Title

SALIENT: Ultra-Fast FPGA-based Short Read Alignment

Authors

Khaleghi, Behnam

Zhang, Tianqi

Martino, Cameron

et al.

Publication Date

2022

Peer reviewed

SALIENT: Ultra-Fast FPGA-based Short Read Alignment

***Abstract*—State-of-the-art high-throughput DNA sequencers output terabytes of short reads that typically need to be aligned to a reference genome in order to perform downstream analyses. Because alignment typically dominates the total run time of bioinformatics pipelines, a number of recent work sought to accelerate it in hardware. However, existing FPGA implementations did not fully optimize the alignment algorithms for the FPGA hardware and mainly focused on a subset of alignment problems, e.g., ungapped alignment with a limited number of mismatches, which hinder their practical utility. In this work, we analyze the existing alignment methods and identify and leverage opportunities for FPGA acceleration. Our alignment framework, SALIENT, first carries out an ultra-fast ungapped alignment, which supports a flexible number of mismatches. Based on the underlying bioinformatics pipeline and the information provided by the ungapped aligner, SALIENT then identifies a fraction of reads that need to go through its gapped aligner, thus improving alignment throughput. We extensively evaluate SALIENT using diverse datasets. Experimental results indicate that SALIENT, running on a single Xilinx Alveo U280 device, delivers an average throughput of 546 million bases/second, outperforming the state-of-the-art minimap2 software by 40×, and Bowtie2 by up to 107×, with comparable alignment accuracy. Compared to the existing ungapped FPGA aligners [1]–[4], SALIENT has 9.4–18× higher throughput/Watt, while compared to the gapped aligners [5], [6], it is 28–35× better. SALIENT achieves 7.6× higher throughput than Illumina DRAGEN Bio-IT Platform [7].**

I. Introduction

Advances in high-throughput and low-cost sequencing technologies have dramatically accelerated the generation of genomics data. As a result, genomics data size now doubles every seven months, outpacing Moore's law. For instance, the Illumina NovaSeq 6000 generates nearly 2.2 TB of data within 44 hours [8]. By 2025, genomics data is predicted to reach exabyte scale (10^{18}) and surpass YouTube and Twitter, requiring thousands of trillions of CPU hours for processing [9]. The application space of genomics data is enormous, from precision microbiome for personalized healthcare [10] to phylogenetic inference of SARS-CoV-2 genomes that enables global COVID-19 epidemiology [11].

Notwithstanding the diversity of applications, short read alignment is a common and significant step of bioinformatics pipelines, which finds the likely position of short DNA sequences of 25–200 base-pairs (bp) within a reference genome of thousands to billions of bases. The alignment also finds the edits (e.g., base change or insertion/deletion) between the read and the aligned part of the reference genome. State-of-the-art software such as minimap2 [12] have taken advantage of novel algorithmic innovations and hardware advancement to gain multiple times higher performance than previously standard software [13], [14]. Nevertheless, aligning the alluded massive data of a single sequencing run, even with recent software (details in Section V), can take above 200 hours, which is more than 4.5× slower than the sequencing throughput.

Recent work has also sought to accelerate read alignment in hardware [15]–[17]. However, the vast majority of FPGA alignment acceleration have only targeted *ungapped alignment* [1]–[4], [18]–[20] which, unlike commonly used software such as Bowtie2 [13] and minimap2 [12], does not support insertions or deletions (aka *indels* or *gaps*) in the sequenced reads. Indels or gaps are extremely common in all species [21] and have implications in the causes of a number of Mendelian diseases, acute myeloid leukemia, and other types of cancer [22]. Our experiments reveal that ungapped alignment, on average, fails to align 11.5% of the reads that could be aligned with gapped alignment. The few FPGA accelerators that support gaps [5], [6], [23] achieve at most 2× speedup over baseline software.

In this paper, based on our analysis of a comprehensive set of datasets that reveals ungapped reads are significantly more abundant than gapped reads (6.4× on average), we propose a framework dubbed SALIENT to speed up the alignment by decoupling it into two steps, ungapped and gapped alignment. It is beneficial since the ungapped alignment does not require costly *pairwise* sequence alignment (Smith-Waterman dynamic programming algorithm [24]) between the read and candidate chunks of the reference genome, which accounts for 60% of Bowtie2 execution time [23]. The first step of SALIENT performs an ultra-fast ungapped alignment that supports a flexible number of *mismatches* (positions where the read differs from the reference genome). This flexibility is crucial, as we observed that 25% of the ungapped short reads have more than two mismatches, whereas previous ungapped aligners can align reads with up to two mismatches [1]–[4] due to performance limitations. After that, SALIENT identifies the reads that require gapped alignment and passes them through its gapped aligner. It includes (1) reads that likely have gaps and could thus not be aligned via ungapped alignment, and (2) reads that *were* aligned via ungapped alignment but might obtain better alignment quality with gapped alignment.

To deal with the mismatches and indels of the reads that make the exact match of a read impossible, more recent alignment algorithms break a given read into smaller pieces called *seeds*, and look up the seeds on the reference. Our investigation discloses that memory access is the bottleneck of seed lookup and dictates the overall performance. Thus, in the proposed accelerator, we leverage a hash-based lookup which lowers the number of memory accesses of finding the location of seeds. To make it practically possible, we prudently optimize the hash-table by reducing its capacity and the required number of accesses while avoiding alignment accuracy degradation. To further improve the performance, we prioritize the candidate locations and early terminate to decrease the number of pairwise Smith-Waterman alignments.

We implemented our design on a Xilinx Alveo U280 FPGA [25] and compared it with previous FPGA-based aligners, including Illumina's DRAGEN platform, as well as commonly-used Bowtie2 and minimap2 software in terms of performance and accuracy. We evaluated SALIENT using a total of 16 datasets gathered from previous work (while most of the previous works try on one dataset each) to have a

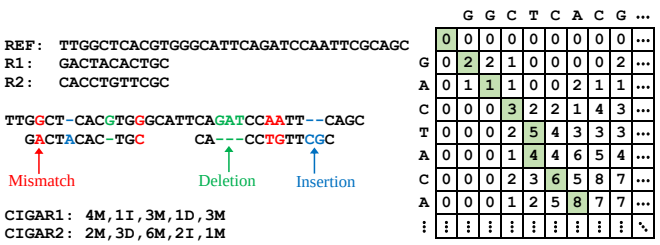

Fig. 1. An example gapped alignment. Here, every match has a score of $+2$, and mismatch and gap have a penalty of 1.

head-to-head comparison, especially since we observed that the relative performance and accuracy vary based on dataset statistics. SALIENT yields an average alignment throughput of 546M bases/second (845M ungapped, 81M gapped) which is 40× higher than minimap2 [12], and 37–107× better than Bowtie2 [13], with a similar accuracy. SALIENT improves the performance/Watt by 9.4–18× over the existing ungapped FPGA-based aligners [2]–[4], by 28–35× over the gapped FPGA platforms [5], [6], and by 7.6× over the commercial DRAGEN platform of Illumina.

II. Background and Analysis

A. Short Read Alignment

DNA is composed of paired strands of nucleotide bases (A, T, G, C) which are identified through the sequencing process, whereby the sequencing machine reads out the large genome as smaller subsequences (aka *reads*) of ∼50–200 base-pairs (bp). Short read alignment process identifies the locations wherein the short reads best align with the reference DNA of thousands (e.g., bacteria) to billions of bases (e.g., human) and the type of the differences between the short reads and the reference. Fig. 1 shows examples of gapped read alignments of reads R1 and R2. Reads are prone to errors in the form of insertions (indels), where an extra base is added, or deletions (gaps), where a base is missing. These may occur due to variants among species or sequencing errors. In Fig. 1, mismatches between the read and reference are distinguished by red, insertion with blue, and deletion by green.

Once the candidate positions are identified, the Smith-Waterman algorithm [24] performs *pairwise alignment* with the candidate locations of the reference to find the alignment score and edits (i.e., mismatches and indels). Fig. 1 shows the pairwise alignment for R1. The edits are represented by a so-called CIGAR string, in which M indicates a match or mismatch, and I/D means insertion/deletion. Note that since *ungapped* alignment does not deal with insertions/deletions, it does not need to run the Smith-Waterman algorithm and can use a more efficient base-to-base comparison (Hamming distance) to determine the mismatches between the short read and the reference.

Various algorithms have been proposed to find the candidate positions on the reference genome where the read may align. These algorithms differ in *indexing*, i.e., whether (1) they look up the entire read or (2) split the read into pieces (aka seeds), look up the seeds, and extend the reference near the position that a seed is found. Also, looking up the seeds can be done differently, e.g., by FM-index or Hash-table. We categorize and review the previous work based on their table generation (indexing) and seed lookup approach in the subsections below, along with more details regarding indexing algorithms.

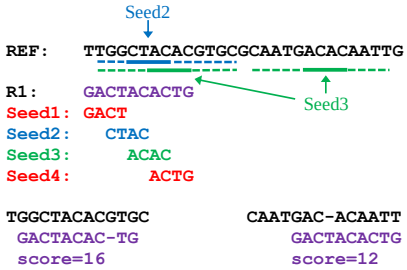

Fig. 2. Example read alignment using seed-and-extend.

B. Indexing with Suffix Arrays

Suffix array-based indexing is used in many alignment tools such as Bowtie [26] and BWA [27]. It converts the reference into suffix-tree and returns the match position stored in the leaf if a matching path is found. A popular algorithm is FM-index that uses Burrows-Wheeler Transform (BWT). FM-index is used in most of the FPGA accelerators [1]–[4], [6], [19]. FM-index is space-efficient, but it cannot handle gaps/indels. To handle mismatches, backtracking and bi-directional FM-index [26] are used, which replace the failed base (mismatch) with other alternatives and traverse the read from different directions to reduce the search space. Several FPGA accelerators have adopted this strategy while supporting only two mismatches due to the intractable growth of search space with the mismatch count [1], [2], [4].

C. Indexing using Suffix Array with Seed-and-Extend

Seed-and-extend technique facilitates the alignment in the presence of gaps and mismatches by finding sub-reads (*seeds*) instead of the whole reads, as seeds are shorter and have a higher likelihood of being error-free. In suffix array with seed-and-extend, the seeding step splits the reads into shorter fragments and finds the *perfectly matched* locations of these seeds in the reference using FM-index. Fig. 2 shows examples of seed-and-extend alignment. Read R1 is split into seeds seed1 to seed4. Seed1 and seed4 (shown in red) are not found on the reference. Seed2 (blue) is found in one position of the reference, which is distinguished by the same color (solid blue line). Seed3 (green) is found in two sites of the reference (one of which is the same position pointed by seed2). The dashed lines indicate extending near the obtained seed positions on the reference. Finally, a pairwise alignment using the Smith-Waterman algorithm between the read and each of the candidate locations finds the best alignment and its edits.

Several FPGA accelerators use FM-index with seed-and-extend [3], [6], [19], [23]. Of these, only [6], [23] support gapped alignment. The work in [6] supports gaps by calling Smith-Waterman pairwise alignment on the candidate locations. Using Xilinx UltraScale+ VU9P, [6] is 2× faster than Bowtie2 software [13] at the cost of 2.8% accuracy loss. However, if we match Bowtie2 and [6] accuracy, it turns out that Bowtie2 can be faster ([14] shows that using the fast setting of Bowtie2 makes it 4× faster with an accuracy within 2% of the sensitive setting). The study in [23] integrates Bowtie2 with Xilinx VU9P to offload the Smith-Waterman calls to FPGA, but the end-to-end speedup is only 35%.

D. Indexing with Hash-based Seed-and-extend

The indexing step of hash-table techniques extract length-L seeds of the reference and store their positions in a table.

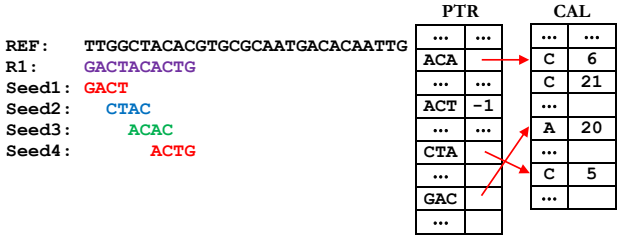

Fig. 3. Hash-based seed lookup.

This technique is used by software BLAT [28], BFAST [29], and minimap2 [12], as well as in an FPGA accelerator [5]. To align a read, a subset of its length-L seeds are extracted, and their positions on the reference is looked up using the prebuilt hash-table that reduces memory accesses compared to FM-index iterative table.

Fig. 3 shows the seed lookup using hash-table. A two-stage table is adopted to handle seeds with multiple positions during indexing the reference. When generating the table, each seed of the reference is split into two parts, *seed_ptr* (prefix) and *seed_cal*. The seed_ptr is used as an address to point the row in the CAL table where the positions of all reference seeds starting with seed_ptr prefix are stored successively. The rows that correspond to a certain seed_ptr are called a CAL *bucket*. Note that not only a seed might appear on multiple positions on the reference, several seeds might also share the same seed_ptr; hence, a bucket stores the positions of all seeds that start with the same seed_ptr prefix.

Items of the PTR table are unique, so access to it is straightforward: to create and then access PTR[see_ptr], we can set $A \leftarrow 00$, $C \leftarrow 01$, $G \leftarrow 10$, $T \leftarrow 11$. Each seed is split into seed_ptr and seed_cal parts. The value in the PTR table refers to the row in the CAL table that stores the positions of all seeds that begin with seed_ptr. Starting from that row, the CAL table is probed to find all seed_cals. Fig. 3 shows an example where seeds have a length of 4, split into seed_ptr of length three and seed_cals of length one. To find the position of the seed ACAC, it is likewise split to seed_ptr = ACA and seed_cal = C. First, the ACA^{th} cell in the PTR table is accessed (i.e., PTR[see_ptr]), which returns the proper row index of the CAL table (bucket's head) to search for seed_cal = C.

E. Comparison of Indexing Techniques

Both the suffix-array and hash-based techniques heavily rely on random memory accesses. To estimate and compare the performance of these different indexing techniques, in Fig. 4 we benchmarked the random access throughput of Xilinx Alveo U280 DDR4 bank by issuing 32b data random accesses (black curve labeled as 32b) to a 4 GB table (representing the FM-index and PTR tables). The latency of an access includes the M-AXI adapter and AXI interconnect buffers, and MIG to DDR latency [30]. When the number of kernels is large enough, the throughput is saturated and a maximum throughput of ∼96 M access/second is achieved. Since suffix-array (FM-index) technique needs at least one access per *each read base*, the throughput of these techniques is limited to ∼96 M base/second. In practice, the seeds can overlap, so more than one access per base is needed (i.e., lower performance).

Using hash-table, however, we need fewer accesses per *seed*. The seed length, L, is usually ∼20, and the sliding step of seeds is at least $0.5\sqrt{|Q|}$ (for Q denoting the read length) in high-sensitivity (accuracy) alignment [13]. Accordingly, Fig. 5 shows the upper bound throughput of hash-based indexing. We can see that, e.g., a read length of 150 bp can achieve a throughput of at least 758 M base/second. Accordingly, in SALIENT, we leverage hash-based seed lookup. However, realizing such a throughput in practice faces several challenges that we elaborate on and address in the following section.

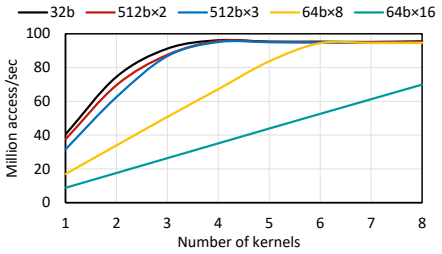

Fig. 4. Throughput of accesses to a DDR4 bank, shared between up to eight kernels instantiated in the host code using Xilinx Vitis software platform. The 32b label shows random access to 32b data (used in FM-index and PTR table), the others indicate searching the CAL table.

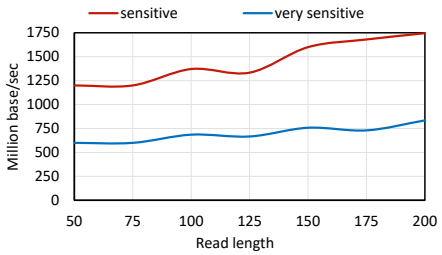

Fig. 5. Throughput bound of hash-based alignment. The throughput generally improves with the read length as the number of seeds grows by $\propto\sqrt{|Q|}$ while number of the processed base improves by $\propto|Q|$ ($|Q|$ is the read length).

III. SALIENT Algorithm

A. SALIENT *Alignment Flow*

Overview: Fig. 6 depicts the alignment flow of SALIENT. It takes advantage of a two-step alignment flow to decouple the ungapped and gapped alignment to avoid costly Smith-Waterman pairwise alignments needed in gapped alignment. Both ungapped and gapped alignment require at least one seed of the read exist on the reference. Thus, if no seed is found in the hash-table, the read is flagged as 'not aligned'. After that, an ungapped alignment is first performed. If the read does not align, it can be possible some insertion or deletion has shifted some part of the read bases so a straight Hamming distance could not find the similarity. Such as read is passed to our gapped alignment, which itself is enhanced by prioritizing the candidate locations suggested by seeds to lower the number of pairwise Smith-Waterman calls. On the other hand, it is also possible that a read could be aligned by the ungapped aligner, but might need to be aligned with gapped alignment, as well. It can happen, for example, when an insertion or deletion is occurred in the tail of a read. It causes a small part of the read tail to have a high percentage of mismatches due to shifting versus the reference chunk. Fig. 7(a) shows such an example, where the ungapped alignment achieves good overall matching except in the last four bases. Such a read is flagged as a potential read for gapped alignment, depending on the application and requirements of the downstream analysis. For instance, the host filtering step of microbiome pipeline [14] aligns the microbial reads to the host (human genome) to discard the human reads for the rest of pipeline. In such

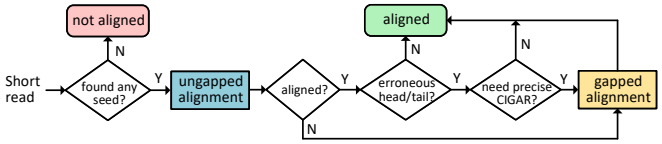

Fig. 6. Two-step flow of SALIENT to reduce gapped alignments.

```
TTGGCTACACGTGCGCAATGATCCAATTG      TTGGCTACACGTGCGCAATGATCCAATTG
  GCCACACGTGCGCAAGATC                GGCTACTACGTGCGCAATG

TTGGCTACACGTGCGCAATGATCCAATTG      TTGGCTAC-ACGTGCGCAATGATCCAATTG
  GCCACACGTGCGCAA-GATC               GGCTACTACGTGCGCAATG
       (a)                                (b)
```

Fig. 7. (a) Successful ungapped alignment as a gap (deletion) is occurred in the end-point of the read. (b) Unsuccessful alignment as a gap (insertion) is occurred in the middle of the read, causing many mismatches.

cases, the exact/best alignment of the reads and reference is not obligatory as a successful alignment (that passes the threshold score) by the ungapped stage provides enough information.

Algorithm 1 outlines the alignment (ungapped or gapped) procedure of SALIENT. Length L seeds are extracted by a moving step of S and split into seed_ptr and seed_cal parts. For each seed $\mathcal{S}$ and its reverse complement $\mathcal{S}_{rc}$, we keep the smaller one (line 3) for the reason we explain in subsection III-B. All seed_ptrs are accessed in the PTR table to obtain their bucket head in the CAL table. The bucket is fetched by multiple wide memory accesses and is searched to compare its seed_cals with the query. The CAL table stores the position of the reference seeds (not the read's). To calculate the mapped position of the read, we adjust the CAL value based on the position of the queried seed on the read (line 10). The pairwise alignment (line 19) carries out Hamming distance for ungapped, and Smith-Waterman for gapped alignment. Algorithm 1 returns the alignment position on the reference and the edit information, which is used to decide whether a read aligned by the ungapped stage needs gapped alignment.

Prioritizing: Multiple seeds may point to the same reference position, e.g., when a read perfectly matches, all seeds return the same position. Also a seed might exist on different sites and return various positions. Therefore, we first store the frequency of each candidate position and carry out pairwise alignment (or Hamming distance in ungapped stage) starting from the most-frequent position (lines 17–18). This is because when k seeds of a read point to the same position on the reference, at least $\mathrm{L}+(k-1)\times\mathrm{S}$ bases of the read match with the reference (up to $\mathrm{L}\times\mathrm{S}$ when those seeds do not overlap). Thus, a candidate position that is suggested with more seeds has higher likelihood of alignment. *We observed that prioritizing the candidate locations reduces the number of Smith-Waterman calls by $\sim$2$\times$ before finding a valid alignment.*

Efficacy: The two-stage alignment is critical for high performance. We observed that we could integrate up to eight Smith-Waterman units in the design (four parallel kernels, two units per kernel), each takes 640 cycles for a pairwise alignment of a read and the reference subsequence. The cycle count could be reduced, but at a proportional cost of higher resource utilization, so the overall Smith-Waterman throughput remains similar. Operating at 100 MHz and ignoring memory and control stalls, the aggregate throughput of Smith-Waterman modules is $\sim$187 M base/second, which nullifies the seed lookup throughput premise of the hash-table seed lookup. This throughput further deteriorates as a read can have multiple candidate locations for pairwise alignment.

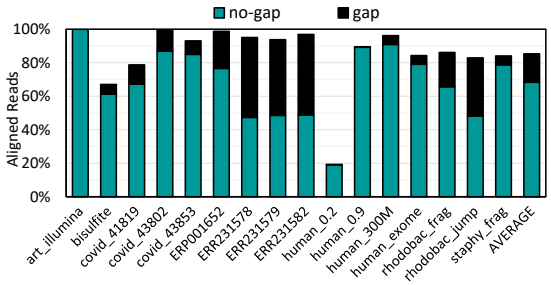

Fig. 8. Alignment rate of different read datasets using Bowtie2 *very-sensitive* option. The average alignment percentage is 85.32%, out of which 17.05% contains gap (insertion, deletion, or both).

Algorithm 1: SALIENT alignment algorithm

Inputs: read Q, ref R, seed length L, seed step S, hash tables PTR and CAL, alignment score threshold ξ
Output: position pos, CIGAR cigar

```
1: pos_count ← {}
2: for i from 0 to (|Q|−L)/S do
3:     seed ← min(Q[i·S : i·S+L], rev_cmp(Q[i·S : i·S+L]))
4:     seed_ptr, seed_cal ← seed[0:29], seed[30:2L]
5:     row ← PTR[seed_ptr]
6:     if row ≠ −1 then
7:         bucket ← CAL[row : row +3]
8:         for j from 0 to 32 do
9:             cal, pos ← bucket[j][0:16], bucket[j][16:48]
10:            pos ← pos − i×S
11:            if cal = seed_cal then
12:                pos_count[pos]++
13:            end if
14:        end for
15:    end if
16: end for
17: pos_count = sort_by_value(pos_count)   // descending
18: for pos in pos_count do
19:    score, cigar ← Pairwise(Q, R[pos : pos +|Q|]) // or Hamming
20:    if score ≥ ξ then
21:        return pos, cigar
22:    end if
23: end for
24: return −1
```

The efficacy of a two-stage alignment entails that the number of reads that require gapped alignment be non-dominant. To investigate it, we gathered 16 short read datasets from previous studies (11 of which are used in FPGA aligners, detailed in Section V) and aligned using Bowtie2's very sensitive setting. As Fig. 8 shows, for the average alignment rate of 85.32% among all datasets, only 17.05% of the reads ended up in an alignment with gap (68.27% ungapped). Interestingly, when we prevent gaps in aligning, the alignment rate becomes 73.88%, which is 5.61% higher than the expected 68.27%, leaving only 11.4% of reads to need gapped alignment. The reason that 5.61% of gapped alignments could be aligned despite preventing gap is explained by the insertions/deletions in the head or tail of a read that we discussed above. Recap that an alignment is accepted if the penalty is higher less a threshold. For instance, the default threshold of Bowtie2 for 150 bp reads is 90, and the mismatch (indel) penalty is less than 6 (5 for indel). Hence, alignment of a 150 bp read can tolerate at least 15 mismatches (18 gaps). Therefore, if a gap exists in the head or tail bases of a read, it can still be aligned with *ungapped* alignment since the penalties due to insertion/deletion do not exceed the threshold.

B. Indexing Optimizations

There are two main challenges with hash-based indexing. First, for large genomes such as the human reference, the size

of the CAL table that stores the position of all seeds becomes larger than the FPGA DRAM capacity. Second, the buckets sizes of CAL table are different and some buckets can have hundreds of rows. Thus, while we can have high throughput accesses to the PTR table, CAL table becomes bottleneck. In the following, we explain our optimizations of the hash-based indexing to address these challenges.

Storing numerically smaller seeds: For a reference genome R, the CAL table consists of $|\mathrm{R}|$ rows, where each row stores a seed_cal with an integer indicating one position of the corresponding seed on the reference, as shown in Fig. 3. Moreover, in paired-end reads in which a genome is sequenced from both ends, it is unknown whether a query read is in the forward or reverse strand. Storing the seeds of both forward and reverse strands increases the CAL table size to $|\mathrm{R}|$ rows. Accordingly, the human genome CAL table needs 3.1 billion rows of eight bytes (one int for seed_cal and one for the position) for each of the reference and its reverse complement, ending up in a 46.6 GB table. To avoid storing all the seeds of both reverse and forward strand, during the reference indexing, instead of storing both a seed $\mathcal{S}$ and its reverse complement $\mathcal{S}_{rc}$, we only store the numerically smaller one (by setting $\mathrm{A} \leftarrow 00$, $\mathrm{C} \leftarrow 01$, $\mathrm{G} \leftarrow 10$, $\mathrm{T} \leftarrow 11$). Thereafter, for any extracted query seed $\mathcal{S}$ of a read, we only look up $\min(\mathcal{S}, \mathcal{S}_{rc})$ knowing that for that position of the reference where the seed aligns to, we have also stored only $\min(\mathcal{S}, \mathcal{S}_{rc})$. Thus, without missing any seed, the CAL table shrinks by half, i.e., 23.3 GB.

Limiting CAL row size: Each CAL row consists of a seed_cal cell that stores part of the seed bases, and another cell that stores the seed position. The seed position needs to be a 32 bit integer so it store any value between 0 and 3.1 billion for the large human genome. For the seed_cal, we limit the number of bases to eight, so that $|\text{seed\_cal}| \leq 16$. Thus, each row needs 48 bits and the CAL table size further reduces to 17.5 GB. $|\text{seed\_cal}| \leq 16$ bits is a reasonable decision as usually seed length L is 20–22 bases (40–44 bits). Specifically, we use a seed length of 21 bases and set seed_cal to six bases (12 bits) which leaves 15 bases to seed_ptr part of the seed and keeps the PTR table size small (4 GB). Note that we could use larger seed_cal (up to eight base) as well, but that makes seed_ptr length smaller and more seeds will share the same seed_ptr prefix. It increases the bucket sizes, and hence, search latency of the CAL table.

Bucket size reduction and seed discarding: To avoid making the CAL table search performance bottleneck, we limit the number of rows to be searched (i.e., the bucket size). According to Fig. 4, with four parallel kernels, we can have $3{\times}512$ b searches (three 512 bits accesses to consecutive addresses) of the CAL table and yet keep up with the throughput of the requests from the PTR table. With $3{\times}512$ bits, we can search at least 21 ($\frac{2\times512}{48}$) and up to 32 ($\frac{3\times512}{48}$) rows of a bucket, depending on the head of the target bucket in 512 bit packed data (more fine-grained access such as 64 bit could mitigate the aligning issue but its throughput will be significantly smaller as shown in Fig. 4). Therefore, we limit the bucket size to 32 rows. However, a fraction of buckets can exceed the 32 rows capacity as alluded above. Some software such as BFAST [29] that leverage hash-table simply discard the highly-frequent seeds during the reference indexing. Nevertheless, we observed such a solution results in accuracy loss as certain reads do not find any seed after discarding high-frequent reference seeds from the table. To reduce the size of a bucket that exceed the maximum limit, we sort its seed_cals by the number of positions they point to, and randomly discard half of the positions (i.e., CAL rows) of the most frequent seed_cal. We repetitively sort and discard until the bucket size decreases to 32. With this strategy, we store as many different seeds as possible, which not only precludes the CAL table access bottleneck, but also shrinks it to 13.4 GB.

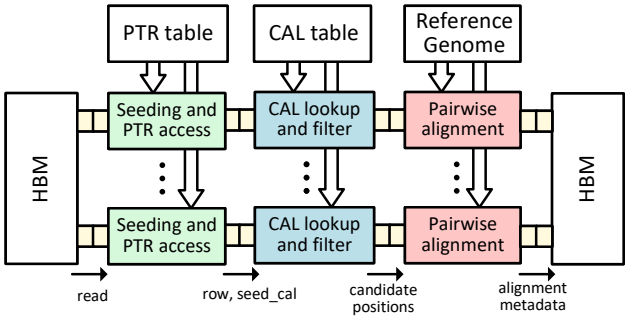

Fig. 9. Top-level diagram of the aligner design. The ungapped and gapped implementations differ in the pairwise aligners and number of kernels. The tables are stored in DRAM banks.

IV. SALIENT Implementation Details

Fig. 9 shows the top-level diagram of SALIENT. Both the ungapped and gapped stages have a similar architecture except the pairwise alignment unit which is Hamming distance in the ungapped, and Smith-Waterman in the gapped alignment. The number of kernels is also different as Smith-Waterman engines consume more resources. Multiple copies of the same kernel are instantiated (which, in our implementation, is eased by leveraging the Xilinx Vitis software platform).

A memory access faces the M-AXI adapter, AXI interconnect, and MIG to DDR latency that adds up to ∼80 cycles according to our experiments, which concurs with [30] as well. Such latency cannot be circumvented by simply issuing simultaneous requests by a limited number of kernels. Therefore, we opted to hide the memory pipeline latency by streaming the accesses. Instead of accessing the PTR table for a seed, using its result for CAL fetch, performing pairwise alignment, and repeating for the next seed, we separate these stages in a dataflow fashion. The seeding module fetches a read from a 512 bit HBM channel, extracts its seeds (in parallel, i.e., unrolled loop), and issues 32b accesses to the PTR table (stored in the DDR4[0] bank) using the seed_ptr part of all the seeds successively in a loop. Note that on the host side we simply pack the ASCII characters into an int32 to avoid costly preprocessing; converting the char to lower-bit nucleotides are done in parallel over all bases in the same seeding function. The result of each call to the PTR table is written to the output dataflow buffer of the seeding module. Thus, the ∼80 cycle latency is only observed once per a read (the first seed only). It can be further improved by batching multiple reads together to amortize the memory pipeline latency. However, we did not find batching necessary since, by using dataflow and multiple kernels, we could saturate the memory bandwidth.

The CAL module receives all the returned PTR values (addresses to CAL buckets) along with the seed_cal from the

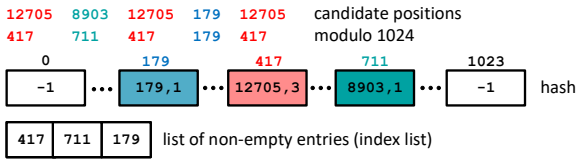

Fig. 10. Structure used for the candidate position frequency counter.

dataflow buffer. This module stores all the received inputs in a local temporary array using BRAMs and starts fetching the CAL buckets from DDR4[1] bank after receiving all the PTR values from the previous module. It fetches each bucket with 512b×3 accesses. Searching the buckets and comparing with the seed_cal cell of each bucket is done next. In case of a match, it saves the candidate position (the second element in a bucket's row; lines 9–12 of Algorithm 1) to a counter-like structure to keep track of the candidates frequencies.

Fig. 10 shows the structure we use to track the frequency of candidate positions. We use a 1024-element array for position-frequency pairs and call it a *hash* array. For a given candidate position, we find the right hash index by using modulo 1024, e.g., $12705 \bmod 1024 = 417$. If the hash entry is empty, we write $'12705, 1'$. Otherwise, we update the frequency by one, i.e., $'12705, 1'$ replaces with $'12705, 2'$. Also, an auxiliary list, namely *index* list, tracks the non-empty hash indexes. When an index of the hash is empty (i.e., new entry to that index), the new index is added to the index list. Once all candidate locations are processed, the index list is used to read the non-empty hash entries and pass them through a bitonic sort that sorts the pairs based on their frequency value. We limit the candidate positions to 32, hence the bitonic sort has a constant size. In case there are less than 32 locations, the rest inputs of the sort module are set to -1. On very rare occasions, when a read has more than one candidate location mapping to the same hash entry, we skip the new values.

Finally, the sorted candidate positions are written to the inputs buffer of the pairwise alignment unit. Since the number of unique candidates is limited, and usually the first candidates have a higher chance of successful alignment, this module fetches the reference chunk only when a candidate location needs to be examined (aligned). The reference is stored in the HBM banks, packed and fetched as 32 bit as we need fine-grained (base-level) access to the reference. In the gapped alignment, we fetch extra 16 bases from each end of the reference chunk to account for indels in the read.

For the Hamming distance in the ungapped alignment, we use a bitwise XOR between the read and reference chunk, followed by a pipelined popcount to count the number of '1's (mismatches). For the gapped pairwise alignment, we implement the Smith-Waterman algorithm according to Fig. 11. The value of a matrix cell depends only on its left, top, and top-left cells. Thus, all the cells in the same anti-diagonal can be computed simultaneously, given that the previous anti-diagonal is computed before. At each cycle, the one-dimensional array of processing engines (PEs) accomplishes one anti-diagonal of the matrix. A PE only needs to store the cell values of the same row (e.g., PE_3 stores the third row's values in successive cycles). Thus, each PE has a simple stack (BRAM) and pushes a new value in successive cycles. At cycle n, the left, top, and top-left cells for a PE_k are, respectively, $PE_{k,n-1}$, $PE_{k-1,n-1}$, and $PE_{k-1,n-2}$ where $PE_{k,n}$ denotes the

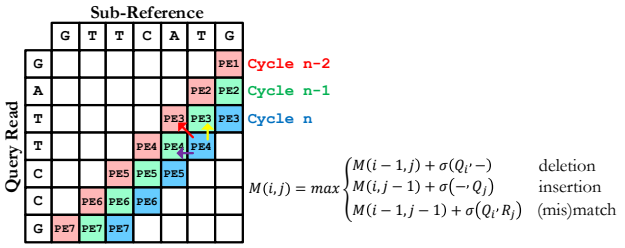

Fig. 11. Smith-Waterman dynamic programming algorithm. σ is the score or penalty of match, mismatch, insertion, and deletion.

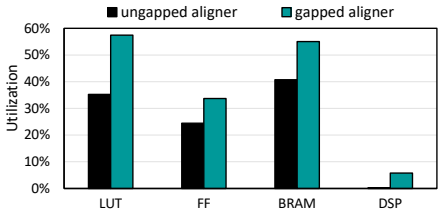

Fig. 12. Resource utilization of SALIENT aligners.

value of PE k at cycle n. Thus, in every cycle, each PE only needs the top stack values of itself or its adjacent PE.

V. Experiments and Results

A. Experimental Setup

We implemented SALIENT using Xilinx Vitis HLS 2021.2 on Alveo U280 accelerator card [25]. The FPGA has two 16 GB DDR4 banks and an 8 GB High-Bandwidth Memory with 32 pseudo channels. The ungapped aligner comprises eight kernels instantiated using the Xilinx Vitis software platform and achieved a frequency of 150 MHz. The kernels share the same PTR table (DDR4[0] bank), CAL table (DDR4[1] bank), and reference genome (HBM). The reads of each kernel are in a different HBM bank. The gapped aligner consists of four kernels, wherein each kernel comprises two Smith-Waterman engines and could achieve 100 MHz. Since there are usually more than one candidate location per read, we decided to have two Smith-Waterman units per kernel to parallelize the pairwise alignments of a read. Resource utilization of the aligners is reported in Fig. 12. Eight ungapped aligner kernels can saturate the hash-table access throughput, so we did not increase the number of kernels. While the gapped aligner could benefit from more kernels as the memory is not its bottleneck, the routing fails when further increasing the kernels or the Smith-Waterman units. A major part of the BRAMs in the ungapped aligner are used by the AXI interfaces and dataflow FIFOs of the kernels. The gapped aligner has fewer kernels, but each kernel consists of two Smith-Waterman units, where each PE uses a BRAM to store the relevant matrix cells.

Table I summarizes the total 16 datasets we used to evaluate SALIENT, most of them are compiled from the previous FPGA aligners (distinguished in bold). The datasets provide a wide range of short read sizes from 75 to 150 bases. Also as we showed in Fig. 8, the datasets present a wide range of gaps from 0.02% up to 48.1%. This is crucial for a head-to-head comparison and also a thorough evaluation of SALIENT as the gap percentage is a decisive factor in the performance.

We compare the performance of SALIENT with Bowtie2 v2.4.5 and minimap2 v2.24-r1122, both running on Ubuntu 18.04 installed in a system with Intel Gen-11 Core i7-11700K @4.8 GHz and 80 GB of physical memory. The Alveo U280

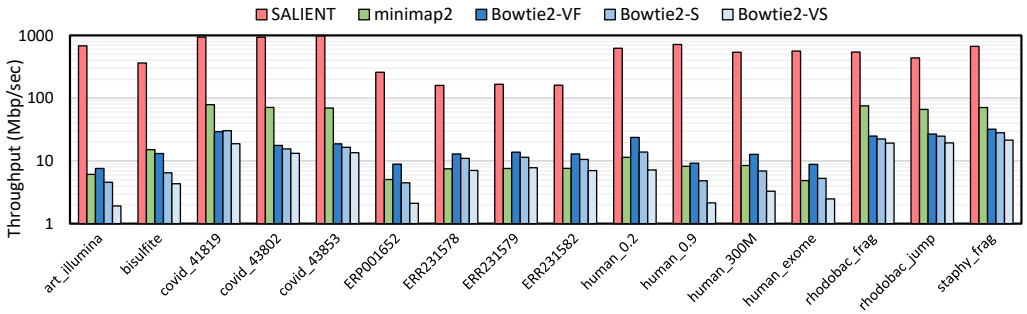

Fig. 13. Overall performance (million base-pair/second) comparison of SALIENT and software tools. –VF, –S, and –VS denote Bowtie2 in *very-fast*, *sensitive*, and *very-sensitive* modes. The ungapped and gapped stages of SALIENT use a seeding step of 15 and 7, respectively.

TABLE I
SHORT READ DATASETS USED IN OUR EXPERIMENTS. DATASETS USED IN PREVIOUS FPGA STUDIES ARE DISTINGUISHED IN BOLD.

Dataset	Description
art_illumina	100 bp synthetic human reads [6]
bisulfite	75 bp synthetic human chromosome 22 reads [2]
covid_x (×3)	151 bp sequenced SARS-CoV-2 reads [31]
ERP001652	90 bp sequenced human reads [3]
ERRx (×3)	101 bp sequenced human reads [1]
human_0.2, 0.9	150 bp synthetic human reads contaminated with microbial [14]
human_300M	101 bp sequenced human reads [4]
human_exome	76 bp sequenced human exome data [32] (as in [5])
rhodobac_x (×2)	101 bp sequenced Rhodobacter sphaeroides reads [1]
staphy_x	101 bp sequenced Staphylococcus aureus reads [1]

card is also installed on the same machine. We used all the threads (16) for both softwares. For Bowtie2 we tried *very-fast* (–VF), *sensitive* (–S), and *very-sensitive* (–VS) options. In SALIENT, we use the same score/penalty of Bowtie2. We also compare the performance with the previous FPGAs [1]–[6] that report the absolute throughput numbers. We reviewed the underlying alignment method of each study in Section II. We also compare SALIENT with Illumina DRAGEN Bio-IT Platform v3.5.7 [7] which embraces an Alveo U200 FPGA.

B. Performance Comparison

Comparison with CPU software: Fig. 13 compares the performance of SALIENT and widely-used software aligners in terms of million base/second. SALIENT on average, achieves a throughput of 546 M base/second. Averaged over all the benchmarks, SALIENT outperforms the minimap2 by 40×, and Bowtie2 by 37×, 56×, and 107× when Bowtie2 runs in very-fast, sensitive, and very-sensitive modes. The ungapped stage of SALIENT uses a seeding step of 15 (similar Bowtie2 default) which, depending on the read length, can theoretically achieve a *seeding* throughput of 1200–1750 M base/second according to Fig. 5. However, since usually more than one candidate location per read is found, the effective throughput is diminishes. The gapped aligner of SALIENT uses a seeding step of 7 (similar to the *very-sensitive* mode of Bowtie2), so it can try more seeds for the unaligned reads.

In certain datasets such as art_illumina that the percentage of the gapped alignments is very low, SALIENT's speedup is massive, e.g., 112× over minimap2 and 356× over Bowtie2-VS, whereas in ERR231578 which is another human dataset, the speedup is 21× due to higher calls of the gapped aligner. On the other hand, in datasets such as covid_43802 that also has relatively high gap rate (12.8%, shown in Fig. 8), SALIENT throughput is very high (938 Mbp/second) because we observed that 10.8% out of the 12.8% could be mapped with the fast ungapped aligner as the gaps occur in head/tail

TABLE II
DETAILED PERFORMANCE RESULTS OF SALIENT.

Dataset	Gapped calls %	Throughput (M bp/second)			Throughput/Watt	
		Ungapped	Gapped	Overall	Ungapped	Overall
art_illumina [6]	1.2%	750	92	683	19	18
bisulfite [2]	6.7%	563	69	363	14	10
covid_41819	0.9%	1073	72	941	28	25
covid_43802	1.0%	1073	72	938	28	25
covid_43853	0.6%	1073	72	978	28	25
ERP001652 [3]	19.7%	675	82	259	17	8
ERR231578 [1]	45.7%	758	93	160	19	5
ERR231579	43.4%	758	93	167	19	5
ERR231582	45.2%	758	93	161	19	5
human_0.2 [7]	4.7%	1066	71	625	27	17
human_0.9 [7]	3.3%	1066	71	716	27	20
human_300M [4]	6.5%	1066	71	539	27	15
human_exome [5]	0.2%	570	70	562	15	14
rhodobac_frag	4.8%	758	93	544	19	15
rhodobac_jump	8.9%	758	93	437	19	12
staphy_frag	1.6%	758	93	670	19	18

of the reads. Notice that for such small datasets, the indexing tables of software tools is small and fits in the cache.

SALIENT performance details: Table II reports the (1) ratio of reads that call the gapped aligner, (2) the ungapped aligner throughput (which depends on the read length), (3) our gapped aligner throughput, and (4) throughput (Mbp)/Watt for the ungapped and overall flow. Note that calls to the gapped aligner can be more or less than the gapped reads. For instance, the majority of the COVID gapped reads could be aligned using the ungapped aligner. On the other hand, a dataset such as human_0.2 with only 0.07% gapped reads calls the gapped aligner for 4.7% of the reads. The majority of such reads contain at least one seed that (accidentally) overlap with the reference genome, which, after an unsuccessful ungapped alignment, needs to be examined with the gapped aligner. To obtain the throughput/Watt, we measured the FPGA power using the xbutil query utility. We observed an average power of 39 W and 32.5 W for the ungapped and gapped aligners.

Comparison with ungapped FPGA aligners: We use the information of Table II to compare SALIENT with previous the FPGA-based aligners. Fig. 14 shows the performance improvement of SALIENT over previous works. SALIENT achieves 4.3–24.0× speedup over the previous *ungapped* FPGA accelerators [1]–[4]. The main reason of SALIENT efficiency is due to using hash-based indexing, as elaborated on Section II-E, delivers significantly higher seed lookup throughput over these FM-index-based techniques. To account for the difference in power consumption, especially as several of previous studies use multi-FPGA platforms, Fig. 14 also compares throughput/Watt (note that [1] has not reported power consumption). Since SALIENT consumes less power than these platforms, its throughput/Watt range improves to 9.4–18×. Note that these works support up to 2 mismatches,

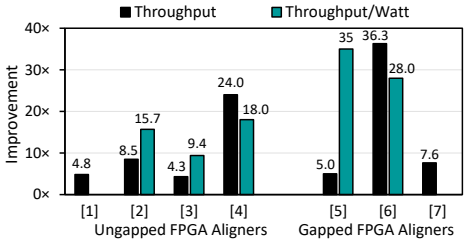

Fig. 14. Improvement of SALIENT over FPGA aligners [1]–[7].

while the default setting of ungapped SALIENT supports 6–9 mismatches depending on the read length. The relative performance of SALIENT further improves if we limit the number of mismatches (which reduces seed lookups).

Comparison with gapped FPGA aligners: According to Fig. 14, SALIENT improves the throughput (throughput/Watt) by 5–36.3× (28–35×) over gapped aligners, including DRAGEN [7]. Similar to SALIENT, [5] also uses hash-table to lookup seeds, and achieves an overall throughput of ∼112 Mbp/second using a system of six Pico M-503 boards, each equipped with a Virtex-6 LX240T. SALIENT improves the throughput (throughput/Watt) by 5.0× (35×) over [5] on a similar dataset. The main source SALIENT's advantage stems from the ungapped aligner that achieves a throughput of 570 Mbp/second (see Table II), and since the gapped aligner is only called for 0.2% of reads, the overall throughput is 562 Mbp/second. Even if we merely compare the gapped stage of SALIENT, it yields 70 Mbp/second on a *single device* with two DIMMs, whereas [5] distributes the reads over *six boards* with independent DIMMs with a *total* throughput of 112 Mbp/second. Having multiple devices with separate DIMMs (that alleviates per-device routing congestion) allows having more Smith-Waterman units to further improve the pairwise alignment bottleneck.

Compared to [6] that uses a Xilinx UltraScale+ VU9P, SALIENT improves the throughput (throughput/Watt) by 36.3× (28×) on the same dataset. [6] uses FM-index for seed lookup which, in Section II-E, we showed that has a limited throughput. Especially, to reduce the size of FM-index table, [6] uses bucketing that increases the number of accesses per base and further degrades the effective throughput.

Finally, to compare with Illumina DRAGEN [7], we used the human genome datasets human_0.2 and human_0.9 used in microbiome pipeline [14]. Thanks to its two-stage flow with an ungapped throughput of 1066 Mbp/second, SALIENT's overall throughput on these two datasets (average 670 Mbp/second) improves the DRAGEN's throughput by 7.6×. The gapped stage of SALIENT alone achieves 71 Mbp/second (Table II) which is close to DRAGEN's throughput. DRAGEN uses Alveo U200 card that consists of *four* DDR4 banks, which can be used to replicate either or both of our PTR and CAL tables and place the Smith-Waterman units within the adjacent SLRs (super logic regions) to alleviate the routing congestion and/or improve the frequency, hence, further increase the throughput of SALIENT's gapped stage.

C. Alignment Rate

Fig. 15 compares the alignment rate of SALIENT framework (ungapped alignment, followed by gapped on determined reads) with minimap2 and Bowtie2. On average, SALIENT

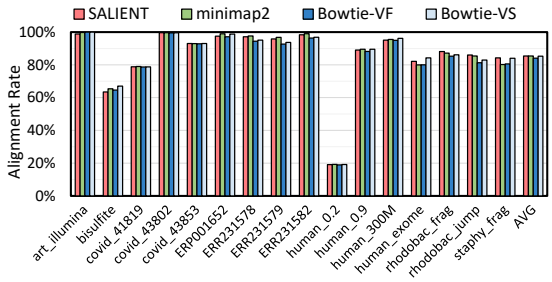

Fig. 15. Comparison of the SALIENT and software tools alignment rate.

TABLE III
ERROR, FALSE NEGATIVE, SENSITIVITY, AND SPECIFITY OF SALIENT

	error	false negative	false positive
human_0.2	0.08%	0.07%	0.01%
human_0.9	0.28%	0.26%	0.02%

obtains an alignment rate of 85.4%, whereas minimap2, Bowtie2-VS, Bowtie2-S, and Bowtie2-VF achieve 85.4%, 85.3%, 84.9%, and 84.1%, respectively. Therefore, SALIENT achieves the same or slightly better accuracy than minimap2 and Bowtie2-VS. Recall that none of the previous ungapped FPGA aligners can achieve such a high alignment rate. Indeed, without gap, the average alignment rate became 73.9% (11.5% drop, see Section III-A), while these works support only two mismatches, which drops the accuracy by an additional 25%. It is noteworthy that among the existing gapped FPGA aligners, [5] does not compare the accuracy with gapped software aligners, and [6] reports 2.8% lower accuracy versus Bowtie2.

False positives can misleadingly increase the alignment rate. In Table III we calculated the false negative, false positive, and error of SALIENT for the human_0.2 and human_0.9 datasets. It can be seen that the rate of false positives is trivial, i.e., the aligned reads by SALIENT are indeed valid. Also the error rate and false negative (i.e., failed to align) rate of SALIENT are indeed better than minimap2 and Bowtie2-VS for the same datasets. E.g., for human_0.9 dataset, the error and false negative rate of minimap2 (Bowtie2-VS) is 0.4% (0.7%) and 0.4% (0.8%), respectively [14].

VI. CONCLUSION

In this paper, we proposed SALIENT, an FPGA-based short read aligner that takes advantage of the fact a significant percentage of aligned reads do not contain gaps. Accordingly, SALIENT decouples the flow into two successive stages of ungapped and gapped alignments to avoid the costly pairwise alignments used in gapped alignment. Then, SALIENT employs a hash-based seed lookup to combat the memory bottleneck of ungapped alignment and achieves a massive ungapped throughput of 845 Mbase/second, over a versatile set of benchmarks. The gapped stage of SALIENT also achieves a high average throughput of 88 Mbp/second, which makes the overall throughput 546 M base/second. SALIENT achieves 40× and 107× higher throughput over minimap2 and Bowtie2 with a similar or slightly better accuracy. SALIENT achieves 9.4–18× higher throughput/Watt compared to the previous ungapped FPGA platforms while supporting higher number of mismatches that leads to significantly higher alignment rate, and 28× higher throughput compared to a state-of-the-art gapped FPGA aligner.

REFERENCES

[1] E. B. Fernandez, J. Villarreal, S. Lonardi, and W. A. Najjar, "Fhast: Fpga-based acceleration of bowtie in hardware," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 12, no. 5, pp. 973–981, 2015.

[2] J. Arram, W. Luk, and P. Jiang, "Ramethy: Reconfigurable acceleration of bisulfite sequence alignment," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 250–259.

[3] J. Arram, T. Kaplan, W. Luk, and P. Jiang, "Leveraging fpgas for accelerating short read alignment," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 14, no. 3, pp. 668–677, 2016.

[4] H.-C. Ng, I. Coleman, S. Liu, and W. Luk, "Reconfigurable acceleration of short read mapping with biological consideration," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 229–239.

[5] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck *et al.*, "Hardware acceleration of short read mapping," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2012, pp. 161–168.

[6] H.-C. Ng, S. Liu, I. Coleman, R. S. Chu, M.-C. Yue, and W. Luk, "Acceleration of short read alignment with runtime reconfiguration," in *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2020, pp. 256–262.

[7] "Illumina dragen bio-it platform 3.7," User Guide, Illumina, Oct 2020.

[8] "Novaseq 6000 sequencing system guide," User Guide, Illumina, September 2020.

[9] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron *et al.*, "Big data: astronomical or genomical?" *PLoS biology*, vol. 13, no. 7, p. e1002195, 2015.

[10] R. F. Schwabe and C. Jobin, "The microbiome and cancer," *Nature Reviews Cancer*, vol. 13, no. 11, pp. 800–812, 2013.

[11] T. Li, D. Liu, Y. Yang, J. Guo, Y. Feng, X. Zhang *et al.*, "Phylogenetic supertree reveals detailed evolution of sars-cov-2," *Scientific reports*, vol. 10, no. 1, pp. 1–9, 2020.

[12] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[13] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature methods*, vol. 9, no. 4, pp. 357–359, 2012.

[14] G. Armstrong, C. Martino, J. Morris, B. Khaleghi, J. Kang, J. DeReus *et al.*, "Swapping metagenomics preprocessing pipeline components offers speed and sensitivity increases," *Msystems*, pp. e01 378–21, 2022.

[15] A. Al Kawam, S. Khatri, and A. Datta, "A survey of software and hardware approaches to performing read alignment in next generation sequencing," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 14, no. 6, pp. 1202–1213, 2016.

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

[17] S. Salamat and T. Rosing, "Fpga acceleration of sequence alignment: A survey," *arXiv preprint arXiv:2002.02394*, 2020.

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

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

[20] T. B. Preuber, O. Knodel, and R. G. Spallek, "Short-read mapping by a systolic custom fpga computation," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2012, pp. 169–176.

[21] J.-Q. Chen, Y. Wu, H. Yang, J. Bergelson, M. Kreitman, and D. Tian, "Variation in the ratio of nucleotide substitution and indel rates across genomes in mammals and bacteria," *Molecular biology and evolution*, vol. 26, no. 7, pp. 1523–1531, 2009.

[22] M. Lin, S. Whitmire, J. Chen, A. Farrel, X. Shi, and J.-t. Guo, "Effects of short indels on protein structure and function in human genomes," *Scientific reports*, vol. 7, no. 1, pp. 1–9, 2017.

[23] K. Koliogeorgi, N. Voss, S. Fytraki, S. Xydis, G. Gaydadjiev, and D. Soudris, "Dataflow acceleration of smith-waterman with traceback for high throughput next generation sequencing," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 74–80.

[24] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[25] "Alveo u280 data center accelerator card data sheet (ds963)," Datasheet, Xilinx, September 2021.

[26] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome biology*, vol. 10, no. 3, pp. 1–10, 2009.

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

[28] W. J. Kent, "Blat—the blast-like alignment tool," *Genome research*, vol. 12, no. 4, pp. 656–664, 2002.

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

[30] "Vitis high-level synthesis user guide," User Guide, Xilinx, December 2021.

[31] N. Moshiri, K. M. Fisch, A. Birmingham, P. DeHoff, G. W. Yeo, K. Jepsen *et al.*, "The vireflow pipeline enables user friendly large scale viral consensus genome reconstruction," *Scientific reports*, vol. 12, no. 1, pp. 1–6, 2022.

[32] Y. S. Ju, J.-I. Kim, S. Kim, D. Hong, H. Park, J.-Y. Shin *et al.*, "Extensive genomic and transcriptional diversity identified through massively parallel dna and rna sequencing of eighteen korean individuals," *Nature genetics*, vol. 43, no. 8, pp. 745–752, 2011.