novo Assembly of Human Genomes. *Nature Reviews Genetics*, 2015.

[52] Erika Check Hayden. Technology: The 1,000 Genome. *Nature News*, 2014.

[53] Peng Chen, Chao Wang, Xi Li, and Xuehai Zhou. Accelerating the Next Generation Long Read Mapping with the FPGA-Based System. *TCBB*, 2014.

[54] Lynda Chin, Jannik N Andersen, and P Andrew Futreal. Cancer Genomics: From Discovery Science to Personalized Medicine. *Nature Medicine*, 2011.

[55] Justin Chu, Hamid Mohamadi, René L Warren, Chen Yang, and Inanc Birol. Innovations and challenges in detecting long read overlaps: an evaluation of the state-of-the-art. *Bioinformatics*, 33(8):1261–1270, 2016.

[56] James Clarke, Hai-Chen Wu, Lakmal Jayasinghe, Alpesh Patel, Stuart Reid, and Hagan Bayley. Continuous base identification for single-molecule nanopore dna sequencing. *Nature Nanotechnology*, 4(4):265, 2009.

[57] Peter JA Cock, Christopher J Fields, Naohisa Goto, et al. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research*, 38(6):1767–1771, 2009.

[58] Phillip EC Compeau, Pavel A Pevzner, and Glenn Tesler. How to apply de Bruijn graphs to genome assembly. *Nature Biotechnology*, 29(11):987–991, 2011.

[59] Computonics GmbH. Computomics is co-organizing the pangenome browser group at COVID-19 Biohackathon: Towards browsing the global genetic variation of SARS-CoV-2. `https://computomics.com/news-reader/covid19hackathon.html`, 2020.

[60] Ian Curtis. The Intel Skylake-X Review: Core i9 7900X, i7 7820X and i7 7800X Tested: Die Size Estimates and Arrangements. `https://www.anandtech.com/show/11550/the-intel-skylakex-review-core-i9-7900x-i7-7820x-and-i7-7800x-tested/6`.

[61] Darlan da Silva Candido, Ingra Morales Claro, Jaqueline Goes de Jesus, William Marciel de Souza, Filipe Romero Rebello Moreira, Simon Dellicour, Thomas A. Mellan, Louis du Plessis, Rafael Henrique Moraes Pereira, Flavia Cristina da Silva Sales, Erika Regina Manuli, Julien Theze, Luis Almeida, Mariane Talon de Menezes, Carolina Moreira Voloch, Marcilio Jorge Fumagalli, Thais de Moura Coletti, Camila Alves Maia Silva, Mariana Severo Ramundo, Mariene Ribeiro Amorim, Henrique Hoeltgebaum, Swapnil Mishra, Mandev Gill, Luiz Max Carvalho, Lewis Fletcher Buss, Carlos Augusto Prete Jr., Jordan Ashworth, Helder Nakaya, Pedro da Silva Peixoto, Oliver J Brady, Samuel M. Nicholls, Amilcar Tanuri, Atila Duque Rossi, Carlos Kaue Vieira Braga, Alexandra Lehmkuhl Gerber, Ana Paula Guimaraes, Nelson Gaburo Jr., Cecilia Salete Alencar, Alessandro Clayton de Souza Ferreira, Cristiano Xavier Lima, Jose Eduardo Levi, Celso Granato, Giula Magalhaes Ferreira,

Ronaldo da Silva Francisco Jr., Fabiana Granja, Marcia Teixeira Garcia, Maria Luiza Moretti, Mauricio Wesley Perroud Jr., Terezinha Marta Pereira Pinto Castineiras, Carolina Dos Santos Lazari, Sarah C Hill, Andreza Aruska de Souza Santos, Camila Lopes Simeoni, Julia Forato, Andrei Carvalho Sposito, Angelica Zaninelli Schreiber, Magnun Nueldo Nunes Santos, Camila Zolini Sa, Renan Pedra Souza, Luciana Cunha Resende Moreira, Mauro Martins Teixeira, Josy Hubner, Patricia Asfora Falabella Leme, Rennan Garcias Moreira, Mauricio Lacerda Nogueira, CADDE-Genomic-Network, Neil Ferguson, Silvia Figueiredo Costa, Jose Luiz Proenca-Modena, Ana Tereza Vasconcelos, Samir Bhatt, Philippe Lemey, Chieh-Hsi Wu, Andrew Rambaut, Nick J Loman, Renato Santana Aguiar, Oliver G Pybus, Ester Cerdeira Sabino, and Nuno Rodrigues Faria. Evolution and Epidemic Spread of SARS-CoV-2 in Brazil. *Science*, 2020.

[62] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. FPGP: Graph processing framework on FPGA a case study of breadth-first search. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016.

[63] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[64] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. GraphH: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):640–653, 2018.

[65] Jeff Daily. Parasail: SIMD C Library for Global, Semi-Global, and Local Pairwise Sequence Alignments. *BMC Bioinformatics*, 2016.

[66] Darwin: A co-processor for long read alignment. `https://github.com/yatisht/darwin`.

[67] Matei David, Lewis Jonathan Dursi, Delia Yao, Paul C Boutros, and Jared T Simpson. Nanocall: an open source basecaller for Oxford Nanopore sequencing data. *Bioinformatics*, 33(1):49–55, 2016.

[68] W Rhett Davis, John Wilson, Stephen Mick, Jian Xu, Hao Hua, Christopher Mineo, Ambarish M Sule, Michael Steer, and Paul D Franzon. Demystifying 3D ICs: The Pros and Cons of Going Vertical. *IEEE Design & Test of Computers*, 2005.

[69] Carlos Victor de Lannoy, Dick de Ridder, and Judith Risse. A sequencer coming of age: de novo genome assembly using MinION reads. *bioRxiv*, 2017.

[70] David Deamer, Mark Akeson, and Daniel Branton. Three decades of nanopore sequencing. *Nature Biotechnology*, 34(5):518–525, 2016.

[71] DeepNano: alternative basecaller for MinION reads. https://bitbucket.org/vboza/deepnano.

[72] Jacob F Degner, John C Marioni, Athma A Pai, Joseph K Pickrell, Everlyne Nkadori, Yoav Gilad, and Jonathan K Pritchard. Effect of Read-Mapping Biases on Detecting Allele-Specific Expression from RNA-Sequencing Data. *Bioinformatics*, 25(24):3207–3212, 2009.

[73] Arthur L Delcher, Simon Kasif, Robert D Fleischmann, Jeremy Peterson, Owen White, and Steven L Salzberg. Alignment of Whole Genomes. *Nucleic Acids Research*, 1999.

[74] Michael Delorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomas E Uribe, F Thomas Jr, and Andre DeHon. GraphStep: A system architecture for sparse-graph algorithms. In *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.

[75] Colin N Dewey. Whole-Genome Alignment. In *Evolutionary Genomics*. 2019.

[76] Alexander Dilthey, Charles Cox, Zamin Iqbal, Matthew R Nelson, and Gil McVean. Improved Genome Inference in the MHC Using a Population Reference Graph. *Nature genetics*, 47(6):682–688, 2015.

[77] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. The Mondrian data engine. In *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[78] Sean R Eddy. Hidden markov models. *Current Opinion in Structural Biology*, 6(3):361–365, 1996.

[79] Robert C Edgar. MUSCLE: Multiple Sequence Alignment with High Accuracy and High Throughput. *Nucleic Acids Research*, 2004.

[80] Robert C Edgar and Serafim Batzoglou. Multiple Sequence Alignment. *COSB*, 2006.

[81] Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, Rebecca L Stamm, and Dean M Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, (5):12–19, 1997.

[82] Hans Ellegren. Genome Sequencing and Population Genomics in Non-Model Organisms. *Trends in Ecology & Evolution*, 2014.

[83] Adam C English, Stephen Richards, Yi Han, Min Wang, Vanesa Vee, Jiaxin Qu, Xiang Qin, Donna M Muzny, Jeffrey G Reid, Kim C Worley, and Richard A. Gibbs. Mind the Gap: Upgrading Genomes with Pacific Biosciences RS Long-Read Sequencing Technology. *PloS One*, 2012.

[84] Nuno Rodrigues Faria, Ester C Sabino, Marcio RT Nunes, Luiz Carlos Junior Alcantara, Nicholas J Loman, and Oliver G Pybus. Mobile Real-Time Surveillance of Zika Virus in Brazil. *Genome Medicine*, 2016.

[85] Anthony S Fauci, H Clifford Lane, and Robert R Redfield. Covid-19–Navigating the Uncharted. *New England Journal of Medicine*, 2020.

[86] Xia Fei, Zou Dan, Lu Lina, Man Xin, and Zhang Chunlei. FPGASW: Accelerating large-scale Smith-Waterman sequence alignment application with backtracking on FPGA linear systolic array. *Interdisciplinary Sciences: Computational Life Sciences*, 10(1):176–188, 2018.

[87] Ivan Fernandez, Ricardo Quislant, Eladio Gutiérrez, Oscar Plata, Christina Giannoula, Mohammed Alser, Juan Gómez-Luna, and Onur Mutlu. NATSA: A Near-Data Processing Accelerator for Time Series Analysis. In *ICCD*, 2020.

[88] Lars Feuk, Andrew R Carson, and Stephen W Scherer. Structural Variation in the Human Genome. *Nature Reviews Genetics*, 2006.

[89] James W Fickett. Fast Optimal Alignment. *Nucleic Acids Research*, 1984.

[90] Can Firtina and Can Alkan. On Genomic Repeats and Reproducibility. *Bioinformatics*, 2016.

[91] Can Firtina, Jeremie S Kim, Mohammed Alser, Damla Senol Cali, A Ercument Cicek, Can Alkan, and Onur Mutlu. Apollo: A Sequencing-Technology-Independent, Scalable, and Accurate Assembly Polishing Algorithm. *Bioinformatics*, 36(12):3669–3679, 2020.

[92] Mauricio Flores, Gustavo Glusman, Kristin Brogaard, Nathan D Price, and Leroy Hood. P4 Medicine: How Systems Medicine Will Transform the Healthcare Sector and Society. *Personalized Medicine*, 2013.

[93] G David Forney. The Viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[94] Edward J Fox, Kate S Reid-Bayliss, Mary J Emond, and Lawrence A Loeb. Accuracy of Next Generation Sequencing Platforms. *Next Generation, Sequencing & Applications*, 2014.

[95] Daichi Fujiki, Aran Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. GenAx: A genome sequencing accelerator. In *ISCA*, 2018.

[96] Daichi Fujiki, Shunhao Wu, Nathan Ozog, Kush Goliya, David Blaauw, Satish Narayanasamy, and Reetuparna Das. SeedEx: A Genome Sequencing Accelerator for Optimal Alignments in Subminimal Space. In *MICRO*, 2020.

[97] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3D memory. In *ACM SIGARCH Computer Architecture News*, 2017.

[98] Yan Gao, Yongzhuang Liu, Yanmei Ma, Bo Liu, Yadong Wang, and Yi Xing. abPOA: an SIMD-based C library for fast partial order alignment using adaptive band. *bioRxiv*, 2020.

[99] Erik Garrison, Jouni Sirén, Adam M Novak, Glenn Hickey, Jordan M Eizenga, Eric T Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F Lin, et al. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 2018.

[100] GATK. Reference Genome Components. `https://gatk.broadinstitute.org/hc/en-us/articles/360041155232-Reference-Genome-Components`.

[101] Evangelos Georganas, Aydin Buluc, Jarrod Chapman, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. merAligner: A fully parallel sequence aligner. In *IEEE International Parallel and Distributed Processing Symposium*, 2015.

[102] Abdullah Gharaibeh, Tahsin Reza, Elizeu Santos-Neto, Lauro Beltrao Costa, Scott Sallinen, and Matei Ripeanu. Efficient large-scale graph processing on hybrid CPU and GPU systems. *arXiv preprint arXiv:1312.3018*, 2013.

[103] Saugata Ghose, Amirali Boroumand, Jeremie S Kim, Juan Gómez-Luna, and Onur Mutlu. Processing-in-Memory: A Workload-Driven Perspective. In *IBM JRD*, 2019.

[104] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. Demystifying Complex Workload-DRAM Interactions: An Experimental Study. In *SIGMETRICS*, 2019.

[105] Geoffrey S Ginsburg and Huntington F Willard. Genomic and Personalized Medicine: Foundations and Applications. *Translational Research*, 2009.

[106] Travis C Glenn. Field guide to next-generation DNA sequencers. *Molecular Ecology Resources*, 2011.

[107] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture. In *arXiv*, 2021.

[108] Sara Goodwin, John D McPherson, and W Richard McCombie. Coming of age: ten years of next-generation sequencing technologies. *Nature Reviews Genetics*, 17(6):333, 2016.

[109] Osamu Gotoh. An Improved Algorithm for Matching Biological Sequences. *Journal of Molecular Biology*, 1982.

[110] Osamu Gotoh. Alignment of Three Biological Sequences with an Efficient Traceback Procedure. *Journal of Theoretical Biology*, 1986.

[111] Amit Goyal, Hyuk Jung Kwon, Kichan Lee, Reena Garg, Seon Young Yun, Yoon Hee Kim, Sunghoon Lee, and Min Seob Lee. Ultra-Fast Next Generation Human Genome Sequencing Data Processing Using DRAGEN Bio-IT Processor for Precision Medicine. *Open Journal of Genetics*, 2017.

[112] GraphMap - A highly sensitive and accurate mapper for long, error-prone reads. `https://github.com/isovic/graphmap`.

[113] Alexander L Greninger, Samia N Naccache, Scot Federman, Guixia Yu, Placide Mbala, Vanessa Bres, Doug Stryke, Jerome Bouquet, Sneha Somasekar, Jeffrey M Linnen, Roger Dodd, Prime Mulembakani, Bradley S. Schneide, Jean-Jacques Muyembe-Tamfum, Susan L. Stramer, and Charles Y. Chiu. Rapid Metagenomic Identification of Viral Pathogens in Clinical Samples by Real-Time Nanopore Sequencing Analysis. *Genome Medicine*, 2015.

[114] Wei-jie Guan, Zheng-yi Ni, Yu Hu, Wen-hua Liang, Chun-quan Ou, Jian-xing He, Lei Liu, Hong Shan, Chun-liang Lei, David SC Hui, et al. Clinical characteristics of coronavirus disease 2019 in China. *New England Journal of Medicine*, 2020.

[115] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao, and Hai Jin. A survey on graph processing accelerators: Challenges and opportunities. *Journal of Computer Science and Technology*, 34(2):339–371, 2019.

[116] Torsten Günther and Carl Nettelblad. The Presence and Impact of Reference Bias on Population Genomic Studies of Prehistoric Human Populations. *PLoS Genetics*, 15(7):e1008302, 2019.

[117] Saransh Gupta, Mohsen Imani, Behnam Khaleghi, Venkatesh Kumar, and Tajana Rosing. RAPID: A ReRAM Processing in-Memory Architecture for DNA Sequence Alignment. In *ISLPED*, 2019.

[118] Nastaran Hajinazar, Geraldo F Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM. In *ASPLOS*, 2021.

[119] Tae Jun Ham, David Bruns-Smith, Brendan Sweeney, Yejin Lee, Seong Hoon Seo, U Gyeong Song, Young H Oh, Krste Asanovic, Jae W Lee, and Lisa Wu Wills. Genesis: A Hardware Acceleration Framework for Genomic Data Analysis. In *ISCA*, 2020.

[120] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[121] Waqar Haque, Alex Aravind, and Bharath Reddy. Pairwise Sequence Alignment Algorithms: A Survey. In *ISTA*, 2009.

[122] Jennifer Harcourt, Azaibi Tamin, Xiaoyan Lu, Shifaq Kamili, Senthil Kumar Sakthivel, Lijuan Wang, Janna Murray, Krista Queen, Brian Lynch, Brett Whitaker, Brian Lynch, Rashi Gautam, Craig Schindewolf, Kumari G. Lokugamage, Dionna Scharton, Jessica A. Plante, Divya Mirchandani, Steven G. Widen, Krishna Narayanan, Shinji Makino, Thomas G. Ksiazek, Kenneth S. Plante, Scott C. Weaver, Stephen Lindstrom, Suxiang Tong, Vineet D. Menachery, and Natalie J. Thornburg. Isolation and Characterization of SARS-CoV-2 from the First US COVID-19 Patient. bioRxiv 2020.03.02.972935, 2020.

[123] Desmond G Higgins and Paul M Sharp. CLUSTAL: A Package for Performing Multiple Sequence Alignment on a Microcomputer. *Gene*, 1988.

[124] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *ACM SIGARCH Computer Architecture News*, 1992.

[125] Michael Höhl, Stefan Kurtz, and Enno Ohlebusch. Efficient Multiple Genome Alignment. *Bioinformatics*, 2002.

[126] Manuel Holtgrewe. Mason–a read simulator for second generation sequencing data. *Technical Report FU Berlin*, 2010.

[127] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *ISCA*, 2016.

[128] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation. In *ICCD*, 2016.

[129] Chaolin Huang, Yeming Wang, Xingwang Li, Lili Ren, Jianping Zhao, Yi Hu, Li Zhang, Guohui Fan, Jiuyang Xu, Xiaoying Gu, et al. Clinical features of patients infected with 2019 novel coronavirus in Wuhan, China. *The Lancet*, 2020.

[130] Wenqin Huangfu, Shuangchen Li, Xing Hu, and Yuan Xie. RADAR: A 3D-ReRAM based DNA Alignment Accelerator Architecture. In *DAC*, 2018.

[131] Wenqin Huangfu, Xueqi Li, Shuangchen Li, Xing Hu, Peng Gu, and Yuan Xie. MEDAL: Scalable DIMM Based Near Data Processing Accelerator for DNA Seeding Algorithm. In *MICRO*, 2019.

[132] Wenqin Huangfu, Krishna T Malladi, Shuangchen Li, Peng Gu, and Yuan Xie. NEST: DIMM based Near-Data-Processing Accelerator for K-mer Counting. In *ICCAD*, 2020.

[133] Hybrid Memory Cube Consortium. HMC Specification 2.1, 2014.

[134] Illumina. Illumina DRAGEN Bio-IT Platform. https://www.illumina.com/products/by-type/informatics-products/dragen-bio-it-platform.html.

[135] Illumina, Inc. MiSeq System. https://www.illumina.com/systems/sequencing-platforms/miseq.html.

[136] Illumina, Inc. NextSeq 2000 System. https://www.illumina.com/systems/sequencing-platforms/nextseq-1000-2000.html.

[137] Illumina, Inc. NovaSeq 6000 System. https://www.illumina.com/systems/sequencing-platforms/novaseq.html.

[138] Genome in a Bottle Consortium. Genome in a Bottle Release. https://ftp-trace.ncbi.nlm.nih.gov/giab/ftp/release/.

[139] Broad Institute. BAM File Format. https://software.broadinstitute.org/software/igv/BAM.

[140] Broad Institute. FASTA File Format. https://software.broadinstitute.org/software/igv/FASTA.

[141] Broad Institute. SAM File Format. https://software.broadinstitute.org/software/igv/SAM.

[142] Intel. A New FPGA Architecture and Leading-Edge FinFET Process Technology Promise to Meet Next-Generation System Requirements. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01220-hyperflex-architecture-fpga-socs.pdf.

[143] Intel. Intel performance counter monitor. https://software.intel.com/en-us/articles/intel-performance-counter-monitor.

[144] Intel. Intel Quartus Prime Software Suite. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html.

[145] Intel. Intel Stratix 10 MX (DRAM System-in-Package) Device Overview. https://www.intel.com/content/www/us/en/programmable/documentation/tqq1499375882772.html.

[146] Intel. Intel® Xeon® Gold 6126 Processor (19.25M Cache, 2.60 GHz) Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/120483/intel-xeon-gold-6126-processor-19-25m-cache-2-60-ghz.html.

[147] Camilla LC Ip, Matthew Loose, John R Tyson, Mariateresa de Cesare, Bonnie L Brown, Miten Jain, Richard M Leggett, David A Eccles, Vadim Zalunin, John M Urban, et al. MinION analysis and reference consortium: Phase 1 data release and analysis. *F1000Research*, 2015.

[148] Marten Jäger, Max Schubach, Tomasz Zemojtel, Knut Reinert, Deanna M Church, and Peter N Robinson. Alternate-Locus Aware Variant Calling in Whole Genome Sequencing. *Genome Medicine*, 2016.

[149] Chirag Jain, Sanchit Misra, Haowen Zhang, Alexander Dilthey, and Srinivas Aluru. Accelerating Sequence Alignment to Graphs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 451–461. IEEE, 2019.

[150] Chirag Jain, Arang Rhie, Haowen Zhang, Claudia Chu, Brian P Walenz, Sergey Koren, and Adam M Phillippy. Weighted minimizer sampling improves long read mapping. *Bioinformatics*, 2020.

[151] Miten Jain, Sergey Koren, Karen H Miga, Josh Quick, Arthur C Rand, Thomas A Sasani, John R Tyson, Andrew D Beggs, Alexander T Dilthey, Ian T Fiddes, et al. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature Biotechnology*, 2018.

[152] Miten Jain, John R Tyson, Matthew Loose, Camilla LC Ip, David A Eccles, Justin O'Grady, Sunir Malla, Richard M Leggett, Ola Wallerman, Hans J Jansen, Vadim Zulunin, Ewan Birney, Bonnie L Brown, Terrance P Snutch, Hugh E Olsen, and MinION Analysis Reference Consortium. MinION analysis and reference consortium: Phase 2 data release and analysis of R9. 0 chemistry. *F1000Research*, 2017.

[153] Phillip James, David Stoddart, Eoghan D Harrington, John Beaulaurier, Lynn Ly, Stuart Reid, Daniel J Turner, and Sissel Juul. LamPORE: Rapid, Accurate and Highly Scalable Molecular Screening for SARS-CoV-2 Infection, Based on Nanopore Sequencing. medRxiv 2020.08.07.20161737, 2020.

[154] JEDEC. JESD235: High Bandwidth Memory (HBM) DRAM, 2013.

[155] Xianyang Jiang, Xinchun Liu, Lin Xu, Peiheng Zhang, and Ninghui Sun. A Reconfigurable Accelerator for Smith–Waterman Algorithm. *TCAS-II*, 2007.

[156] Arthur B Kahn. Topological sorting of large networks. *CACM*, 1962.

[157] Roman Kaplan, Leonid Yavits, and Ran Ginosar. RASSA: Resistive Pre-Alignment Accelerator for Approximate DNA Long Read Mapping. *IEEE Micro*, 2018.

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

[159] John J Kasianowicz, Eric Brandin, Daniel Branton, and David W Deamer. Characterization of individual polynucleotide molecules using a membrane channel. *Proceedings of the National Academy of Sciences*, 93(24):13770–13773, 1996.

[160] W James Kent. BLAT—The BLAST-Like Alignment Tool. *Genome Research*, 2002.

[161] Soroosh Khoram, Jialiang Zhang, Maxwell Strange, and Jing Li. Accelerating graph analytics by co-optimizing storage and access on an FPGA-HMC platform. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018.

[162] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, 2014.

[163] Daehwan Kim, Joseph M Paggi, Chanhee Park, Christopher Bennett, and Steven L Salzberg. Graph-based Genome Alignment and Genotyping with HISAT2 and HISAT-genotype. *Nature Biotechnology*, 37(8):907–915, 2019.

[164] Jeremie S Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-memory Technologies. *BMC Genomics*, 19(2):23–40, 2018.

[165] Jeremie S Kim, Can Firtina, Damla Senol Cali, Mohammed Alser, Nastaran Hajinazar, Can Alkan, and Onur Mutlu. AirLift: A Fast and Comprehensive Technique for Translating Alignments between Reference Genomes. *arXiv preprint arXiv:1912.08735*, 2019.

[166] Yeseong Kim, Mohsen Imani, Niema Moshiri, and Tajana Rosing. GenieHD: Efficient DNA Pattern Matching Accelerator using Hyperdimensional Computing. In *DATE*, 2020.

[167] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer architecture letters*, 15(1):45–49, 2016.

[168] Sergey Koren, Gregory P Harhay, Timothy PL Smith, James L Bono, Dayna M Harhay, Scott D Mcvey, Diana Radune, Nicholas H Bergman, and Adam M Phillippy. Reducing assembly complexity of microbial genomes with single-molecule sequencing. *Genome Biology*, 14(9):R101, 2013.

[169] Sergey Koren, Brian P Walenz, Konstantin Berlin, Jason R Miller, Nicholas H Bergman, and Adam M Phillippy. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Research*, 27(5):722–736, 2017.

[170] Hsiang Tsung Kung. Why Systolic Architectures? *IEEE Computer*, 1982.

[171] Hsiang Tsung Kung and Charles E. Leiserson. Systolic Arrays (for VLSI). In *Sparse Matrix Proceedings*, 1978.

[172] Stefan Kurtz, Adam Phillippy, Arthur L Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L Salzberg. Versatile and Open Software for Comparing Large Genomes. *Genome Biology*, 2004.

[173] Ann Franchesca Laguna, Hasindu Gamaarachchi, Xunzhao Yin, Michael Niemier, Sri Parameswaran, and X Sharon Hu. Seed-and-Vote based In-Memory Accelerator for DNA Read Mapping. In *ICCAD*, 2020.

[174] Ben Langmead and Steven L Salzberg. Fast Gapped-Read Alignment with Bowtie 2. *Nature Methods*, 2012.

[175] Dominique Lavenier, Jean-Francois Roy, and David Furodet. DNA Mapping using Processor-in-Memory Architecture. In *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2016.

[176] Thomas Laver, J Harrison, PA O'Neill, Karen Moore, Audrey Farbos, Konrad Paszkiewicz, and David J Studholme. Assessing the performance of the Oxford Nanopore Technologies MinION. *Biomolecular Detection and Quantification*, 3:1–8, 2015.

[177] Christopher Lee, Catherine Grasso, and Mark F Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 2002.

[178] Donghyuk Lee, Saugata Ghose, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost. *TACO*, 2016.

[179] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, 1966.

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

[181] Heng Li. Minimap and Miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, 2016.

[182] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

[183] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The sequence alignment/map format and SAMtools. *Bioinformatics*, 2009.

[184] Hongjian Li, Bing Ni, Man-Hon Wong, and Kwong-Sak Leung. A Fast CUDA Implementation of Agrep Algorithm for Approximate Nucleotide Sequence Matching. In *SASP*, 2011.

[185] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: An Improved Ultrafast Tool for Short Read Alignment. *Bioinformatics*, 2009.

[186] Zhenyu Li, Yanxiang Chen, Desheng Mu, Jianying Yuan, Yujian Shi, Hao Zhang, Jun Gan, Nan Li, Xuesong Hu, Binghang Liu, et al. Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. *Briefings in Functional Genomics*, 11(1):25–37, 2012.

[187] Hsin-Nan Lin and Wen-Lian Hsu. GSAlign: An Efficient Sequence Alignment Tool for Intra-Species Genomes. *BMC Genomics*, 2020.

[188] David J Lipman, Stephen F Altschul, and John D Kececioglu. A Tool for Multiple Sequence Alignment. *PNAS*, 1989.

[189] David J Lipman and William R Pearson. Rapid and Sensitive Protein Similarity Searches. *Science*, 1985.

[190] Yongchao Liu and Bertil Schmidt. GSWABE: Faster GPU-accelerated sequence alignment with optimal alignment retrieval for short DNA sequences. *Concurrency Computation*, 2015.

[191] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 2013.

[192] Yuansheng Liu, Leo Yu Zhang, and Jinyan Li. Fast detection of maximal exact matches via fixed sampling of query k-mers and bloom filtering of index k-mers. *Bioinformatics*, 35(22):4560–4567, 2019.

[193] Glennis A Logsdon, Mitchell R Vollger, and Evan E Eichler. Long-Read Human Genome Sequencing and Its Applications. *Nature Reviews Genetics*, 2020.

[194] Nicholas J Loman. Nanopore R9 rapid run data release. `http://lab.loman.net/2016/07/30/nanopore-r9-data-release/`.

[195] Nicholas J Loman. Thar she blows! Ultra long read method for nanopore sequencing. `http://lab.loman.net/2017/03/09/ultrareads-for-nanopore/`.

[196] Nicholas J Loman, Joshua Quick, and Jared T Simpson. A complete bacterial genome assembled de novo using only nanopore sequencing data. *Nature Methods*, 12(8):733–735, 2015.

[197] Qian Lou, Sarath Janga, and Lei Jiang. Helix: Algorithm/Architecture Co-Design for Accelerating Nanopore Genome Base-Calling. In *PACT*, 2020.

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

[199] Mohammed-Amin Madoui, Stefan Engelen, Corinne Cruaud, et al. Genome assembly using Nanopore-guided long and error-free DNA reads. *BMC Genomics*, 16(1), 2015.

[200] Alberto Magi, Roberto Semeraro, Alessandra Mingrino, Betti Giusti, and Romina D'aurizio. Nanopore sequencing data analysis: state of the art, applications and challenges. *Briefings in Bioinformatics*, 19(6):1256–1272, 2017.

[201] William Magro, Paul Petersen, and Sanjiv Shah. Hyper-threading technology: impact on compute-intensive workloads. *Intel Technology Journal*, 6(1), 2002.

[202] Tuomo Mantere, Simone Kersten, and Alexander Hoischen. Long-Read Sequencing Emerging in Medical Genetics. *Frontiers in Genetics*, 2019.

[203] Guillaume Marçais, Arthur L Delcher, Adam M Phillippy, Rachel Coston, Steven L Salzberg, and Aleksey Zimin. MUMmer4: A Fast and Versatile Genome Alignment System. *PLoS Computational Biology*, 2018.

[204] Debbie Marr, Frank Binns, D Hill, Glenn Hinton, D Koufaty, et al. Hyper-threading technology in the netburst® microarchitecture. In *Hot Chips*, 2002.

[205] Vivien Marx. Nanopores: a sequencer in your backpack. *Nature Methods*, 12(11), 2015.

[206] Metrichor, Oxford Nanopore Technologies. `https://nanoporetech.com/products/metrichor`.

[207] Webb Miller and Eugene W Myers. Sequence Comparison with Concave Weighting Functions. *Bulletin of Mathematical Biology*, 1988.

[208] Miniasm - Ultrafast de novo assembly for long noisy reads. `https://github.com/lh3/miniasm`.

[209] Minimap. `https://github.com/lh3/minimap`.

[210] MinION, Oxford Nanopore Technologies. `https://nanoporetech.com/products/minion`.

[211] MUMmer 3.x. `https://github.com/garviz/MUMmer`.

[212] Onur Mutlu. Accelerating Genome Analysis: A Primer on an Ongoing Journey. *Keynote Talk at AACBB*, 2019.

[213] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. Processing Data Where It Makes Sense: Enabling In-Memory Computation. *MICPRO*, 2019.

[214] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A Modern Primer on Processing in Memory. In *Emerging Computing: From Devices to Systems-Looking Beyond Moore and Von Neumann. Springer*, 2021.

[215] Eugene W Myers and Webb Miller. Optimal Alignments in Linear Space. *Bioinformatics*, 1988.

[216] Gene Myers. A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming. *Journal of the ACM*, 1999.

[217] Anirban Nag, CN Ramachandra, Rajeev Balasubramonian, Ryan Stutsman, Edouard Giacomin, Hari Kambalasubramanyam, and Pierre-Emmanuel Gaillardon. GenCache: Leveraging In-Cache Operators for Efficient Sequence Alignment. In *MICRO*, 2019.

[218] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[219] Kazuma Nakano, Akino Shiroma, Makiko Shimoji, Hinako Tamotsu, Noriko Ashimine, Shun Ohki, Misuzu Shinzato, Maiko Minami, Tetsuhiro Nakanishi, Kuniko Teruya, Kazuhito Satou, and Takashi Hirano. Advantages of Genome Sequencing by Long-Read Sequencer using SMRT Technology in Medical Area. *Human Cell*, 2017.

[220] Nanocall: An Oxford Nanopore Basecaller. `https://github.com/mateidavid/nanocall`.

[221] Nanonet, Oxford Nanopore Technologies. `https://github.com/nanoporetech/nanonet`.

[222] Nanopolish. `https://github.com/jts/nanopolish`.

[223] First DNA Sequencing in Space a Game Changer. `https://www.nasa.gov/mission_pages/station/research/news/dna_sequencing`.

[224] Gonzalo Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys (CSUR)*, 2001.

[225] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 1970.

[226] Cédric Notredame. Recent Progress in Multiple Sequence Alignment: A Survey. *Pharmacogenomics*, 2002.

[227] Cédric Notredame, Desmond G Higgins, and Jaap Heringa. T-Coffee: A Novel Method for Fast and Accurate Multiple Sequence Alignment. *JMB*, 2000.

[228] Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V Bzikadze, Alla Mikheenko, Mitchell R Vollger, Nicolas Altemose, Lev Uralsky, Ariel Gershman, et al. The complete sequence of a human genome. *bioRxiv*, 2021.

[229] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C Hoe, José F Martínez, and Carlos Guestrin. GraphGen: An FPGA framework for vertex-centric graph computation. In *IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014.

[230] NVIDIA. NVIDIA Clara Parabricks. https://developer.nvidia.com/clara-parabricks.

[231] NVIDIA. NVIDIA Tesla V100 GPU Architecture. *White Paper*, 2017.

[232] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture. *White Paper*, 2020.

[233] NVIDIA Corp. NVIDIA TITAN V. https://www.nvidia.com/en-us/titan/titan-v/.

[234] NVIDIA Corp. nvprof. https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview.

[235] Geraldo Francisco Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks. In *arXiv*, 2021.

[236] Yukiteru Ono, Kiyoshi Asai, and Michiaki Hamada. PBSIM: PacBio reads simulator–toward accurate genome assembly. *Bioinformatics*, 29(1):119–121, 2012.

[237] Yukiteru Ono, Kiyoshi Asai, and Michiaki Hamada. PBSIM2: a simulator for long-read sequencers with a novel generative model of quality scores. *Bioinformatics*, 2021.

[238] Oxford Nanopore Technologies. https://nanoporetech.com.

[239] Samtools Organisation. Sequence Alignment/Map (SAM) Format Specification. http://samtools.github.io/hts-specs/SAMv1.pdf.

[240] Samtools Organisation. The Variant Call Format (VCF). https://samtools.github.io/hts-specs/VCFv4.2.pdf.

[241] Oxford Nanopore Technologies Ltd. GridION. https://nanoporetech.com/products/gridion.

[242] Oxford Nanopore Technologies Ltd. PromethION. https://nanoporetech.com/products/promethion.

[243] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. Energy efficient architecture for graph analytics accelerators. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[244] Pacific Biosciences (PacBio). https://www.pacb.com.

[245] Pacific Biosciences of California, Inc. Sequel Systems. https://www.pacb.com/products-and-services/sequel-system.

[246] Benedict Paten, Dent Earl, Ngan Nguyen, Mark Diekhans, Daniel Zerbino, and David Haussler. Cactus: Algorithms for Genome Multiple Sequence Alignment. *Genome Research*, 2011.

[247] Benedict Paten, Adam M Novak, Jordan M Eizenga, and Erik Garrison. Genome Graphs and the Evolution of Genome Inference. *Genome Research*, 27(5):665–676, 2017.

[248] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, and Chita R Das. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *PACT*, 2016.

[249] Barak A Pearlmutter. Learning state space trajectories in recurrent neural networks. *Learning*, 1(2), 2008.

[250] William R Pearson and David J Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.

[251] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[252] Mihai Pop. Genome assembly reborn: recent computational challenges. *Briefings in Bioinformatics*, 10(4):354–366, 2009.

[253] Ryan Poplin, Pi-Chuan Chang, David Alexander, Scott Schwartz, Thomas Colthurst, Alexander Ku, Dan Newburger, Jojo Dijamco, Nam Nguyen, Pegah T Afshar, et al. A universal SNP and small-indel variant caller using deep neural networks. *Nature Biotechnology*, 2018.

[254] Javier Prado-Martinez, Peter H. Sudmant, Jeffrey M. Kidd, Heng Li, Joanna L. Kelley, Belen Lorente-Galdos, Krishna R. Veeramah, August E. Woerner, Timothy D. O'Connor, Gabriel Santpere, Alexander Cagan, Christoph Theunert, Ferran Casals, Hafid Laayouni, Kasper Munch, Asger Hobolth, Anders E. Halager, Maika Malig, Jessica Hernandez-Rodriguez, Irene Hernando-Herraez, Kay Prüfer, Marc Pybus, Laurel Johnstone, Michael Lachmann, Can Alkan, Dorina Twigg, Natalia Petit, Carl Baker, Fereydoun Hormozdiari, Marcos Fernandez-Callejo, Marc Dabad, Michael L. Wilson, Laurie Stevison, Cristina Camprubí, Tiago Carvalho, Aurora Ruiz-Herrera, Laura Vives, Marta Mele, Teresa Abello, Ivanela Kondova, Ronald E. Bontrop, Anne Pusey, Felix Lankester, John A. Kiyang, Richard A. Bergl, Elizabeth Lonsdorf, Simon Myers, Mario Ventura, Pascal Gagneux, David Comas, Hans Siegismund, Julie Blanc, Lidia Agueda-Calpena, Marta Gut, Lucinda Fulton, Sarah A. Tishkoff, James C. Mullikin, Richard K. Wilson, Ivo G. Gut, Mary Katherine Gonder, Oliver A. Ryder, Beatrice H. Hahn, Arcadi Navarro, Joshua M. Akey, Jaume Bertranpetit, David Reich, Thomas Mailund, Mikkel H. Schierup, Christina Hvilsom, Aida M. Andrés,

Jeffrey D. Wall, Carlos D. Bustamante, Michael F. Hammer, Evan E. Eichler, and Tomas Marques-Bonet. Great ape genetic diversity and population history. *Nature*, 2013.

[255] Ana Prohaska, Fernando Racimo, Andrew J Schork, Martin Sikora, Aaron J Stern, Melissa Ilardo, Morten Erik Allentoft, Lasse Folkersen, Alfonso Buil, J Víctor Moreno-Mayar, Thorfinn Korneliussen, Daniel Geschwind, Andrés Ingason, Thomas Werge, Rasmus Nielsen, and Eske Willerslev. Human disease variation in the light of population genomics. *Cell*, 2019.

[256] Michael A Quail, Miriam Smith, Paul Coupland, Thomas D Otto, Simon R Harris, Thomas R Connor, Anna Bertoni, Harold P Swerdlow, and Yong Gu. A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers. *BMC Genomics*, 13(1):341, 2012.

[257] Joshua Quick, Nicholas J Loman, Sophie Duraffour, et al. Real-time, portable genome sequencing for Ebola surveillance. *Nature*, 530(7589), 2016.

[258] Joshua Quick, Aaron R Quinlan, and Nicholas J Loman. A reference bacterial genome dataset generated on the MinION portable single-molecule nanopore sequencer. *Gigascience*, 3(1), 2014.

[259] Update: New R9 nanopore for faster, more accurate sequencing, and new ten minute preparation kit. `https://nanoporetech.com/about-us/news/update-new-r9-nanopore-faster-more-accurate-sequencing-and-new-ten-minute-preparation`.

[260] Racon - Consensus module for raw de novo DNA assembly of long uncorrected reads. `https://github.com/isovic/racon`.

[261] Goran Rakocevic, Vladimir Semenyuk, Wan-Ping Lee, James Spencer, John Browning, Ivan J Johnson, Vladan Arsenijevic, Jelena Nadj, Kaushik Ghose, Maria C Suciu, et al. Fast and Accurate Genomic Analyses Using Genome Graphs. *Nature Genetics*, 51(2):354–362, 2019.

[262] Mikko Rautiainen and Tobias Marschall. GraphAligner: rapid and versatile sequence-to-graph alignment. *Genome Biology*, 2020.

[263] Jason A Reuter, Damek V Spacek, and Michael P Snyder. High-throughput sequencing technologies. *Molecular Cell*, 58(4):586–597, 2015.

[264] Anthony Rhoads and Kin Fai Au. PacBio Sequencing and Its Applications. *Genomics, Proteomics & Bioinformatics*, 2015.

[265] Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 2004.

[266] Richard J Roberts, Mauricio O Carneiro, and Michael C Schatz. The Advantages of SMRT Sequencing. *Genome Biology*, 2013.

[267] Enzo Rucci, Carlos Garcia, Guillermo Botella, Armando De Giusti, Marcelo Naiouf, and Manuel Prieto-Matias. SWIFOLD: Smith–Waterman Implementation on FPGA with OpenCL for Long DNA Sequences. *BMC Systems Biology*, 2018.

[268] SAFARI Research Group. GenASM — GitHub Repository. https://github.com/CMU-SAFARI/GenASM.

[269] SAFARI Research Group. Shouji — GitHub Repository. https://github.com/CMU-SAFARI/Shouji.

[270] Leena Salmela and Eric Rivals. Lordec: Accurate and efficient long read error correction. *Bioinformatics*, 30(24):3506–3514, 2014.

[271] David Sankoff. Minimal Mutation Trees of Sequences. *SIAP*, 1975.

[272] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, 2003.

[273] Valerie A Schneider, Tina Graves-Lindsay, Kerstin Howe, Nathan Bouk, Hsiu-Chuan Chen, Paul A Kitts, Terence D Murphy, Kim D Pruitt, Françoise Thibaud-Nissen, Derek Albracht, et al. Evaluation of GRCh38 and De Novo Haploid Genome Assemblies Demonstrates the Enduring Quality of the Reference Assembly. *Genome Research*, 27(5):849–864, 2017.

[274] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[275] Scott Schwartz, W James Kent, Arian Smit, Zheng Zhang, Robert Baertsch, Ross C Hardison, David Haussler, and Webb Miller. Human–Mouse Alignments with BLASTZ. *Genome Research*, 2003.

[276] Clive Brown Technical Update: GridION X5 - The Sequel. https://nanoporetech.com/resource-centre/videos/gridion-x5-sequel.

[277] Scrappie, Oxford Nanopore Technologies. https://github.com/nanoporetech/scrappie.

[278] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. GraphReduce: processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[279] Damla Senol Cali, Gurpreet S Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, et al. GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis. In *MICRO*, 2020.

[280] Damla Senol Cali, Jeremie S Kim, Saugata Ghose, Can Alkan, and Onur Mutlu. Nanopore Sequencing Technology and Tools for Genome Assembly: Computational Analysis of the Current State, Bottlenecks and Future Directions. *Briefings in Bioinformatics*, 2018.

[281] Jay Shendure, Shankar Balasubramanian, George M Church, Walter Gilbert, Jane Rogers, Jeffery A Schloss, and Robert H Waterston. DNA sequencing at 40: Past, present and future. *Nature*, 2017.

[282] Premkishore Shivakumar and Norman P Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. 2001.

[283] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. Abyss: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

[284] Gagandeep Singh, Mohammed Alser, Damla Senol Cali, Dionysios Diamantopoulos, Juan Gómez-Luna, Henk Corporaal, and Onur Mutlu. FPGA-based Near-Memory Acceleration of Modern Data-Intensive Applications. *IEEE Micro*, 2021.

[285] Gagandeep Singh, Dionysios Diamantopoulos, Juan Gómez-Luna, Christoph Hagleitner, Sander Stuijk, Henk Corporaal, and Onur Mutlu. NERO: Accelerating Weather Prediction using Near-Memory Reconfigurable Fabric. *arXiv preprint arXiv:2107.08716*, 2021.

[286] Gagandeep Singh, Juan Gómez-Luna, Giovanni Mariani, Geraldo F Oliveira, Stefano Corda, Sander Stuijk, Onur Mutlu, and Henk Corporaal. NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning. In *DAC*, 2019.

[287] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 1981.

[288] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. GraphR: Accelerating graph processing using ReRAM. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[289] Martin Šošić and Mile Šikić. Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 2017.

[290] Ivan Sović, Mile Šikić, Andreas Wilm, Shannon Nicole Fenlon, Swaine Chen, and Niranjan Nagarajan. Fast and sensitive mapping of nanopore sequencing reads with GraphMap. *Nature Communications*, 7:11307, 2016.

[291] Arun Subramaniyan, Jack Wadden, Kush Goliya, Nathan Ozog, Xiao Wu, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Accelerating Maximal-Exact-Match Seeding with Enumerated Radix Trees. *BioRxiv*, 2020.

[292] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 2014.

[293] Haotian Teng, Minh Duc Cao, Michael B Hall, Tania Duarte, Sheng Wang, and Lachlan JM Coin. Chiron: translating nanopore raw signal directly into nucleotide sequence using deep learning. *GigaScience*, 7(5):giy037, 2018.

[294] Julie D Thompson, Desmond G Higgins, and Toby J Gibson. CLUSTAL W: Improving the Sensitivity of Progressive Multiple Sequence Alignment Through Sequence Weighting, Position-Specific Gap Penalties and Weight Matrix Choice. *Nucleic Acids Research*, 1994.

[295] Cole Trapnell and Steven L Salzberg. How to Map Billions of Short Reads onto Genomes. *Nature Biotechnology*, 2009.

[296] Todd J Treangen and Steven L Salzberg. Repetitive DNA and next-generation sequencing: computational challenges and solutions. *Nature Reviews Genetics*, 13(1), 2011.

[297] Nathan Tuck and Dean M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[298] Dean M Tullsen, Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, and Rebecca L Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. *ACM SIGARCH Computer Architecture News*, 24(2):191–202, 1996.

[299] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[300] Y. Turakhia, S. D. Goenka, G. Bejerano, and W. J. Dally. Darwin-WGA: A Co-processor Provides Increased Sensitivity in Whole Genome Alignments with High Speedup. In *HPCA*, 2019.

[301] Yatish Turakhia, Gill Bejerano, and William J Dally. Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly. In *ASPLOS*, 2018.

[302] Esko Ukkonen. Algorithms for Approximate String Matching. *Information and Control*, 1985.

[303] Erwin L Van Dijk, Hélène Auger, Yan Jaszczyszyn, et al. Ten years of next-generation sequencing technology. *Trends in Genetics*, 30(9):418–426, 2014.

[304] Erwin L van Dijk, Yan Jaszczyszyn, Delphine Naquin, and Claude Thermes. The Third Revolution in Sequencing Technology. *Trends in Genetics*, 2018.

[305] Robert Vaser, Ivan Sović, Niranjan Nagarajan, and Mile Šikić. Fast and accurate de novo genome assembly from long uncorrected reads. *Genome Research*, 27(5):737–746, 2017.

[306] J. Craig Venter, Mark D. Adams, Eugene W. Myers, Peter W. Li, Richard J. Mural, Granger G. Sutton, Hamilton O. Smith, Mark Yandell, Cheryl A. Evans, Robert A. Holt, Jeannine D. Gocayne, Peter Amanatides, Richard M. Ballew, Daniel H. Huson, Jennifer Russo Wortman, Qing Zhang, Chinnappa D. Kodira, Xiangqun H. Zheng, Lin Chen, Marian Skupski, Gangadharan Subramanian, Paul D. Thomas, Jinghui Zhang, George L. Gabor Miklos, Catherine Nelson, Samuel Broder, Andrew G. Clark, Joe Nadeau, Victor A. McKusick, Norton Zinder, Arnold J. Levine, Richard J. Roberts, Mel Simon, Carolyn Slayman, Michael Hunkapiller1, Randall Bolanos, Arthur Delcher, Ian Dew, Daniel Fasulo, Michael Flanigan, Liliana Florea, Aaron Halpern, Sridhar Hannenhalli, Saul Kravitz, Samuel Levy, Clark Mobarry, Knut Reinert, Karin Remington, Jane Abu-Threideh, Ellen Beasley, Kendra Biddick, Vivien Bonazzi, Rhonda Brandon, Michele Cargill, Ishwar Chandramouliswaran, Rosane Charlab, Kabir Chaturvedi, Zuoming Deng, Valentina Di Francesco, Patrick Dunn, Karen Eilbeck, Carlos Evangelista, Andrei E. Gabrielian, Weiniu Gan, Wangmao Ge, Fangcheng Gong, Zhiping Gu, Ping Guan, Thomas J. Heiman, Maureen E. Higgins, Rui-Ru Ji, Zhaoxi Ke, Karen A. Ketchum, Zhongwu Lai, Yiding Lei, Zhenya Li, Jiayin Li, Yong Liang, Xiaoying Lin, Fu Lu, Gennady V. Merkulov, Natalia Milshina, Helen M. Moore, Ashwinikumar K Naik, Vaibhav A. Narayan, Beena Neelam, Deborah Nusskern, Douglas B. Rusch, Steven Salzberg, Wei Shao, Bixiong Shue, Jingtao Sun, Zhen Yuan Wang, Aihui Wang, Xin Wang, Jian Wang, Ming-Hui Wei, Ron Wides, Chunlin Xiao, Chunhua Yan, Alison Yao, Jane Ye, Ming Zhan, Weiqing Zhang, Hongyu Zhang, Qi Zhao, Liansheng Zheng, Fei Zhong, Wenyan Zhong, Shiaoping C. Zhu, Shaying Zhao, Dennis Gilbert, Suzanna Baumhueter, Gene Spier, Christine Carter, Anibal Cravchik, Trevor Woodage, Feroze Ali, Huijin An, Aderonke Awe, Danita Baldwin, Holly Baden, Mary Barnstead, Ian Barrow, Karen Beeson, Dana Busam, Amy Carver, Angela Center, Ming Lai Cheng, Liz Curry, Steve Danaher, Lionel Davenport, Raymond Desilets, Susanne Dietz, Kristina Dodson, Lisa Doup, Steven Ferriera, Neha Garg, Andres Gluecksmann, Brit Hart, Jason Haynes, Charles Haynes, Cheryl Heiner, Suzanne Hladun, Damon Hostin, Jarrett Houck, Timothy Howland, Chinyere Ibegwam, Jeffery Johnson, Francis Kalush, Lesley Kline, Shashi Koduru, Amy Love, Felecia Mann, David May, Steven McCawley, Tina McIntosh, Ivy McMullen, Mee Moy, Linda Moy, Brian Murphy, Keith Nelson, Cynthia Pfannkoch, Eric Pratts, Vinita Puri, Hina Qureshi, Matthew Reardon, Robert Rodriguez, Yu-Hui Rogers, Deanna Romblad, Bob Ruhfel, Richard Scott, Cynthia Sitter, Michelle Smallwood, Erin Stewart, Renee Strong, Ellen Suh, Reginald Thomas, Ni Ni Tint, Sukyee Tse, Claire Vech, Gary Wang, Jeremy Wetter, Sherita Williams, Monica Williams, Sandra Windsor, Emily Winn-Deen, Keriellen Wolfe, Jayshree Zaveri, Karena Zaveri, Josep F. Abril, Roderic Guigó, Michael J. Campbell, Kimmen V. Sjolander,

Brian Karlak, Anish Kejariwal, Huaiyu Mi, Betty Lazareva, Thomas Hatton, Apurva Narechania, Karen Diemer, Anushya Muruganujan, Nan Guo, Shinji Sato, Vineet Bafna, Sorin Istrail, Ross Lippert, Russell Schwartz, Brian Walenz, Shibu Yooseph, David Allen, Anand Basu, James Baxendale, Louis Blick, Marcelo Caminha, John Carnes-Stine, Parris Caulk, Yen-Hui Chiang, My Coyne, Carl Dahlke, Anne Deslattes Mays, Maria Dombroski, Michael Donnelly, Dale Ely, Shiva Esparham, Carl Fosler, Harold Gire, Stephen Glanowski, Kenneth Glasser, Anna Glodek, Mark Gorokhov, Ken Graham, Barry Gropman, Michael Harris, Jeremy Heil, Scott Henderson, Jeffrey Hoover, Donald Jennings, Catherine Jordan, James Jordan, John Kasha, Leonid Kagan, Cheryl Kraft, Alexander Levitsky, Mark Lewis, Xiangjun Liu, John Lopez, Daniel Ma, William Majoros, Joe McDaniel, Sean Murphy, Matthew Newman, Trung Nguyen, Ngoc Nguyen, Marc Nodell, Sue Pan, Jim Peck, Marshall Peterson, William Rowe, Robert Sanders, John Scott, Michael Simpson, Thomas Smith, Arlan Sprague, Timothy Stockwell, Russell Turner, Eli Venter, Mei Wang, Meiyuan Wen, David Wu, Mitchell Wu, Ashley Xia, Ali Zandieh, and Xiaohong Zhu. The Sequence of the Human Genome. *Science*, 2001.

[307] Jing Wang, Nicole Elizabeth Moore, Yi-Mo Deng, David A Eccles, and Richard J Hall. MinION Nanopore Sequencing of an Influenza Genome. *Frontiers in Microbiology*, 2015.

[308] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *ACM SIGPLAN Notices*, 2016.

[309] Yu Wang, James C Hoe, and Eriko Nurvitadhi. Processor assisted worklist scheduling for FPGA accelerated graph processing on a shared-memory platform. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[310] Michael S Waterman. Efficient Sequence Alignment Algorithms. *Journal of Theoretical Biology*, 1984.

[311] Michael S Waterman, Temple F Smith, and William A Beyer. Some Biological Sequence Metrics. *Advances in Mathematics*, 1976.

[312] Jason L Weirather, Mariateresa de Cesare, Yunhao Wang, Paolo Piazza, Vittorio Sebastiano, Xiu-Jie Wang, David Buck, and Kin Fai Au. Comprehensive comparison of pacific biosciences and oxford nanopore technologies and their applications to transcriptome analysis. *F1000Research*, 2017.

[313] Aaron M Wenger, Paul Peluso, William J Rowell, Pi-Chuan Chang, Richard J Hall, Gregory T Concepcion, Jana Ebler, Arkarachai Fungtammasan, Alexey Kolesnikov, Nathan D Olson, Armin Töpfer, Michael Alonge, Medhat Mahmoud, Yufeng Qian, Chen-Shan Chin, Adam M. Phillippy, Michael C. Schatz, Gene Myers, Mark A. DePristo, Jue Ruan, Tobias Marschall, Fritz J. Sedlazeck, Justin M. Zook, Heng Li, Sergey Koren, Andrew Carroll, David R. Rank, and Michael W. Hunkapiller.

Accurate Circular Consensus Long-Read Sequencing Improves Variant Detection and Assembly of a Human Genome. *Nature Biotechnology*, 2019.

[314] Ryan R Wick, Louise M Judd, and Kathryn E Holt. Comparison of Oxford Nanopore basecalling tools. `https://github.com/rrwick/Basecalling-comparison`.

[315] Steven JE Wilton and Norman P Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, 1996.

[316] Fan Wu, Su Zhao, Bin Yu, Yan-Mei Chen, Wen Wang, Zhi-Gang Song, Yi Hu, Zhao-Wu Tao, Jun-Hua Tian, Yuan-Yuan Pei, Ming-Li Yuan, Yu-Ling Zhang, Fa-Hui Dai, Yi Liu, Qi-Min Wang, Jiao-Jiao Zheng, Lin Xu, Edward C. Holmes, and Yong-Zhen Zhang. A New Coronavirus Associated with Human Respiratory Disease in China. *Nature*, 2020.

[317] Sun Wu and Udi Manber. Fast Text Searching Allowing Errors. *Communications of the ACM*, 1992.

[318] Chuan-Le Xiao, Ying Chen, Shang-Qian Xie, et al. MECAT: fast mapping, error correction, and de novo assembly for single-molecule sequencing reads. *Nature Methods*, 14(11):1072, 2017.

[319] Xilinx, Inc. Virtex® UltraScale+™. `https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html`.

[320] Xilinx, Inc. Vivado Design Suite. `https://www.xilinx.com/products/design-tools/vivado.html`.

[321] Hongyi Xin, John Greth, John Emmons, Gennady Pekhimenko, Carl Kingsford, Can Alkan, and Onur Mutlu. Shifted Hamming distance: a fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping. *Bioinformatics*, 31(10):1553–1560, 2015.

[322] Hongyi Xin, Donghyuk Lee, Farhad Hormozdiari, Samihan Yedkar, Onur Mutlu, and Can Alkan. Accelerating read mapping with FastHASH. *BMC Genomics*, 14(1):S13, 2013.

[323] Hongyi Xin, Sunny Nahar, Richard Zhu, John Emmons, Gennady Pekhimenko, Carl Kingsford, Can Alkan, and Onur Mutlu. Optimal Seed Solver: Optimizing Seed Selection in Read Mapping. *Bioinformatics*, 2016.

[324] Wayne Yamamoto and Mario Nemirovsky. Increasing superscalar performance through multistreaming. In *Proceedings of the Working Conference on Parallel Architectures and Compilation Techniques*, 1995.

[325] Yaran Yang, Bingbing Xie, and Jiangwei Yan. Application of Next-Generation Sequencing Technology in Forensic Science. *Genomics, Proteomics & Bioinformatics*, 2014.

[326] Daniel R Zerbino and Ewan Birney. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.

[327] Jialiang Zhang, Soroosh Khoram, and Jing Li. Boosting the performance of FPGA-based graph processor using hybrid memory cube: A case for breadth first search. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[328] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian. GraphP: Reducing Communication for PIM-based Graph Processing with Efficient Data Partition. In *HPCA*, 2018.

[329] Tao Zhang, Jingjie Zhang, Wei Shu, Min-You Wu, and Xiaoyao Liang. Efficient graph computation on hybrid CPU and GPU systems. *The Journal of Supercomputing*, 71(4):1563–1586, 2015.

[330] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2013.

[331] Jinhong Zhou, Shaoli Liu, Qi Guo, Xuda Zhou, Tian Zhi, Daofu Liu, Chao Wang, Xuehai Zhou, Yunji Chen, and Tianshi Chen. Tunao: A high-performance and energy-efficient reconfigurable accelerator for graph processing. In *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017.

[332] Shijie Zhou, Charalampos Chelmis, and Viktor K Prasanna. High-throughput and energy-efficient graph processing on FPGA. In *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[333] Farzaneh Zokaee, Hamid R Zarandi, and Lei Jiang. AligneR: A Process-in-Memory Architecture for Short Read Alignment in ReRAMs. *IEEE CAL*, 2018.

[334] Farzaneh Zokaee, Mingzhe Zhang, and Lei Jiang. FindeR: Accelerating FM-Index-Based Exact Pattern Matching in Genomic Sequences Through ReRAM Technology. In *PACT*, 2019.

[335] Quan Zou, Qinghua Hu, Maozu Guo, and Guohua Wang. HAlign: Fast Multiple Similar DNA/RNA Sequence Alignment Based on the Centre Star Strategy. *Bioinformatics*, 2015.