2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)

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

Damla Senol Cali[†⋈] Gurpreet S. Kalsi[⋈] Zülal Bingöl[▽] Can Firtina[◇] Lavanya Subramanian[‡] Jeremie S. Kim[◇†]
Rachata Ausavarungnirun[⊙] Mohammed Alser[◇] Juan Gomez-Luna[◇] Amirali Boroumand[†] Anant Nori[⋈]
Allison Scibisz[†] Sreenivas Subramoney[⋈] Can Alkan[▽] Saugata Ghose[⋆†] Onur Mutlu[◇†▽]

[†] *Carnegie Mellon University* [⋈] *Processor Architecture Research Lab, Intel Labs* [▽] *Bilkent University* [◇] *ETH Zürich*
[‡] *Facebook* [⊙] *King Mongkut's University of Technology North Bangkok* [⋆] *University of Illinois at Urbana–Champaign*

Genome sequence analysis has enabled significant advancements in medical and scientific areas such as personalized medicine, outbreak tracing, and the understanding of evolution. To perform genome sequencing, devices extract small random fragments of an organism's DNA sequence (known as reads*). The first step of genome sequence analysis is a computational process known as* read mapping*. In read mapping, each fragment is matched to its potential location in the reference genome with the goal of identifying the original location of each read in the genome. Unfortunately, rapid genome sequencing 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. A major contributor to this bottleneck is* approximate string matching *(ASM), which is used at multiple points during the mapping process. ASM enables read mapping to account for sequencing errors and genetic variations in the reads.*

We propose GenASM, the first 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 conclude that GenASM is a flexible, high-performance, and low-power framework, and we briefly discuss four other use cases that can benefit from GenASM.*

1. Introduction

Genome sequencing, which determines the DNA sequence of an organism, plays a pivotal role in enabling many medical and scientific advancements in personalized medicine [6, 20, 34, 53, 59], evolutionary theory [46, 139, 140], and forensics [17, 25, 179]. Modern genome sequencing machines [77–79, 132–135, 152] can rapidly generate massive amounts of genomics data at low cost [8, 118, 153], 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 *read mapping*, which takes each read, aligns it to one or more possible locations within the reference genome, and finds the matches and differences (i.e., *distance*) between the read and the reference genome segment at that location [6, 177]. Read mapping is the first key step in genome sequence analysis.

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 [30, 158]) are generated using short-read sequencing (SRS) technologies [144, 164], 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 [166]), short reads incur a number of reproducibility (e.g., non-deterministic mapping) and computational challenges [7, 10, 12, 52, 118, 159, 176–178]. *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 [26, 35, 40, 82, 83, 89, 97, 112, 113, 116, 143, 152] and Pacific Biosciences' (PacBio) single-molecule real-time (SMRT) sequencing [18, 47, 114, 123, 145, 146, 165, 171] 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 [165]. Second, LRS devices perform real-time sequencing, and can enable concurrent sequencing and analysis [111, 142, 146]. Third, ONT's pocket-sized device (MinION [133]) 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 [37, 48, 64, 68, 85, 142, 167, 173]). Unfortunately, LRS devices are much more error-prone in sequencing (with a typical error rate of 10–15% [19, 83, 165, 170]) compared to SRS devices (typically 0.1% [60, 61, 141]), which leads to new computational challenges [152].

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 [121, 125, 154, 163, 169, 174]. 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 expen-

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00081

sive dynamic programming based algorithm [100, 126, 154] that scales quadratically in both execution time and required storage. This ASM algorithm has been shown to be the major bottleneck in read mapping [8, 10, 55, 66, 75, 122, 162]. Unfortunately, as sequencing technologies advance, the growth in the rate that sequencing devices generate reads is far outpacing the corresponding growth in computational power [8, 32], 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) [27,28,41,42,70,95,102,106,115,151,160] and multiple sequence alignment (MSA) [29,45,69,98,107,127,128,136,150], 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 [22, 33, 49, 65, 87, 88, 147, 162], we base GenASM upon the *Bitap* algorithm [21, 174]. 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.

To use Bitap for GenASM, we make two key algorithmic modifications that allow us to overcome key limitations that prevent the original Bitap algorithm from being efficient for genome sequence analysis (we discuss these limitations in Section 2.3). First, to improve Bitap's applicability to different sequencing technologies and its performance, we (1) modify the algorithm to support long reads (in addition to already supported short reads), and (2) eliminate loop-carried data dependencies so that we can parallelize a single string matching operation. Second, we develop a novel Bitap-compatible algorithm for *traceback*, a method that utilizes information collected during ASM about the different types of errors to identify the optimal alignment of reads. The original Bitap algorithm is not capable of performing traceback.

In GenASM, we *co-design* our modified Bitap algorithm and our new Bitap-compatible *traceback* algorithm with an area- and power-efficient hardware accelerator, which consists of two components: (1) *GenASM-DC*, which provides hardware support to efficiently execute our modified Bitap algorithm to generate bitvectors (each of which represents one of the four possible cases: match, insertion, deletion, or substitution) and perform distance calculation (DC) (which calculates the minimum number of errors between the read and the reference segment); and (2) *GenASM-TB*, which provides hardware support to efficiently execute our novel traceback (TB) algorithm to find the optimal alignment of a read, using the bitvectors generated by GenASM-DC. 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.

Use Cases. GenASM is an efficient framework for accelerating genome sequence analysis that has multiple possible use cases. In this paper, we describe and rigorously evaluate three use cases of GenASM. First, we show that GenASM can effectively accelerate the read alignment step of read mapping (Section 10.2). Second, we illustrate that GenASM can be employed as the most efficient (to date) pre-alignment filter [9, 10] for short reads (Section 10.3). Third, we demonstrate how GenASM can efficiently find the edit distance (i.e., Levenshtein distance [100]) between two sequences of arbitrary lengths (Section 10.4). In addition, GenASM can be utilized in several other parts of genome sequence analysis as well as in text analysis, which we briefly discuss in Section 11.

Results Summary. We evaluate GenASM for three different use cases of ASM in genome sequence analysis using a combination of the synthesized SystemVerilog model of our hardware accelerators and detailed simulation-based performance modeling. (1) For read alignment, we compare GenASM to state-of-the-art software (Minimap2 [102] and BWA-MEM [101]) and hardware approaches (GACT in Darwin [162] and SillaX in GenAx [55]), and find that GenASM is significantly more efficient in terms of both speed and power consumption. For this use case, we compare GenASM *only* with the read alignment steps of the baseline tools and accelerators. For long reads, GenASM achieves 116× and 648× speedup over 12-thread runs of the alignment steps of Minimap2 and BWA-MEM, respectively, while reducing power consumption by 37× and 34×. Compared to GACT, GenASM provides 6.6× the throughput per unit area and 10.5× the throughput per unit power for long reads. For short reads, GenASM achieves 158× and 111× speedup over 12-thread runs of the alignment steps of Minimap2 and BWA-MEM, respectively, while reducing power consumption by 31× and 33×. Compared to SillaX, GenASM is 1.9× faster at a comparable area and power consumption. (2) For pre-alignment filtering of short reads, we compare GenASM with a state-of-the-art FPGA-based filter, Shouji [9]. GenASM provides 3.7× speedup over Shouji, while reducing power consumption by 1.7×, and also significantly improving the filtering accuracy. (3) For edit distance calculation, we compare GenASM with a state-of-the-art software library, Edlib [155], and FPGA-based accelerator, ASAP [22]. Compared to Edlib, GenASM provides 22–12501× speedup, for varying sequence lengths and similarity values, while reducing power consumption by 548–582×. Compared to ASAP, GenASM provides 9.3–400× speedup, while reducing power consumption by 67×.

This paper makes the following contributions:

- To our knowledge, 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. We open source our software implementations of the GenASM algorithms [148].
- 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.
- 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). 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.

2. Background

2.1. Genome Sequence Analysis Pipeline

A common approach to the first step in 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 1 shows, typical read mapping [6, 96, 101, 102, 105, 177] is a four-step process. First, read mapping starts with *indexing* ⓿, which is an offline preprocessing 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 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 [62, 117, 168]). The optimal alignment is defined using a *CIGAR string* [103], 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.

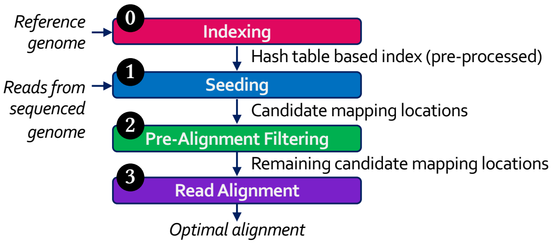

Reference genome
0
Indexing
Hash table based index (pre-processed)
Reads from sequenced genome
1
Seeding
Candidate mapping locations
2
Pre-Alignment Filtering
Remaining candidate mapping locations
3
Read Alignment
Optimal alignment

Figure 1. Four steps of read mapping.

2.2. Approximate String Matching (ASM)

The goal of approximate string matching [125] 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 [18, 54] and/or genetic variations [5, 50]. Reads are prone to sequencing errors, which account for about 0.1% of the length of short reads [60, 61, 141] and 10–15% of the length of long reads [19, 83, 165, 170].

The differences, known as *edits*, can be classified as *substitutions*, *deletions*, or *insertions* in one or both sequences [100]. Figure 2 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.

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

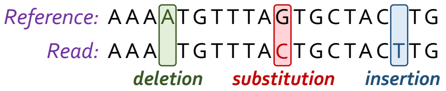


Figure 2. Three types of errors (i.e., edits).

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 [100], Smith-Waterman [154], and Needleman-Wunsch [126], 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.

2.3. Bitap Algorithm

One candidate to replace dynamic programming based algorithms for ASM is the *Bitap* algorithm [21, 174]. 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 3 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 3) 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 3). 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 ($R[d]$; $d = 1 \ldots k$) 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 3) 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 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

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            ▷ check if MSB is 0
21:             startLoc ← i                              ▷ matching location
22:             editDist ← d                       ▷ found minimum edit distance
```

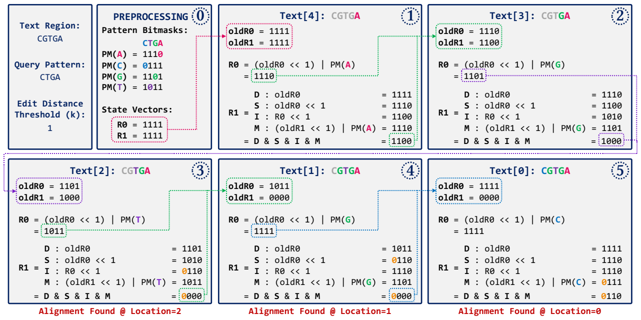


Figure 3. Example for the Bitap algorithm.

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.

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.

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 2.3, 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 [14, 51, 62, 63, 117, 120, 154, 163, 168, 169] 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 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 [104], 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.

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. 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: (1) GenASM-DC (Section 5), which for each read generates the bitvectors and performs the minimum edit distance calculation (DC); and (2) GenASM-TB (Section 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 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). 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.

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 7.

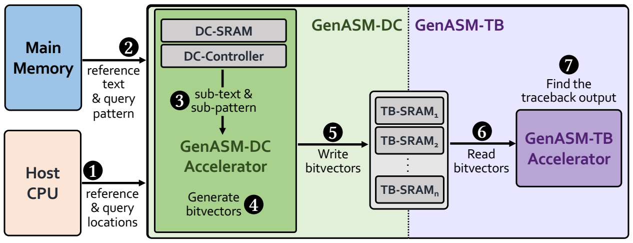


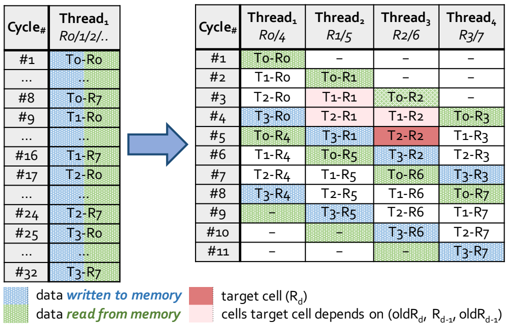
Figure 4. Overview of GenASM.

5. GenASM-DC Algorithm

We modify the baseline Bitap algorithm (Section 2.3) 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 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 3.1), GenASM-DC performs loop unrolling and enables computing non-neighbor (i.e., independent) bitvectors in parallel. Figure 5 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.



Cycle#	Thread₁ Ro/1/2/..
#1	T0-R0
...	...
#8	T0-R7
#9	T1-R0
...	...
#16	T1-R7
#17	T2-R0
...	...
#24	T2-R7
#25	T3-R0
...	...
#32	T3-R7

Cycle#	Thread₁ R0/4	Thread₂ R1/5	Thread₃ R2/6	Thread₄ R3/7
#1	T0-R0	–	–	–
#2	T1-R0	T0-R1	–	–
#3	T2-R0	T1-R1	T0-R2	–
#4	T3-R0	T2-R1	T1-R2	T0-R3
#5	T0-R4	T3-R1	T2-R2	T1-R3
#6	T1-R4	T0-R5	T3-R2	T2-R3
#7	T2-R4	T1-R5	T0-R6	T3-R3
#8	T3-R4	T2-R5	T1-R6	T0-R7
#9	–	T3-R5	T2-R6	T1-R7
#10	–	–	T3-R6	T2-R7
#11	–	–	–	T3-R7

data *written to memory*
data *read from memory*
target cell (R_d)
cells target cell depends on ($oldR_d$, R_{d-1}, $oldR_{d-1}$)

Figure 5. Loop unrolling in GenASM-DC.

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.

6. GenASM-TB Algorithm

After finding the matching location of the text and the edit distance with GenASM-DC, our new traceback [14, 51, 62, 63, 117, 120, 154, 163, 168, 169] 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 6 shows an example for the execution of the algorithm for each of the alignments found in ③–⑤ of Figure 3. In Figure 6, $<x, y, z>$ 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 6) 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 6a, 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>.

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 6b).

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 6c).

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 6a).

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'
14:             status ⟵ 3; add "I" to CIGAR;               ▷ insertion-extend
15:         else if del[textI][curError][patternI]=0 & prev='D'
16:             status ⟵ 4; add "D" to CIGAR;                ▷ deletion-extend
17:         else if match[textI][curError][patternI]=0
18:             status ⟵ 1; add "M" to CIGAR; prev ⟵ "M"             ▷ match
19:         else if subs[textI][curError][patternI]=0
20:             status ⟵ 2; add "S" to CIGAR; prev ⟵ "S"      ▷ substitution
21:         else if ins[textI][curError][patternI]=0
22:             status ⟵ 3; add "I" to CIGAR; prev ⟵ "I" ▷ insertion-open
23:         else if del[textI][curError][patternI]=0
24:             status ⟵ 4; add "D" to CIGAR; prev ⟵ "D"  ▷ deletion-open
25:         if (status > 1)
26:             curError--                                     ▷ S, D, or I
27:         if (status > 0) && (status != 3)
28:             textI++; textConsumed++                         ▷ M, S, or D
29:         if (status > 0) && (status != 4)
30:             patternI--; patternConsumed++                   ▷ M, S, or I
31:     curPattern ⟵ curPattern+patternConsumed
32:     curText ⟵ curText+textConsumed
```

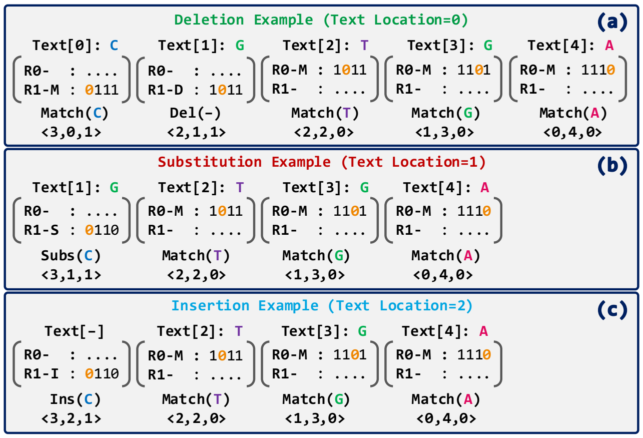


Figure 6. Traceback example with GenASM-TB algorithm.

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 [162]). 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 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 10.2) for non-unit costs for different edits and the affine gap penalty model [14, 62, 117, 168]. 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).

7. GenASM Hardware Design

GenASM-DC Hardware. We implement GenASM-DC as a linear cyclic systolic array [93, 94] 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 [93] and to reduce memory footprint.

A GenASM-DC accelerator consists of a processing block (PB; Figure 7a) 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 7b) 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 7a) or internally for cyclic operations.

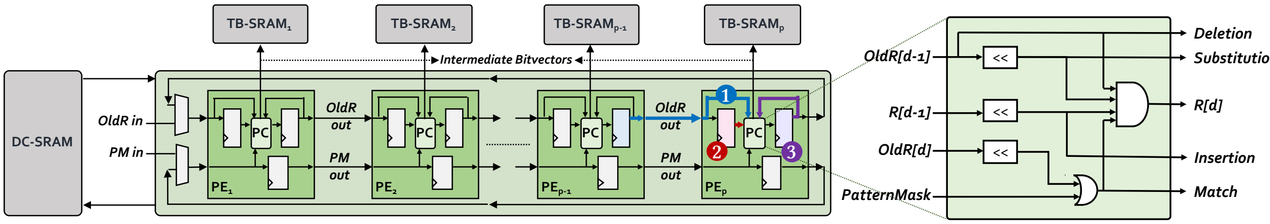
GenASM-DC uses two types of SRAM buffers (Figure 7a): (1) DC-SRAM, which stores the reference text, the pattern bitmasks for the query read, and the intermediate data generated from PEs (i.e., *oldR* values and MSBs required for shifts; Section 5); and (2) TB-SRAM, which stores the intermediate bitvectors 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 5 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 5 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 7), 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.

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 6), we have one 1.5KB TB-SRAM (which fits our 24B/cycle × 64 cycles/window output storage requirement) for each of the 64 PEs. As Figure 8 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

[1] Although we use 10Kbp-long reads in our analysis (Section 9), GenASM does *not* have any limitation on the length of reads as a result of our divide-and-conquer approach (Section 6).



(a) Processing Block (PB), DC-SRAM and TB-SRAMs for each PE

(b) Processing Core (PC)

Figure 7. Hardware design of GenASM-DC.

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 8), 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.

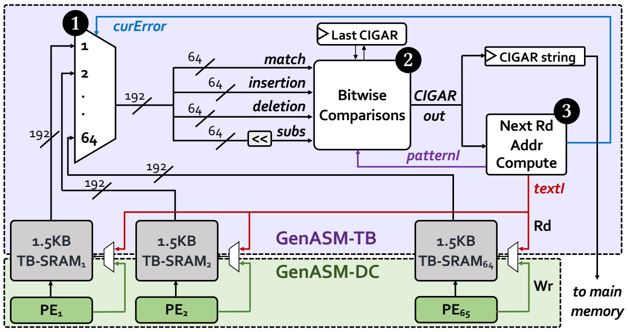


Figure 8. Hardware design of GenASM-TB.

Overall System. We design our system to take advantage of modern 3D-stacked memory systems [58, 92], such as the Hybrid Memory Cube (HMC) [76] or High-Bandwidth Memory (HBM) [86, 99]. 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 [39, 99]. 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 [3, 4, 23, 24, 43, 56, 72, 73, 91, 119, 137].

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 [76], pseudo-channels in HBM [86]) 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 [119]. Both of our hardware accelerators are highly efficient in terms of area and power (Section 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 [76]. 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).

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 11.

Read Alignment of Short and Long Reads. As we explain in Section 2.1, 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, 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 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 [9, 10, 13, 176, 177].

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 [100], 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 [155]. As we explain in Section 2.3, the Bitap algorithm, which is the underlying algorithm of GenASM-DC, is originally designed for edit distance calculation. Thus, GenASM framework can accelerate

[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.

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).

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 [156] 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 [76] and 256GB/s of internal bandwidth [23, 76], and a clock frequency of 1.25GHz [76]. The amount of available area in the logic layer for GenASM is around 3.5–4.4 mm^2 per vault [23, 43]. The power budget of our PIM logic per vault is 312mW [43].

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 [102] and BWA-MEM [101], running on an Intel® Xeon® Gold 6126 CPU [80] 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 [81].

We also compare GenASM with a state-of-the-art GPU-accelerated short read alignment tool, GASAL2 [2]. We run GASAL2 on an Nvidia Titan V GPU [129] with 12GB HBM2 memory [86]. 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 [2]. 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* [130].

We also compare GenASM with two state-of-the-art hardware-based alignment accelerators, GACT of Darwin [162] and SillaX of GenAx [55]. We synthesize and execute the open-source RTL for GACT [161]. We estimate the performance of SillaX using data reported by the original work [55].

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 [9], 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 [9]. For power analysis, we report the total power consumption of Shouji using the power analysis tool in Xilinx Vivado [175], after synthesizing and implementing the open-source FPGA design of Shouji [149].

Edit Distance Calculation Comparisons. We compare GenASM with the state-of-the-art software-based read alignment library, Edlib [155], running on an Intel® Xeon® Gold 6126 CPU [80] operating at 2.60GHz, with 64GB DDR4 memory. Edlib uses the Myers' bitvector algorithm [121] to find the edit distance between two sequences. We use the default global Needleman-Wunsch (NW) [126] mode of Edlib to perform our comparisons. We measure the power consumed by Edlib using Intel's PCM power utility [81].

We also compare GenASM with ASAP [22], 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 [22].

Datasets. For the read alignment use case, we evaluate GenASM using the latest major release of the human genome assembly, GRCh38 [124]. 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 [131] and three sets of short reads (i.e., Illumina datasets) using Mason [71]. 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 [84]. 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 [9] 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 [155] 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.

10. Results

10.1. Area and Power Analysis

Table 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 [80] (which we conservatively estimate to use 10.4 W [80] and 32.2 mm^2 [36] 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 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)**

10.2. Use Case 1: Read Alignment

Software Baselines (CPU). Figure 9 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 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 9 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.

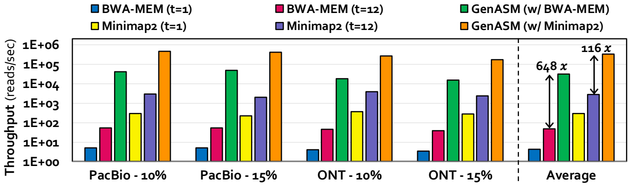


Figure 9. Throughput comparison of GenASM and the alignment steps of BWA-MEM and Minimap2 for long reads.

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

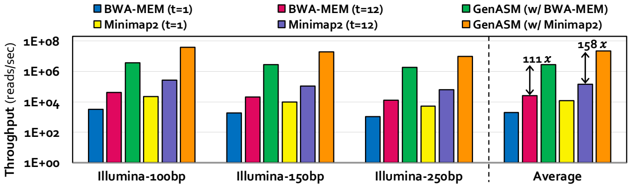


Figure 10. Throughput comparison of GenASM and the alignment steps of BWA-MEM and Minimap2 for short reads.

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 11 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 11 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.

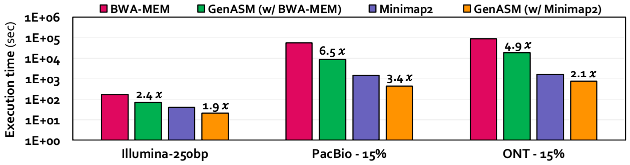


Figure 11. Total execution time of the entire BWA-MEM and Minimap2 pipelines with and without GenASM.

Software Baselines (GPU). We compare GenASM with the state-of-the-art GPU aligner, GASAL2 [2], 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 [162]) and SillaX (from GenAx [55]).

Darwin is a hardware accelerator designed for *long* read alignment [162]. 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 12 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.

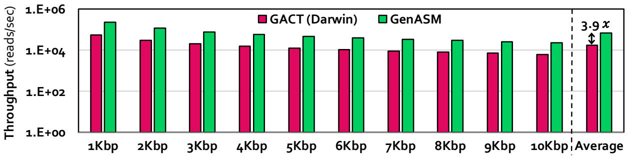


Figure 12. Throughput comparison of GenASM and GACT from Darwin for long reads.

As Figure 13 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.

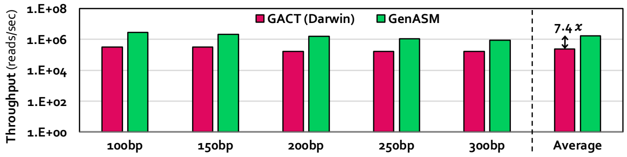


Figure 13. Throughput comparison of GenASM and GACT from Darwin for short reads.

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.

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 [55]. 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 [55] 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 [172] 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.

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 [9], using two datasets provided in [9]. 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×.

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

In pre-alignment filtering for short reads, only GenASM-DC is executed (Section 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 [155] 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 [9]).

The false accept rate [9] 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 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 [9] 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.

10.4. Use Case 3: Edit Distance Calculation

We compare GenASM with the state-of-the-art edit distance calculation library, Edlib [155]. Figure 14 compares the execution time of Edlib (with and without finding the traceback 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 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 14 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.

Although both the GenASM algorithm and Edlib's underlying Myers' algorithm [121] 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.

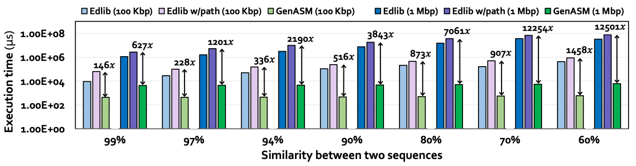


Figure 14. Execution time comparison of GenASM and Edlib for edit distance calculation.

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 [22], 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 [22] 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 [22]. Based on [22], 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.

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

[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.

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.

11. Other Use Cases of GenASM

We have quantitatively evaluated three use cases of approximate string matching for genome sequence analysis (Section 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 [31] 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 [152]. Pairwise read alignment (i.e., read-to-read alignment) is the last step of read-to-read overlap finding [102, 138]. 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 [42, 136] 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 [15, 16, 44, 67, 69, 90, 108, 126, 126, 128, 154, 157, 182].

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 [8].

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 [177] that is implemented for standard CPUs, SHD [176] that uses SIMD-capable CPUs, and GRIM-Filter [91] 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 [10], MAGNET [11], Shouji [9], and SneakySnake [13]. A recent work, GenCache [122], proposes an in-cache accelerator to improve the filtering (i.e., seeding) mechanism of GenAx (for short reads) by using in-cache operations [1] 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 [74], FindeR [181], and AligneR [180], which make use of ReRAM based designs for faster FM-index search, or RAPID [65] and BioSEAL [88], which target dynamic programming acceleration with processing-in-memory. Other read alignment acceleration works include SIMD-capable CPUs [38], multicore CPUs [57, 109], and specialized hardware accelerators such as GPUs (e.g., GSWABE [109], CUDASW++ 3.0 [110]), FPGAs (e.g., FPGASW [49], ASAP [22]), or ASICs (e.g., Darwin [162] and GenAx [55]).

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.

13. Conclusion

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. We hope that GenASM inspires future work in co-designing algorithms and hardware together to create powerful frameworks that accelerate other bioinformatics workloads and emerging applications.

Acknowledgments

Part of this work was completed during Damla Senol Cali's internship at Intel Labs. This work is supported by funding from Intel, the Semiconductor Research Corporation, the National Institutes of Health (NIH), the industrial partners of the SAFARI Research Group, and partly by EMBO Installation Grant 2521 awarded to Can Alkan. We thank the anonymous reviewers of MICRO 2019, ASPLOS 2020, ISCA 2020, and MICRO 2020 for their comments.

References

[1] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, “Compute Caches,” in *HPCA*, 2017.

[2] N. Ahmed, J. Lévy, S. Ren, H. Mushtaq, K. Bertels, and Z. Al-Ars, “GASAL2: A GPU Accelerated Sequence Alignment Library for High-Throughput NGS Data,” *BMC Bioinformatics*, 2019.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing,” in *ISCA*, 2015.

[4] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture,” in *ISCA*, 2015.

[5] C. Alkan, B. P. Coe, and E. E. Eichler, “Genome Structural Variation Discovery and Genotyping,” *Nature Reviews Genetics*, 2011.

[6] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, S. C. Sahinalp, R. A. Gibbs, and E. E. Eichler, “Personalized Copy Number and Segmental Duplication Maps Using Next-Generation Sequencing,” *Nature Genetics*, 2009.

[7] C. Alkan, S. Sajjadian, and E. E. Eichler, “Limitations of Next-Generation Genome Sequence Assembly,” *Nature Methods*, 2011.

[8] M. Alser, Z. Bingöl, D. Senol Cali, J. Kim, S. Ghose, C. Alkan, and O. Mutlu, “Accelerating Genome Analysis: A Primer on an Ongoing Journey,” *IEEE Micro*, 2020.

[9] M. Alser, H. Hassan, A. Kumar, O. Mutlu, and C. Alkan, “Shouji: A Fast and Efficient Pre-Alignment Filter for Sequence Alignment,” *Bioinformatics*, 2019.

[10] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, “GateKeeper: A New Hardware Architecture for Accelerating Pre-Alignment in DNA Short Read Mapping,” *Bioinformatics*, 2017.

[11] M. Alser, O. Mutlu, and C. Alkan, “MAGNET: Understanding and Improving the Accuracy of Genome Pre-Alignment Filtering,” *TIR*, 2017.

[12] M. Alser, J. Rotman, K. Taraszka, H. Shi, P. I. Baykal, H. T. Yang, V. Xue, S. Knyazev, B. D. Singer, B. Balliu, D. Koslicki, P. Skums, A. Zelikovsky, C. Alkan, O. Mutlu, and S. Mangul, “Technology Dictates Algorithms: Recent Developments in Read Alignment,” arXiv:2003.00110 [q-bio.GN], 2020.

[13] M. Alser, T. Shahroodi, J. Gomez-Luna, C. Alkan, and O. Mutlu, “SneakySnake: A Fast and Accurate Universal Genome Pre-Alignment Filter for CPUs, GPUs, and FPGAs,” arXiv:1910.09020 [q-bio.GN], 2019.

[14] S. F. Altschul and B. W. Erickson, “Optimal Sequence Alignment using Affine Gap Costs,” *Bulletin of Mathematical Biology*, 1986.

[15] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, “Basic Local Alignment Search Tool,” *Journal of Molecular Biology*, 1990.

[16] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, “Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs,” *Nucleic Acids Research*, 1997.

[17] M. J. Alvarez-Cubero, M. Saiz, B. Martínez-García, S. M. Sayalero, C. Entrala, J. A. Lorente, and L. J. Martinez-Gonzalez, “Next Generation Sequencing: An Application in Forensic Sciences?” *Annals of Human Biology*, 2017.

[18] S. L. Amarasinghe, S. Su, X. Dong, L. Zappia, M. E. Ritchie, and Q. Gouil, “Opportunities and Challenges in Long-Read Sequencing Data Analysis,” *Genome Biology*, 2020.

[19] S. Ardui, A. Ameur, J. R. Vermeesch, and M. S. Hestand, “Single Molecule Real-Time (SMRT) Sequencing Comes of Age: Applications and Utilities for Medical Diagnostics,” *Nucleic Acids Research*, 2018.

[20] E. A. Ashley, “Towards Precision Medicine,” *Nature Reviews Genetics*, 2016.

[21] R. Baeza-Yates and G. H. Gonnet, “A New Approach to Text Searching,” *CACM*, 1992.

[22] S. S. Banerjee, M. El-Hadedy, J. B. Lim, Z. T. Kalbarczyk, D. Chen, S. S. Lumetta, and R. K. Iyer, “ASAP: Accelerated Short-Read Alignment on Programmable Hardware,” *TC*, 2019.

[23] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, “Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks,” in *ASPLOS*, 2018.

[24] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng, and O. Mutlu, “CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators,” in *ISCA*, 2019.

[25] C. Børsting and N. Morling, “Next Generation Sequencing and Its Applications in Forensic Genetics,” *Forensic Science International: Genetics*, 2015.

[26] D. Branton, D. W. Deamer, A. Marziali, H. Bayley, S. A. Benner, T. Butler, M. D. Ventra, S. Garaj, A. Hibbs, X. Huang, S. B. Jovanovich, P. S. Krstic, S. Lindsay, X. S. Ling, C. H. Mastrangelo, A. Meller *et al.*, “The Potential and Challenges of Nanopore Sequencing,” *Nature Biotechnology*, 2008.

[27] N. Bray, I. Dubchak, and L. Pachter, “AVID: A Global Alignment Program,” *Genome Research*, 2003.

[28] M. Brudno, C. B. Do, G. M. Cooper, M. F. Kim, E. Davydov, NISC Comparative Sequencing Program, E. D. Green, A. Sidow, and S. Batzoglou, “LAGAN and Multi-LAGAN: Efficient Tools for Large-Scale Multiple Alignment of Genomic DNA,” *Genome Research*, 2003.

[29] H. Carrillo and D. Lipman, “The Multiple Sequence Alignment Problem in Biology,” *SIAP*, 1988.

[30] M. Chaisson, P. Pevzner, and H. Tang, “Fragment Assembly with Short Reads,” *Bioinformatics*, 2004.

[31] M. J. Chaisson, R. K. Wilson, and E. E. Eichler, “Genetic Variation and the De Novo Assembly of Human Genomes,” *Nature Reviews Genetics*, 2015.

[32] E. Check Hayden, “Technology: The 1,000 Genome,” *Nature News*, 2014.

[33] P. Chen, C. Wang, X. Li, and X. Zhou, “Accelerating the Next Generation Long Read Mapping with the FPGA-Based System,” *TCBB*, 2014.

[34] L. Chin, J. N. Andersen, and P. A. Futreal, “Cancer Genomics: From Discovery Science to Personalized Medicine,” *Nature Medicine*, 2011.

[35] J. Clarke, H.-C. Wu, L. Jayasinghe, A. Patel, S. Reid, and H. Bayley, “Continuous Base Identification for Single-Molecule Nanopore DNA Sequencing,” *Nature Nanotechnology*, 2009.

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

[37] D. da Silva Candido, I. M. Claro, J. G. de Jesus, W. M. de Souza, F. R. R. Moreira, S. Dellicour, T. A. Mellan, L. du Plessis, R. H. M. Pereira, F. C. da Silva Sales, E. R. Manuli, J. Theze, L. Almeida, M. T. de Menezes, C. M. Voloch, M. J. Fumagalli *et al.*, “Evolution and Epidemic Spread of SARS-CoV-2 in Brazil,” *Science*, 2020.

[38] J. Daily, “Parasail: SIMD C Library for Global, Semi-Global, and Local Pairwise Sequence Alignments,” *BMC Bioinformatics*, 2016.

[39] W. R. Davis, J. Wilson, S. Mick, J. Xu, H. Hua, C. Mineo, A. M. Sule, M. Steer, and P. D. Franzon, “Demystifying 3D ICs: The Pros and Cons of Going Vertical,” *IEEE Design & Test of Computers*, 2005.

[40] D. Deamer, M. Akeson, and D. Branton, “Three Decades of Nanopore Sequencing,” *Nature Biotechnology*, 2016.

[41] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, “Alignment of Whole Genomes,” *Nucleic Acids Research*, 1999.

[42] C. N. Dewey, “Whole-Genome Alignment,” in *Evolutionary Genomics*, 2019.

[43] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, “The Mondrian Data Engine,” in *ISCA*, 2017.

[44] R. C. Edgar, “MUSCLE: Multiple Sequence Alignment with High Accuracy and High Throughput,” *Nucleic Acids Research*, 2004.

[45] R. C. Edgar and S. Batzoglou, “Multiple Sequence Alignment,” *COSB*, 2006.

[46] H. Ellegren, “Genome Sequencing and Population Genomics in Non-Model Organisms,” *Trends in Ecology & Evolution*, 2014.

[47] A. C. English, S. Richards, Y. Han, M. Wang, V. Vee, J. Qu, X. Qin, D. M. Muzny, J. G. Reid, K. C. Worley, and R. A. Gibbs, “Mind the Gap: Upgrading Genomes with Pacific Biosciences RS Long-Read Sequencing Technology,” *PloS One*, 2012.

[48] N. R. Faria, E. C. Sabino, M. R. Nunes, L. C. J. Alcantara, N. J. Loman, and O. G. Pybus, “Mobile Real-Time Surveillance of Zika Virus in Brazil,” *Genome Medicine*, 2016.

[49] X. Fei, Z. Dan, L. Lina, M. Xin, and Z. Chunlei, “FPGASW: Accelerating Large-Scale Smith–Waterman Sequence Alignment Application with Backtracking on FPGA Linear Systolic Array,” *Interdisciplinary Sciences: Computational Life Sciences*, 2018.

[50] L. Feuk, A. R. Carson, and S. W. Scherer, “Structural Variation in the Human Genome,” *Nature Reviews Genetics*, 2006.

[51] J. W. Fickett, “Fast Optimal Alignment,” *Nucleic Acids Research*, 1984.

[52] C. Firtina and C. Alkan, “On Genomic Repeats and Reproducibility,” *Bioinformatics*, 2016.

[53] M. Flores, G. Glusman, K. Brogaard, N. D. Price, and L. Hood, “P4 Medicine: How Systems Medicine Will Transform the Healthcare Sector and Society,” *Personalized Medicine*, 2013.

[54] E. J. Fox, K. S. Reid-Bayliss, M. J. Emond, and L. A. Loeb, “Accuracy of Next Generation Sequencing Platforms,” *Next Generation Sequencing & Applications*, 2014.

[55] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, “GenAx: A Genome Sequencing Accelerator,” in *ISCA*, 2018.

[56] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, “TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory,” in *ASPLOS*, 2017.

[57] E. Georganas, A. Buluc, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, “merAligner: A Fully Parallel Sequence Aligner,” in *IPDPS*, 2015.

[58] S. Ghose, T. Li, N. Hajinazar, D. S. Cali, and O. Mutlu, “Demystifying Complex Workload-DRAM Interactions: An Experimental Study,” in *SIGMETRICS*, 2019.

[59] G. S. Ginsburg and H. F. Willard, “Genomic and Personalized Medicine: Foundations and Applications,” *Translational Research*, 2009.

[60] T. C. Glenn, “Field Guide to Next-Generation DNA Sequencers,” *Molecular Ecology Resources*, 2011.

[61] S. Goodwin, J. D. McPherson, and W. R. McCombie, “Coming of Age: Ten Years of Next-Generation Sequencing Technologies,” *Nature Reviews Genetics*, 2016.

[62] O. Gotoh, “An Improved Algorithm for Matching Biological Sequences,” *Journal of Molecular Biology*, 1982.

[63] O. Gotoh, “Alignment of Three Biological Sequences with an Efficient Traceback Procedure,” *Journal of Theoretical Biology*, 1986.

[64] A. L. Greninger, S. N. Naccache, S. Federman, G. Yu, P. Mbala, V. Bres, D. Stryke, J. Bouquet, S. Somasekar, J. M. Linnen, R. Dodd, P. Mulembakani, B. S. Schneide, J.-J. Muyembe-Tamfum, S. L. Stramer, and C. Y. Chiu, “Rapid Metagenomic Identification of Viral Pathogens in Clinical Samples by Real-Time Nanopore Sequencing Analysis,” *Genome Medicine*, 2015.

[65] S. Gupta, M. Imani, B. Khaleghi, V. Kumar, and T. Rosing, “RAPID: A ReRAM Processing in-Memory Architecture for DNA Sequence Alignment,” in *ISLPED*, 2019.

[66] T. J. Ham, D. Bruns-Smith, B. Sweeney, Y. Lee, S. H. Seo, U. G. Song, Y. H. Oh, K. Asanovic, J. W. Lee, and L. W. Wills, “Genesis: A Hardware Acceleration Framework for Genomic Data Analysis,” in *ISCA*, 2020.

[67] W. Haque, A. Aravind, and B. Reddy, “Pairwise Sequence Alignment Algorithms: A Survey,” in *ISTA*, 2009.

[68] J. Harcourt, A. Tamin, X. Lu, S. Kamili, S. K. Sakthivel, L. Wang, J. Murray, K. Queen, B. Lynch, B. Whitaker, B. Lynch, R. Gautam, C. Schindewolf, K. G. Lokugamage, D. Scharton, J. A. Plante *et al.*, “Isolation and Characterization of SARS-CoV-2 from the First US COVID-19 Patient,” bioRxiv 2020.03.02.972935, 2020.

[69] D. G. Higgins and P. M. Sharp, “CLUSTAL: A Package for Performing Multiple Sequence Alignment on a Microcomputer,” *Gene*, 1988.

[70] M. Höhl, S. Kurtz, and E. Ohlebusch, “Efficient Multiple Genome Alignment,” *Bioinformatics*, 2002.

[71] M. Holtgrewe, “Mason–A Read Simulator for Second Generation Sequencing Data,” Free Univ. of Berlin, Dept. of Mathematics and Computer Sci., Tech. Rep. TR-B-10-06, 2010.

[72] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O’Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, “Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems,” in *ISCA*, 2016.

[73] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, “Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation,” in *ICCD*, 2016.

[74] W. Huangfu, S. Li, X. Hu, and Y. Xie, “RADAR: A 3D-ReRAM based DNA Alignment Accelerator Architecture,” in *DAC*, 2018.

[75] W. Huangfu, X. Li, S. Li, X. Hu, P. Gu, and Y. Xie, “MEDAL: Scalable DIMM Based Near Data Processing Accelerator for DNA Seeding Algorithm,” in *MICRO*, 2019.

[76] Hybrid Memory Cube Consortium, “Hybrid Memory Cube Specification 2.1,” 2015.

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

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

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

[80] Intel Corp., “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

[81] Intel Corp., “Intel® Performance Counter Monitor,” 2017. https://www.intel.com/software/pcm

[82] C. L. Ip, M. Loose, J. R. Tyson, M. de Cesare, B. L. Brown4, M. Jain, R. M. Leggett, D. A. Eccles, V. Zalunin, J. M. Urban, P. Piazza, R. J. Bowden, B. Paten, S. Mwaigwisya, E. M. Batty, J. T. Simpson *et al.*, “MinION Analysis and Reference Consortium: Phase 1 Data Release and Analysis,” *F1000Research*, 2015.

[83] M. Jain, S. Koren, K. H. Miga, J. Quick, A. C. Rand, T. A. Sasani, J. R. Tyson, A. D. Beggs, A. T. Dilthey, I. T. Fiddes, S. Malla, H. Marriott, T. Nieto, J. O’Grady, H. E. Olsen, B. S. Pedersen *et al.*, “Nanopore Sequencing and Assembly of a Human Genome with Ultra-Long Reads,” *Nature Biotechnology*, 2018.

[84] M. Jain, J. R. Tyson, M. Loose, C. L. Ip, D. A. Eccles, J. O’Grady, S. Malla, R. M. Leggett, O. Wallerman, H. J. Jansen, V. Zulunin, E. Birney, B. L. Brown, T. P. Snutch, H. E. Olsen, and MinION Analysis Reference Consortium, “MinION Analysis and Reference Consortium: Phase 2 Data Release and Analysis of R9.0 Chemistry,” *F1000Research*, 2017.

[85] P. James, D. Stoddart, E. D. Harrington, J. Beaulaurier, L. Ly, S. Reid, D. J. Turner, and S. Juul, “LamPORE: Rapid, Accurate and Highly Scalable Molecular Screening for SARS-CoV-2 Infection, Based on Nanopore Sequencing,” medRxiv 2020.08.07.20161737, 2020.

[86] JEDEC Solid State Technology Assn., “JESD235C: High Bandwidth Memory (HBM) DRAM,” January 2020.

[87] X. Jiang, X. Liu, L. Xu, P. Zhang, and N. Sun, “A Reconfigurable Accelerator for Smith–Waterman Algorithm,” *TCAS-II*, 2007.

[88] R. Kaplan, L. Yavits, and R. Ginosar, “BioSEAL: In-Memory Biological Sequence Alignment Accelerator for Large-Scale Genomic Data,” in *PACT*, 2019.

[89] J. J. Kasianowicz, E. Brandin, D. Branton, and D. W. Deamer, “Characterization of Individual Polynucleotide Molecules using a Membrane Channel,” *PNAS*, 1996.

[90] W. J. Kent, “BLAT—The BLAST-Like Alignment Tool,” *Genome Research*, 2002.

[91] J. S. Kim, D. Senol Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, “GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping using Processing-in-Memory Technologies,” *BMC Genomics*, 2018.

[92] Y. Kim, W. Yang, and O. Mutlu, “Ramulator: A Fast and Extensible DRAM Simulator,” *IEEE CAL*, 2016.

[93] H. T. Kung, “Why Systolic Architectures?” *IEEE Computer*, 1982.

[94] H. T. Kung and C. E. Leiserson, “Systolic Arrays (for VLSI),” in *Sparse Matrix Proceedings*, 1978.

[95] S. Kurtz, A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. L. Salzberg, “Versatile and Open Software for Comparing Large Genomes,” *Genome Biology*, 2004.

[96] B. Langmead and S. L. Salzberg, “Fast Gapped-Read Alignment with Bowtie 2,” *Nature Methods*, 2012.

[97] T. Laver, J. Harrison, P. O’neill, K. Moore, A. Farbos, K. Paszkiewicz, and D. J. Studholme, “Assessing the Performance of the Oxford Nanopore Technologies MinION,” *Biomolecular Detection and Quantification*, 2015.

[98] C. Lee, C. Grasso, and M. F. Sharlow, “Multiple Sequence Alignment using Partial Order Graphs,” *Bioinformatics*, 2002.

[99] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, “Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost,” *TACO*, 2016.

[100] V. I. Levenshtein, “Binary Codes Capable of Correcting Deletions, Insertions, and Reversals,” in *Soviet Physics Doklady*, 1966.

[101] H. Li, “Aligning Sequence Reads, Clone Sequences and Assembly Contigs with BWA-MEM,” arXiv:1303.3997 [q-bio.GN], 2013.

[102] H. Li, “Minimap2: Pairwise Alignment for Nucleotide Sequences,” *Bioinformatics*, 2018.

[103] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin, “The Sequence Alignment/Map Format and SAMtools,” *Bioinformatics*, 2009.

[104] H. Li, B. Ni, M.-H. Wong, and K.-S. Leung, “A Fast CUDA Implementation of Agrep Algorithm for Approximate Nucleotide Sequence Matching,” in *SASP*, 2011.

[105] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, “SOAP2: An Improved Ultrafast Tool for Short Read Alignment,” *Bioinformatics*, 2009.

[106] H.-N. Lin and W.-L. Hsu, “GSAlign: An Efficient Sequence Alignment Tool for Intra-Species Genomes,” *BMC Genomics*, 2020.

[107] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu, “A Tool for Multiple Sequence Alignment,” *PNAS*, 1989.

[108] D. J. Lipman and W. R. Pearson, “Rapid and Sensitive Protein Similarity Searches,” *Science*, 1985.

[109] Y. Liu and B. Schmidt, “GSWABE: Faster GPU-Accelerated Sequence Alignment with Optimal Alignment Retrieval for Short DNA Sequences,” *Concurrency Computation*, 2015.

[110] Y. Liu, A. Wirawan, and B. Schmidt, “CUDASW++ 3.0: Accelerating Smith–Waterman Protein Database Search by Coupling CPU and GPU SIMD Instructions,” *BMC Bioinformatics*, 2013.

[111] G. A. Logsdon, M. R. Vollger, and E. E. Eichler, “Long-Read Human Genome Sequencing and Its Applications,” *Nature Reviews Genetics*, 2020.

[112] H. Lu, F. Giordano, and Z. Ning, “Oxford Nanopore MinION Sequencing and Genome Assembly,” *Genomics, Proteomics & Bioinformatics*, 2016.

[113] A. Magi, R. Semeraro, A. Mingrino, B. Giusti, and R. D’Aurizio, “Nanopore Sequencing Data Analysis: State of the Art, Applications and Challenges,” *Briefings in Bioinformatics*, 2017.

[114] T. Mantere, S. Kersten, and A. Hoischen, “Long-Read Sequencing Emerging in Medical Genetics,” *Frontiers in Genetics*, 2019.

[115] G. Marçais, A. L. Delcher, A. M. Phillippy, R. Coston, S. L. Salzberg, and A. Zimin, “MUMmer4: A Fast and Versatile Genome Alignment System,” *PLoS Computational Biology*, 2018.

[116] V. Marx, “Nanopores: A Sequencer in Your Backpack,” *Nature Methods*, 2015.

[117] W. Miller and E. W. Myers, “Sequence Comparison with Concave Weighting Functions,” *Bulletin of Mathematical Biology*, 1988.

[118] O. Mutlu, “Accelerating Genome Analysis: A Primer on an Ongoing Journey,” *Keynote Talk at AACBB*, 2019.

[119] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, “Processing Data Where It Makes Sense: Enabling In-Memory Computation,” *MICPRO*, 2019.

[120] E. W. Myers and W. Miller, “Optimal Alignments in Linear Space,” *Bioinformatics*, 1988.

[121] G. Myers, “A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming,” *Journal of the ACM*, 1999.

[122] A. Nag, C. Ramachandra, R. Balasubramonian, R. Stutsman, E. Giacomin, H. Kambalasubramanyam, and P.-E. Gaillardon, "GenCache: Leveraging In-Cache Operators for Efficient Sequence Alignment," in *MICRO*, 2019.

[123] K. Nakano, A. Shiroma, M. Shimoji, H. Tamotsu, N. Ashimine, S. Ohki, M. Shinzato, M. Minami, T. Nakanishi, K. Teruya, K. Satou, and T. Hirano, "Advantages of Genome Sequencing by Long-Read Sequencer using SMRT Technology in Medical Area," *Human Cell*, 2017.

[124] National Center for Biotechnology Information, "GRCh38.p13," 2019. https://www.ncbi.nlm.nih.gov/assembly/GCA_000001405.28

[125] G. Navarro, "A Guided Tour to Approximate String Matching," *CSUR*, 2001.

[126] S. B. Needleman and C. D. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," *Journal of Molecular Biology*, 1970.

[127] C. Notredame, "Recent Progress in Multiple Sequence Alignment: A Survey," *Pharmacogenomics*, 2002.

[128] C. Notredame, D. G. Higgins, and J. Heringa, "T-Coffee: A Novel Method for Fast and Accurate Multiple Sequence Alignment," *JMB*, 2000.

[129] NVIDIA Corp., "NVIDIA TITAN V." https://www.nvidia.com/en-us/titan/titan-v/

[130] NVIDIA Corp., "nvprof." https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview

[131] Y. Ono, K. Asai, and M. Hamada, "PBSIM: PacBio Reads Simulator–Toward Accurate Genome Assembly," *Bioinformatics*, 2012.

[132] Oxford Nanopore Technologies Ltd., "GridION." https://nanoporetech.com/products/gridion

[133] Oxford Nanopore Technologies Ltd., "MinION." https://nanoporetech.com/products/minion

[134] Oxford Nanopore Technologies Ltd., "PromethION." https://nanoporetech.com/products/promethion

[135] Pacific Biosciences of California, Inc., "Sequel Systems." https://www.pacb.com/products-and-services/sequel-system

[136] B. Paten, D. Earl, N. Nguyen, M. Diekhans, D. Zerbino, and D. Haussler, "Cactus: Algorithms for Genome Multiple Sequence Alignment," *Genome Research*, 2011.

[137] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities," in *PACT*, 2016.

[138] P. A. Pevzner, H. Tang, and M. S. Waterman, "An Eulerian Path Approach to DNA Fragment Assembly," *PNAS*, 2001.

[139] J. Prado-Martinez, P. H. Sudmant, J. M. Kidd, H. Li, J. L. Kelley, B. Lorente-Galdos, K. R. Veeramah, A. E. Woerner, T. D. O'Connor, G. Santpere, A. Cagan, C. Theunert, F. Casals, H. Laayouni, K. Munch, A. Hobolth *et al.*, "Great Ape Genetic Diversity and Population History," *Nature*, 2013.

[140] A. Prohaska, F. Racimo, A. J. Schork, M. Sikora, A. J. Stern, M. Ilardo, M. E. Allentoft, L. Folkersen, A. Buil, J. V. Moreno-Mayar, T. Korneliussen, D. Geschwind, A. Ingason, T. Werge, R. Nielsen, and E. Willerslev, "Human Disease Variation in the Light of Population Genomics," *Cell*, 2019.

[141] M. A. Quail, M. Smith, P. Coupland, T. D. Otto, S. R. Harris, T. R. Connor, A. Bertoni, H. P. Swerdlow, and Y. Gu, "A Tale of Three Next Generation Sequencing Platforms: Comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq Sequencers," *BMC Genomics*, 2012.

[142] J. Quick, N. J. Loman, S. Duraffour, J. T. Simpson, E. Severi, L. Cowley, J. A. Bore, R. Koundouno, G. Dudas, A. Mikhail, N. Ouédraogo, B. Afrough, A. Bah, J. H. J. Baum, B. Becker-Ziaja, J. P. Boettcher *et al.*, "Real-Time, Portable Genome Sequencing for Ebola Surveillance," *Nature*, 2016.

[143] J. Quick, A. R. Quinlan, and N. J. Loman, "A Reference Bacterial Genome Dataset Generated on the MinION Portable Single-Molecule Nanopore Sequencer," *Gigascience*, 2014.

[144] J. A. Reuter, D. V. Spacek, and M. P. Snyder, "High-Throughput Sequencing Technologies," *Molecular Cell*, 2015.

[145] A. Rhoads and K. F. Au, "PacBio Sequencing and Its Applications," *Genomics, Proteomics & Bioinformatics*, 2015.

[146] R. J. Roberts, M. O. Carneiro, and M. C. Schatz, "The Advantages of SMRT Sequencing," *Genome Biology*, 2013.

[147] E. Rucci, C. Garcia, G. Botella, A. De Giusti, M. Naiouf, and M. Prieto-Matias, "SWIFOLD: Smith–Waterman Implementation on FPGA with OpenCL for Long DNA Sequences," *BMC Systems Biology*, 2018.

[148] SAFARI Research Group, "GenASM — GitHub Repository." https://github.com/CMU-SAFARI/GenASM

[149] SAFARI Research Group, "Shouji — GitHub Repository." https://github.com/CMU-SAFARI/Shouji

[150] D. Sankoff, "Minimal Mutation Trees of Sequences," *SIAP*, 1975.

[151] S. Schwartz, W. J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, and W. Miller, "Human–Mouse Alignments with BLASTZ," *Genome Research*, 2003.

[152] D. Senol Cali, J. S. Kim, S. Ghose, C. Alkan, and O. Mutlu, "Nanopore Sequencing Technology and Tools for Genome Assembly: Computational Analysis of the Current State, Bottlenecks and Future Directions," *Briefings in Bioinformatics*, 2018.

[153] J. Shendure, S. Balasubramanian, G. M. Church, W. Gilbert, J. Rogers, J. A. Schloss, and R. H. Waterston, "DNA Sequencing at 40: Past, Present and Future," *Nature*, 2017.

[154] T. F. Smith and M. S. Waterman, "Identification of Common Molecular Subsequences," *Journal of Molecular Biology*, 1981.

[155] M. Šošić and M. Šikić, "Edlib: A C/C++ Library for Fast, Exact Sequence Alignment Using Edit Distance," *Bioinformatics*, 2017.

[156] Synopsys, Inc., "Design Compiler." https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html

[157] J. D. Thompson, D. G. Higgins, and T. 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.

[158] C. Trapnell and S. L. Salzberg, "How to Map Billions of Short Reads onto Genomes," *Nature Biotechnology*, 2009.

[159] T. J. Treangen and S. L. Salzberg, "Repetitive DNA and Next-Generation Sequencing: Computational Challenges and Solutions," *Nature Reviews Genetics*, 2011.

[160] 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.

[161] Y. Turakhia, "Darwin — GitHub Repository." https://github.com/yatisht/darwin

[162] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A Genomics Co-Processor Provides up to 15,000x Acceleration on Long Read Assembly," in *ASPLOS*, 2018.

[163] E. Ukkonen, "Algorithms for Approximate String Matching," *Information and Control*, 1985.

[164] E. L. van Dijk, H. Auger, Y. Jaszczyszyn, and C. Thermes, "Ten Years of Next-Generation Sequencing Technology," *Trends in Genetics*, 2014.

[165] E. L. van Dijk, Y. Jaszczyszyn, D. Naquin, and C. Thermes, "The Third Revolution in Sequencing Technology," *Trends in Genetics*, 2018.

[166] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt, J. D. Gocayne, P. Amanatides, R. M. Ballew, D. H. Huson, J. R. Wortman, Q. Zhang *et al.*, "The Sequence of the Human Genome," *Science*, 2001.

[167] J. Wang, N. E. Moore, Y.-M. Deng, D. A. Eccles, and R. J. Hall, "MinION Nanopore Sequencing of an Influenza Genome," *Frontiers in Microbiology*, 2015.

[168] M. S. Waterman, "Efficient Sequence Alignment Algorithms," *Journal of Theoretical Biology*, 1984.

[169] M. S. Waterman, T. F. Smith, and W. A. Beyer, "Some Biological Sequence Metrics," *Advances in Mathematics*, 1976.

[170] J. L. Weirather, M. de Cesare, Y. Wang, P. Piazza, V. Sebastiano, X.-J. Wang, D. Buck, and K. F. Au, "Comprehensive Comparison of Pacific Biosciences and Oxford Nanopore Technologies and Their Applications to Transcriptome Analysis," *F1000Research*, 2017.

[171] A. M. Wenger, P. Peluso, W. J. Rowell, P.-C. Chang, R. J. Hall, G. T. Concepcion, J. Ebler, A. Fungtammasan, A. Kolesnikov, N. D. Olson, A. Töpfer, M. Alonge, M. Mahmoud, Y. Qian, C.-S. Chin, A. M. Phillippy *et al.*, "Accurate Circular Consensus Long-Read Sequencing Improves Variant Detection and Assembly of a Human Genome," *Nature Biotechnology*, 2019.

[172] S. J. Wilton and N. P. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," *JSSC*, 1996.

[173] F. Wu, S. Zhao, B. Yu, Y.-M. Chen, W. Wang, Z.-G. Song, Y. Hu, Z.-W. Tao, J.-H. Tian, Y.-Y. Pei, M.-L. Yuan, Y.-L. Zhang, F.-H. Dai, Y. Liu, Q.-M. Wang, J.-J. Zheng *et al.*, "A New Coronavirus Associated with Human Respiratory Disease in China," *Nature*, 2020.

[174] S. Wu and U. Manber, "Fast Text Searching Allowing Errors," *CACM*, 1992.

[175] Xilinx, Inc., "Vivado Design Suite." https://www.xilinx.com/products/design-tools/vivado.html

[176] H. Xin, J. Greth, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu, "Shifted Hamming Distance: A Fast and Accurate SIMD-Friendly Filter to Accelerate Alignment Verification in Read Mapping," *Bioinformatics*, 2015.

[177] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating Read Mapping with FastHASH," *BMC Genomics*, 2013.

[178] H. Xin, S. Nahar, R. Zhu, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu, "Optimal Seed Solver: Optimizing Seed Selection in Read Mapping," *Bioinformatics*, 2016.

[179] Y. Yang, B. Xie, and J. Yan, "Application of Next-Generation Sequencing Technology in Forensic Science," *Genomics, Proteomics & Bioinformatics*, 2014.

[180] F. Zokaee, H. R. Zarandi, and L. Jiang, "AligneR: A Process-in-Memory Architecture for Short Read Alignment in ReRAMs," *IEEE CAL*, 2018.

[181] F. Zokaee, M. Zhang, and L. Jiang, "FindeR: Accelerating FM-Index-Based Exact Pattern Matching in Genomic Sequences Through ReRAM Technology," in *PACT*, 2019.

[182] Q. Zou, Q. Hu, M. Guo, and G. Wang, "HAlign: Fast Multiple Similar DNA/RNA Sequence Alignment Based on the Centre Star Strategy," *Bioinformatics*, 2015.