2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00038

Accelerated Seeding for Genome Sequence Alignment with Enumerated Radix Trees*

Arun Subramaniyan
University of Michigan
Ann Arbor, MI, USA
arunsub@umich.edu

Jack Wadden
University of Michigan
Ann Arbor, MI, USA
wadden@umich.edu

Kush Goliya
University of Michigan
Ann Arbor, MI, USA
kgoliya@umich.edu

Nathan Ozog
University of Michigan
Ann Arbor, MI, USA
ozog@umich.edu

Xiao Wu
University of Michigan
Ann Arbor, MI, USA
lydiaxia@umich.edu

Satish Narayanasamy
University of Michigan
Ann Arbor, MI, USA
nsatish@umich.edu

David Blaauw
University of Michigan
Ann Arbor, MI, USA
blaauw@umich.edu

Reetuparna Das
University of Michigan
Ann Arbor, MI, USA
reetudas@umich.edu

***Abstract*—Read alignment is a time-consuming step in genome sequencing analysis. The most widely used software for read alignment, BWA-MEM, and the recently published faster version BWA-MEM2 are based on the seed-and-extend paradigm for read alignment. The seeding step of read alignment is a major bottleneck contributing ∼40% to the overall execution time of BWA-MEM2 when aligning whole human genome reads from the Platinum Genomes dataset. This is because both BWA-MEM and BWA-MEM2 use a compressed index structure called the FMD-Index, which results in high bandwidth requirements, primarily due to its character-by-character processing of reads. For instance, to seed each read (101 DNA base-pairs stored in 37.8 bytes), the FMD-Index solution in BWA-MEM2 requires ∼68.5 KB of index data.**

We propose a novel indexing data structure named Enumerated Radix Tree (ERT) and design a custom seeding accelerator based on it. ERT improves bandwidth efficiency of BWA-MEM2 by 4.5× while guaranteeing 100% identical output to the original software, and still fitting in 64 GB DRAM. Overall, the proposed seeding accelerator implemented on AWS F1 FPGA (f1.4xlarge) improves seeding throughput of BWA-MEM2 by 3.3×. When combined with seed-extension accelerators, we observe a 2.1× improvement in overall read alignment throughput over BWA-MEM2. The software implementation of ERT is integrated into BWA-MEM2 (ert branch: https://github.com/bwa-mem2/bwa-mem2/tree/ert) and is open sourced for the benefit of the research community.

***Index Terms*—Genomics, Sequence Alignment, Bioinformatics, Computer Architecture.**

I. Introduction

Genomics can transform precision health over the next decade. A genome is essentially a long string of DNA base-pairs (`bp`) `A`, `G`, `C`, and `T` (`3 Giga bp` for a human genome). During primary analysis, a sequencing instrument splits a DNA into *billions* of short (∼100 bp) strings called *reads*. Secondary analysis aligns the reads to a reference genome and determines genetic variants in the analyzed genome compared to the reference. This work focuses on aligning short reads, since more than 70% of the direct-to-consumer (DTC) genomics market is currently serviced by Illumina short read sequencers [3], [4].

This work was supported in part by Precision Health at the University of Michigan, by the Kahn foundation, by the NSF under the CAREER-1652294 award and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

Read alignment is one of the major compute bottlenecks in secondary analysis [5]. Every read needs to be *aligned* to a position in the reference genome. Naively aligning by matching a string to every possible position in the reference genome is computationally intractable. Read aligners solve this using **seeding** [40], [43], [47]. Seeding finds a set of candidate locations (*hits*) in the reference genome where a read can potentially align. Hits for a read are determined by finding exact matches for its substrings (*seeds*) in the reference. The seed-extension phase then uses approximate string matching to select the hit with the best score as the read's alignment position. In addition to read alignment, seeding is also an important kernel in several other sequencing applications: metagenomics classification (e.g., Centrifuge [36]), de-novo assembly [50] and read error correction [28].

Several studies in the past have designed efficient accelerator solutions for seed-extension [13], [18], [25], [26], [32], [51]. However, efficient accelerators for seeding are lacking despite being a performance bottleneck in commonly used read aligners [40], [43], [47]. For instance, seeding contributes ∼40% to the overall run time of state-of-the-art read aligner BWA-MEM2 [47]. We focus on seeding in BWA-MEM2, as it is the fastest available implementation of BWA-MEM [43], widely used as part of the Broad Institute's best practices genomics pipeline [2].

The primary performance bottleneck in seeding is memory bandwidth. This is because both BWA-MEM and BWA-MEM2 use a compressed index structure called the FMD-Index. When compared to BWA-MEM, BWA-MEM2 uses a lower compression factor for the index to reduce memory bandwidth requirements, but because of iterative processing of each base-pair in a read it still has high bandwidth requirements. Our experiments on real whole human genome data show that each short read (with 101 base-pairs or 37.8 B, using 3-bits per bp) requires an average of 68.5 KB of

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00038

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on March 06,2022 at 19:59:37 UTC from IEEE Xplore. Restrictions apply.

data to seed. That is about 50.2 TB of data for the whole genome. Furthermore, each of the index accesses tends to touch a different part of the 10 GB index data-structure, and exhibits little spatial or temporal locality.

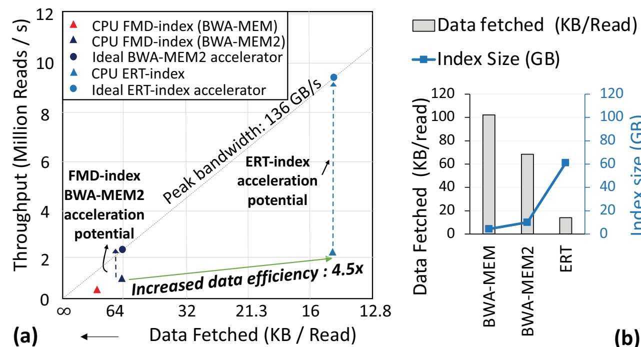

Fig. 1. (a) ERT improves bandwidth efficiency of FMD-Index based seeding allowing for both software and hardware acceleration of BWA-MEM2. Note that the x-axis is scaled linearly based on read / kB as in typical roofline plots. We show kB / read in the figure for ease of explanation. (b) Trade-off between index size and data required for seeding. Experiment performed on Platinum Genome reads ERR194147_1

The memory bandwidth bottleneck can be understood using the roofline plot shown in Figure 1 (a). The roofline is the maximum performance achievable on a system with a given peak bandwidth (e.g., peak memory bandwidth of 136 GB/s), for a given data efficiency (data required per analyzed read) [54]. Based on the data requirements per read, to match the throughput observed on an AWS CPU instance with 72 threads, FMD-index based accelerators for BWA-MEM2 would utilize about half of the peak memory bandwidth (blue triangle). Hence, even an infinitely fast and parallel FMD-index hardware accelerator cannot achieve more than 2.1× speedup over the CPU instance due to its memory bandwidth bottleneck (blue circle), unless the data requirements of the seeding algorithm are reduced and/or the locality of the seeding algorithm is improved. Existing hardware accelerators for seeding [17], [20], [23] based on the FMD-Index are thus all limited by this upper limit. To address these challenges, this paper presents a novel data structure with 4.5× higher data efficiency than BWA-MEM2 and an accompanying custom accelerator architecture for seeding.

Data Structure Innovation. The highly compressed FMD-index in BWA-MEM (4.3 GB for human genome) trades off high memory bandwidth for small memory space. In contrast, we propose a novel data structure for seeding that makes the opposite trade-off: it trades off increased memory space for reducing bandwidth required, while still fitting within a modern server's main memory (64 GB) as shown in Figure 1 (b). This design tradeoff is similar in spirit to that made in BWA-MEM2 from Broad Institute and Intel (which uses a lower compression factor for the FMD-index, resulting in a 10 GB index), but our solution further improves bandwidth efficiency by virtue of supporting multi-character lookup and exploiting re-use opportunities present in the seeding algorithm. We refer to our bandwidth-efficient data structure as Enumerated Radix Trees (ERT).

Like the FMD-index, ERT enables variable length exact match search functionality. But, unlike the FMD-index, it avoids iterative lookup for every base-pair in a large structure. It achieves this by coalescing all substrings in a reference genome that start with the same k-mer (string of length k, where k is less than the minimum length for a seed) together, and representing them using a variant of a radix tree. As we discuss later, ERT allows *multiple consecutive base-pairs to be matched with one lookup*, and exhibits better spatial locality than the FMD-index. ERT also helps *reduce computation when substrings* within a read that need to be matched with the reference *overlap* using a prefix encoded radix tree. ERT increases data efficiency (data fetched per read) by 4.5× as shown in Figure 1 (a). While it increases memory space requirement to 62.1 GB (Figure 1 (b)), it can still fit well within the main memory of modern servers.

Seeding Accelerator. While seeding is inherently a memory-bound algorithm, a CPU implementation is still compute bound because of the inefficiencies of general purpose processing. GPUs are not well-suited because of significant memory divergence during tree traversal. We observe that ERT's increase in bandwidth efficiency unlocks significant acceleration potential as shown in roofline Figure 1 (a). An ideal ERT based accelerator could achieve a speedup of ∼10× over BWA-MEM2 seeding. To shift the problem to be more memory-bound and exploit this acceleration potential, we design a custom seeding accelerator. The seeding accelerator leverages a butterfly network to efficiently feed data to parallel specialized seeding processors. Each seeding processor leverages light-weight context switching to provide high compute density and hide the long latency of DRAM accesses.

We also observe that a radix tree of a k-mer is frequently re-used across multiple reads. However, typically several radix trees need to be accessed to find seeds for a read, and their aggregate size exceeds that of on-chip caches. As a result, a radix tree usually gets evicted before it can be re-used, resulting in a low hit rate in traditional caches. To expose this latent temporal locality, we instead design our accelerator to partition seeding into two phases. The first phase discovers all the reads that need a radix tree (during so-called forward extension). Then, the second phase fetches a tree once, and processes all the reads that need it (for so-called backward extension), thereby increasing ERT re-use by 2×.

Binary equivalent acceleration and FPGA demonstration. Given its clinical relevance, we pay particular attention to guaranteeing exact output as the original BWA-MEM2 software. ERT-based seeding is bit equivalent and fully verified.

In summary, this paper makes the following contributions:

- We propose a novel seeding data structure called Enumerated Radix Trees (ERT) that trades off increased memory footprint for reducing memory bandwidth required for seeding while still fitting within a modern server's main memory (64 GB).

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on March 06,2022 at 19:59:37 UTC from IEEE Xplore. Restrictions apply.

- We design a seeding accelerator to fully exploit the improved data-efficiency provided by ERT. Our seeding accelerator uses specialized seeding processors that support fine-grained context switching to hide the long latency of DRAM accesses while providing high compute density and improving memory bandwidth utilization.
- We evaluate ERT on the GRCh38 human genome assembly with 787,265,109 Illumina Platinum Genomes 101 bp single-ended reads. A software implementation of ERT provides a 2.1× speedup over state-of-the-art BWA-MEM2 seeding [47]. Our FPGA prototype of ERT implemented on AWS F1 FPGA (f1.4xlarge) achieves 3.6 million reads/s, resulting in 3.3× higher seeding throughput over 72-thread software BWA-MEM2. When combined with seed-extension accelerators, we observe a 2.1× improvement in overall read alignment throughput over BWA-MEM2.
- We open source the ERT software implementation for the benefit of the research community. ERT-based seeding is also integrated into BWA-MEM2 (*ert* branch: https://github.com/bwa-mem2/bwa-mem2/tree/ert).

II. Background and Motivation

Both the seeding and seed-extension steps of read alignment are important candidates for acceleration. Seeding contributes 40% to the overall execution time of BWA-MEM2 [1] and seed-extension contributes 35%. The remaining steps include chaining that groups nearby seeds in the reference (10%), output creation (7%) and memory allocation overheads (8%).

A. Seeding Algorithm in BWA-MEM2

The seeding algorithm in BWA-MEM2 is based on identifying substrings in the read that have super-maximal exact matches (SMEMs) with the reference genome [43] (Figure 2 (a)). A maximal exact match (MEM) is an exact match between the read and the reference that cannot be extended in either direction without encountering a mismatch. An SMEM is a maximal length match (MEM) that is not *fully contained* in any other MEM. Short matches lead to an excessive number of hits to be verified by seed extensions, while longer matches can lead to incorrect alignments. BWA-MEM only reports SMEMs greater than a certain minimum length (e.g., 19), empirically determined to be a good trade-off between performance and accuracy.

Figure 2 (b) shows the two steps involved in determining SMEMs for a sample read and reference pair.

(1) ***Forward search:*** For a given query position in the read (e.g., pivot x_0 in Figure 2), subsequent base pairs to its right are looked up one at a time in a reference index (e.g., FMD-index) to find the longest exact match in the forward direction. The end position of the longest match becomes the next pivot. During this step, all the positions in the read where there is a *change* in the set of candidate reference locations (hits) are marked (left extension points (LEP) in Figure 2 (b). for

[1] measured on the whole human genome dataset (ERR194147_1) consisting of 787,265,109 reads of 101 bp length.

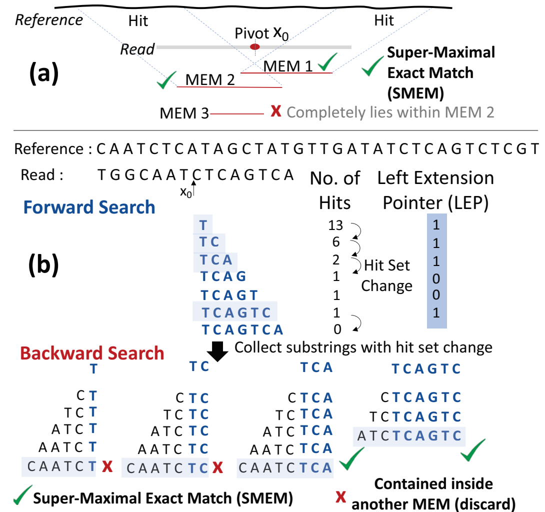

Fig. 2. (a) Super-Maximal Exact Matches example. (b) Forward and backward search to identify super-maximal exact matches (SMEMs).

substrings T, TC, TCA and TCAGTC). Only these positions are used as the starting query positions to identify MEMs that extend in the backward direction. Other positions are guaranteed to produce MEMs that are contained within those identified from LEP.

(2) ***Backward search:*** For each query position identified in the previous forward search step as part of LEP (substrings T, TC, TCA and TCAGTC), subsequent base pairs to its left are looked up one at a time to find the longest exact match in the backward direction. After this process, SMEMs are identified by discarding MEMs fully contained in other longer matches. In Figure 2 (b), CAATCTCA and ATCTCAGTC are reported as SMEMs. The MEMs CAATCT and CAATCTC are discarded because they are fully contained in another MEM CAATCTCA. SMEMs obtained during seeding are assumed to be part of the final alignment.

B. FMD-Index

To identify SMEMs and their locations in the reference genome, BWA-MEM2 uses a highly compressed data structure called the FMD-index [24], [42] which is built using both strands of DNA. As shown in Figure 3 (a), the FMD-index consists of: (1) the *suffix array (SA)*, which contains the locations of lexicographically sorted suffixes of the reference genome R, (2) the *Burrows Wheeler Transform (BWT)*, computed as the last column of the cyclically sorted suffix array of the reference, (3) the *count table* (C) which stores the number of characters in R lexicographically smaller than a given character c and (4) the *occurrence table (Occ)* which stores the number of occurrences of a character up to a certain index in the suffix array. Using the count and occurrence tables, one can identify intervals (s and e) in the Burrows-Wheeler matrix where a particular query string exists in the reference by performing iterative lookups for each successive character in the query as shown in Figure 3 (b).

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on March 06,2022 at 19:59:37 UTC from IEEE Xplore. Restrictions apply.

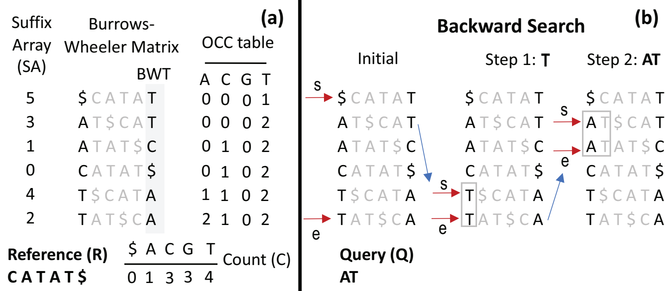

Fig. 3. (a) FM-index for reference R (b) Backward search using the FM-index.

C. FMD-index based seeding bottlenecks

The FMD-index allows the lookup of a query `Q` of length `N` in a reference `R` using approximately $\mathcal{O}(N)$ memory operations. The BWA-MEM2 implementation of FMD-index uses 10 GB (6 GB occurrence table (8× compressed) + 4 GB suffix array (8× compressed)) [47] compared to 4.3 GB in BWA-MEM. Further decompressing the occurrence table and suffix array of BWA-MEM2 is shown to only improve performance slightly: 3–7% [47].

Starting from a single character in the read, the FMD-index enables forward and backward MEM searches to determine the number of hits of progressively longer substrings using at most 2 extra memory lookups per character. However, these memory lookups touch different parts of a 10 GB data structure and rarely exhibit spatial locality. This reduces the effectiveness of caching in modern processors and leads to high memory bandwidth requirements. Software implementations of the FMD-index (e.g., BWA-MEM) have attempted to improve the locality of MEM search in two ways. *First*, occurrence table entries are typically co-located with portions of the BWT in tightly packed cache-line aligned data structures to improve the spatial locality of an index lookup. *Second*, backward search passes for substrings sharing the same prefix (e.g., `TC` and `TCA` in Figure 2 (b)) are performed in lock-step leading to access of occurrence table data belonging to the same or nearby cache lines [43], [56]. Despite these optimizations, our experiments on real whole human genome reads (details in Section V) show that FMD-index based seeding still has high data requirements (i.e., each read can require ∼68.5 KB of index data for seeding). Further, ∼40% cycles are spent in core stalling for memory/cache.

FMD-Index based seeding also inherently involves sequential dependent memory accesses and its performance is limited by memory access latency. We mitigate this problem using hardware multiplexing, where one physical compute unit context switches between different reads on a memory stall.

III. Enumerated Radix Trees

A. ERT Design

1) K-mer Enumerated Index:

FMD-index stores a compressed representation of the set of *all suffixes that exist in the reference genome* in lexicographic order. We now consider a substring of length k in the read (referred to as a *k-mer*). Due to natural genome variation and sequencing errors, not all k-mers will exist in the reference and, hence, in the FMD-index. Therefore, when looking up a k-mer in the FMD-index, we must start with a 1-mer and grow the string, character by character, for as long as it exists in the FMD-index, or till we reach the desired k-mer length. This iterative, character-by-character access to the FMD-index substantially increases the required number of DRAM accesses, creating a memory bottleneck. This is further aggravated by the fact that accesses to the index rarely follow lexicographic order, making it difficult to exploit locality over such a large window (i.e., set of all suffixes of the k-mer).

To overcome these two limitations, we instead enumerate *all possible k-mers* (whether they exist in the reference or not) and store them in an index table. For each k-mer (an index entry), we then store all its suffixes in the reference. Since all possible k-mers are represented in the index, k *characters* from the read can be looked up in a single memory access, significantly reducing the number of DRAM accesses. Furthermore, subsequent accesses to the suffixes of the k-mer have much improved spatial locality, since they are co-located together. LEP information for the k-mer, resulting from each of the *k* single character lookups is pre-computed and stored in the index table entry. Figure 4 shows an example index table enumerating all 6-character substrings.

To **choose k**, we observe that BWA-MEM2 only reports SMEMs greater than a certain minimum length (e.g., 19). This is because shorter substrings lead to an excessive number of hits to be verified by seed extensions. Thus `k` can be set to any value less than 19. The higher we set it, the more characters can be looked up at once, but it would require more space. We choose $k = 15$ to keep the size of index table tractable ($O(4^k)$), i.e, 1 G entries when $k = 15$. Later, in section III-E we discuss a solution to effectively increase k by selectively using a multi-level index.

2) Customized Radix Tree:

The next question is how to store the suffixes of a k-mer in an index entry, so that we can support MEM searches for strings longer than `k`. One option is to augment the index table with an FMD-index, and iteratively grow the k-mer prefix. However, even within the subset of all suffixes sharing the same k-mer prefix, FMD-index lookups have poor locality. Also, they still operate with a single character at a time.

To overcome this problem, we observe that a radix tree can naturally support *multi-character lookups*. This is because in a radix tree, we can merge all singleton paths into a single node, thereby addressing a multiple character lookup with a single memory access. Figure 4 shows a radix tree for one k-mer `GACAGC` in the index table (note radix is 4 for the genome alphabet). The proposed ERT merges singleton paths (`GC` in Figure 4) using variable-size internal nodes that store the full singleton path string (designated as `UNIFORM`). A singleton path is encountered when all paths in the tree from a certain node onward share a common string.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on March 06,2022 at 19:59:37 UTC from IEEE Xplore. Restrictions apply.

Fig. 4. Enumerated radix tree (ERT) example. ERT supports multi-character lookup using a multi-level index table and radix tree. Each index table entry contains a `Type` field, k-1 bit `LEP` field (Section II) and a `pointer` field that indicates the address of the root node of a radix tree (not shown). `Type` can be any of the following: (1) `EMPTY`: k-mer is absent in the reference. (2) `LEAF`: k-mer is unique (i.e., has the same suffix at all occurrences in the reference). (3) `TREE`: k-mer exists in the reference and has a pointer to the root of the radix tree. (4) `TABLE`: k-mer has a large number (>256) of unique suffix strings in the reference. The index table entry for this k-mer points to a 2nd-level index table to succinctly represent these suffix strings. ERT represents a singleton path using a variable-size internal node (`UNIFORM`) supporting multi-character lookup (5). If a singleton path ends in a leaf, it is truncated at its start with a `LEAF` node that points to the reference genome (path compression), 6). Each path from the root to the leaf in ERT encodes a prefix of a sequence occurring in the reference genome. To indicate absence of prefixes in the reference, ERT also includes `EMPTY` nodes (ending with $). `UNIFORM` nodes have only one valid child branch for the prefix, while `DIVERGE` nodes have more than one.

Early Path Compression: To further improve the space-efficiency of the ERT, we observe that a k-mer frequently becomes *unique* in the reference genome as it increases in length. This means that, past a certain length, a prefix is followed by a single, unique suffix string in the reference genome. This would introduce a `UNIFORM` node in the ERT with a singleton string of characters (up to the length of the read). To avoid storing this long string, we instead replace it with a pointer to the occurrence of this string in the reference genome. In Figure 4 we show how in the ERT, these nodes are marked as *leaf nodes*, containing a single pointer. Leaf nodes encountered during a MEM search are decompressed, by fetching the full reference string corresponding to the reference pointer stored at the leaf node. Note that the pointer in the leaf node is required regardless of this compression technique since it is necessary to indicate the location of the traversed k-mer in the reference genome. Hence, it does not present any storage overhead. Instead, this optimization results in $\sim 2\times$ space savings and was critical for being able to store the full human genome in under 64 GB of storage, which is a common configuration for servers.

3) ERT Construction and SMEM Operation:

The ERT `k-mer` index table and corresponding radix trees are built by first enumerating all possible k-mers and then querying a pre-built FMD-index of the reference genome to grow the trees for each k-mer according to all existing sequences in the reference. Each k-mer and ERT path from the root to the leaf of the tree corresponds to a unique sequence in the reference. The locations of these sequences are stored as pointers at the leaves of the tree, as noted above. Note that if a particular k-mer does not exist (referred to as `EMPTY` in Figure 4), we do not store a pointer to an ERT tree since no SMEM with length $k < 19$ is required. In our implementation where $k = 15$, 38.8% of the index entries are empty. For an `EMPTY` entry, we still compute the LEP bits corresponding to the k-mer and store it in the index table, to indicate at which positions along this k-mer a backward traversal must be initiated.

The size of the ERT index depends both on the size and the repetitiveness of the reference genome under study. We empirically estimate the space occupied by the ERT index to be $\sim$20 N bytes, where N is the size of the reference genome in Giga base-pairs. For instance, the ERT index size for the human genome is 62.1 GB (index table–8 GB; radix tree–54.1 GB) and for the wheat genome ($\sim$17 Giga bp) is 320 GB. ERT index construction is fast and takes $\sim$1 hr wall-clock time for the human genome with 72-threads. Note that ERT index construction is not a bottleneck, since it is done only once per reference genome and reused across several read alignment runs.

Once constructed, we can use the ERT to search for MEMs according to the SMEM seeding algorithm (Section II-A). For a given `k-mer` scanned from a read, based on the starting position of the MEM, we do the following: **(1)** The index table is looked up using the `k-mer` with a single DRAM access (❶–❹ in Figure 4). If an entry in the index table exists, the root of the `k-mer` tree is also fetched with a second memory access (❺). **(2)** The branches in the `k-mer` tree are then traversed according to the remaining base pairs in the read

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on March 06,2022 at 19:59:37 UTC from IEEE Xplore. Restrictions apply.

until a leaf node is encountered or an empty node is reached (i.e., no further characters in the read match with strings existing in the reference). **(3)** If a leaf node is reached, the reference sequence corresponding to that leaf is fetched with a DRAM access to determine the final characters matching with the read (❻). **(4)** If we reach a dead end (EMPTY node) in the tree, we have found the end of the MEM. At this point, all locations where this MEM exists in the reference (i.e., all leaf nodes in the downstream sub-tree) are gathered using a depth-first traversal, referred to as *leaf gathering*. **(5)** The index table entry only contains the LEP bits for the k-mer. We compute the LEP bits for the suffix of the k-mer as follows. Each time a DIVERGE node is encountered during the traversal of an ERT path, a bit is set in the LEP bitvector since this indicates a *hit set* change, i.e., the hits are divided across the divergent paths from that node. **(6)** After the depth-first sub-tree traversal completes or in case the end of the read is reached, a backward traversal is initiated for each LEP position along the traversed path. The backward traversal operates in the same way as the forward traversal by searching the same ERT data structure for MEMs in the reverse complemented read. Note that bases A and T and bases C and G are complements of each other.

B. Optimization: Prefix-Merged Radix Tree

The goal of prefix-merged radix trees is to re-use work across MEM searches (forward or backward) from consecutive positions in the read.

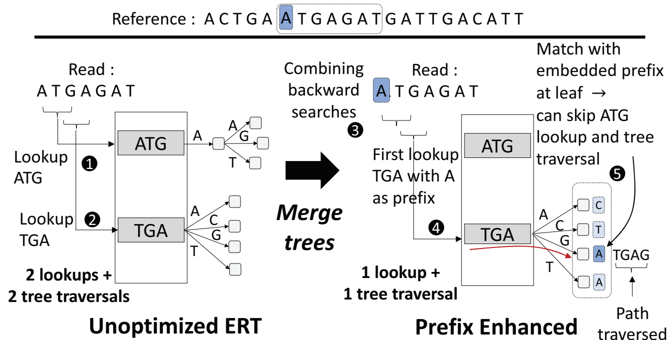

Fig. 5. Merging radix-trees by adding prefix data at the leaf nodes allows ERT to leverage prefix information to perform multiple MEM searches in a single tree traversal.

Opportunity: In the seeding computation, the time spent doing backward MEM searches is $\sim 2\times$ that of forward search, making it important to optimize this step. On average, we find that there are $\sim$10 backward searches for each forward search from a pivot. Also, it is common to observe searches from adjacent query positions in the read (consecutive bits of LEP are '1'). Normally, these lead to multiple independent index table lookups and tree traversals as shown in Figure 5.

Insight: In the unoptimized ERT, there exists a radix tree for each k-mer that occurs in the reference, including adjacent, sliding window k-mers (e.g., ATG and TGA). We recognize that radix trees for adjacent k-mers contain redundant information and that the information contained in one of the trees can be reconstructed from the adjacent k-mer's tree by storing prefix information at each of its nodes. In the example shown in Figure 5, a string ATGA, which is normally found by accessing the ATG tree can be instead reconstructed from the TGA tree by indicating the presence or absence of prefix character A in each of the nodes of TGA's tree.

The key observation is that with such a prefix-merged radix tree, multiple backward searches (TGAxyz and ATGAxyz) can be performed in a *single index table lookup and tree traversal* by checking for prefix character matches at each visited node. In Figure 5, when we reach the leaf node represented by string TGAG, we can also match character A from the read as a prefix, resulting in the MEM represented as ATGAG. This reduces two MEM searches into one.

Design: Augmenting each of the nodes with prefix information in order to merge k-mer trees takes up significant space and offsets the benefit from merging trees. Therefore, in our prefix optimized ERT, only leaf nodes are augmented with prefix characters (2 bits per prefix character) found at the corresponding reference positions (Figure 5). Storing prefix information at the leaf nodes is sufficient as prefix information at each of the internal nodes can be reconstructed by visiting all of the leaf nodes in its corresponding sub-tree. We chose a 1-character prefix at leaf nodes after observing that each backward search on average matches $\sim$1 prefix character at the leaf nodes. This resulted in 50% fewer backward searches. The index table entry for k-mers with merged radix trees (e.g., ATG) contains a pointer to the adjacent k-mer's tree (e.g., TGA, not shown in figure). We also add a *tree-present* bit to each k-mer entry to distinguish k-mers with/without merged radix trees.

C. Optimization: Locality with K-mer Reuse

The goal is to increase the re-use of the index table entry and the radix tree for a k-mer.

Opportunity: For a batch of 1000 reads, we observe that $\sim$45% index table and radix tree accesses from k-mers can be re-used, with re-use improving slightly with larger batch sizes. This is expected given the highly redundant nature of the human genome and high coverage of sequenced reads needed to correct sequencing errors (each position in the reference genome can be covered by 30–50 reads on average). However, typically several radix trees need to be accessed to find seeds for a read, and their aggregate size exceeds that of on-chip caches. As a result, a radix tree usually gets evicted before it can be re-used, resulting in a low hit rate in traditional caches. This problem can be mitigated if we can determine in advance the set of k-mers for which a radix tree needs to be fetched from DRAM.

Insight: The forward and backward search phases of the SMEM algorithm need not be performed sequentially for each read. Instead they can be decoupled to expose temporal locality. More specifically, we can perform forward search for a batch of reads, identify all the unique k-mers that are to be used in backward search (using LEPs), fetch each radix tree once for each unique k-mer and perform all backward searches

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on March 06,2022 at 19:59:37 UTC from IEEE Xplore. Restrictions apply.

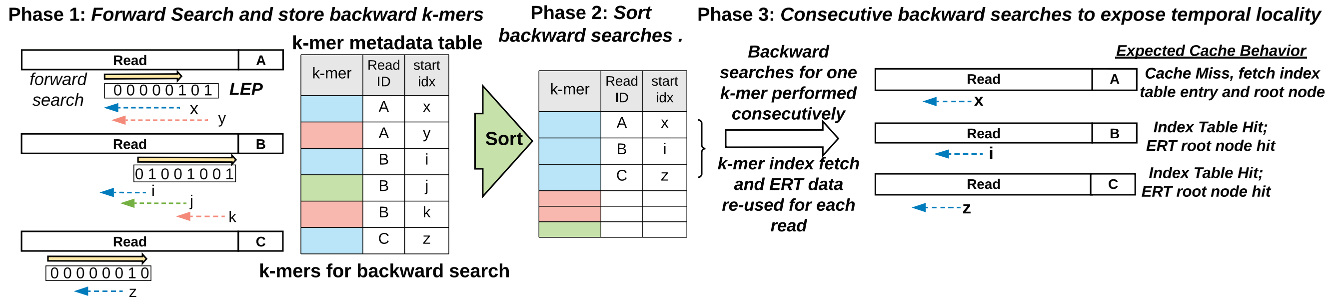

Fig. 6. **Phase 1)** Perform forward extension for a batch of reads. Identify all k-mers required for backward extension (dashed lines) using LEP. Store k-mer metadata (`k-mer, read ID, start index`) in a metadata table. **Phase 2)** Sort k-mers to bring backward extension tasks involving the same k-mer together. **Phase 3)** Perform all backward extensions involving the same k-mer together to exploit locality. The ERT for each k-mer is fetched only once from DRAM, reducing bandwidth requirements.

for that k-mer tree before moving to the next k-mer. We refer to this technique as *k-mer reuse*.

Design: Figure 6 describes the steps to be performed to leverage k-mer reuse. While processing the forward searches for a batch of N reads, we store each backward search that must be computed in a k-mer metadata table implemented on-chip. Each backward search entry is composed of: **(1)** k-mer starting from the backward search point in the read, **(2)** the read ID in the batch, and **(3)** start position of backward search in the read. Once all forward searches have been completed for a batch of reads, we sort all entries in the metadata table, grouping each required backward search by k-mer. We then proceed one k-mer at a time and compute all backward searches associated with a k-mer sequentially. The first time a k-mer is encountered, we perform one index table lookup, as well as fetch a portion of the k-mer's tree into an on-chip cache. Subsequent backward extensions then consult this cache during tree walking, skipping two otherwise mandatory DRAM accesses. We find that forward, backward and sort phases of seeding take 26.4%, 67.6% and 6% time respectively.

D. Optimization: Tiled Layout for Spatial Locality

Similar to [19], [35], the spatial locality of ERT accesses can be improved by using a tiled layout for radix tree nodes as shown in Figure 7. In this layout, sub-trees of nodes that are likely to be accessed at the same time are clustered together into a single cache block- or a DRAM page-sized tile. Compared to breadth-first or depth-first layout of nodes, the tiled layout guarantees at least $log_4(n+1)$ nodes accesses per tile, where n is the number of nodes in the tile. With this optimization, ERT traverses ~3 nodes on average per 64 B, utilizing 50% of the data it fetches from memory.

E. Optimization: Memory Space

Opportunity: Enumerating all k-character prefixes in the index table can have prohibitive space overheads for large k. For example, a 19-mer table has 4^{19} entries, resulting in 2 TB of space, assuming 8 bytes per entry. However, the human genome is not a random string of characters from the genome

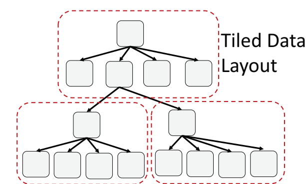

Fig. 7. Cache-friendly tiled data-layout for ERT.

alphabet. The repetitive nature of the human genome makes the distribution of hits (or leaf nodes in the radix tree) for different k-mers heavily skewed.

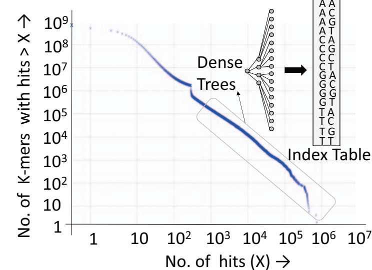

Fig. 8. Figure showing skewed hit distribution for k-mers.

Insight: We leverage the skewed distribution of k-mers in the human genome to design a multi-level index table. For a given number of hits X, Figure 8 shows the number of k-mers in the human genome that have hits > X. It can be seen that very few k-mers (~ 0.01%) have greater than 1000 hits. However, these k-mers have dense radix trees, which can be compactly represented using an index table as shown in Figure 8.

Design: Instead of enumerating all k-character prefixes for large k, we decompose the index table into two levels (Figure 4 ❼), wherein the first level enumerates all k-mers and the subsequent level enumerates all x-character suffixes for a subset of k-mers (such that $k + x$ = min. SMEM length). The multi-level index table further extends the benefit of multi-character lookup. While choosing a larger x helps reduce tree traversal time, for the human genome we were able to

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on March 06,2022 at 19:59:37 UTC from IEEE Xplore. Restrictions apply.

accommodate up to x = 4 (fan-out = 256) for a subset of 15-mer dense trees without increasing space overheads. Compared to x = 1, x = 4 improves CPU performance by 10%. Since most trees are shallow (83% of leaf nodes have depths $<=$ 8), we did not explore more than two-levels or higher fan-out for the internal nodes of ERT.

F. Optimization: Pruning Wasteful Backward Searches

Typically backward search is performed starting from each query position where the set of candidate hits changes (as given by the LEPs), in no particular order. However by imposing an order for the backward searches, namely starting from the rightmost query position where the hit set changes and proceeding leftward, it is possible to prune out subsequent backward searches as illustrated in Figure 9.

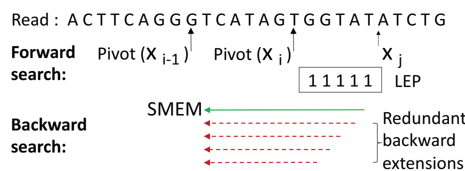

Fig. 9. Pruning wasteful backward searches by performing backward searches in right-to-left order

The forward pass partitions the read into multiple non-overlapping MEMs. As a result, each backward search is guaranteed to not produce a MEM that spans across multiple pivots. If any backward search from position x_j in the read reaches the previous pivot x_{i-1}, then backward searches $\forall x$, where $x < x_j$ are guaranteed to produce MEMs that are contained within that of x_j and are redundant.

IV. Seeding Accelerator

A. Overview

The overall architecture of our seeding accelerator is shown in Figure 10. The accelerator is composed of multiple parallel seeding machines connected to the available DRAM channels using a crossbar network. Each seeding machine is composed of a control processor that issues commands to three types of processing elements. Each processing element (PE) is provisioned with multiple lightweight contexts and performs a sub-task associated with SMEM identification (i.e. index table lookups, walking ERTs, and depth-first search based leaf gathering). When a processing element issues a memory request to the Data Fetcher–a rudimentary address generation unit and memory controller– and a memory stall occurs, the processing element immediately switches to a new context. This fine-grained context switching greatly increases compute density of each seeding machine and is essential to an FPGA implementation with limited logic and routing resources. When the memory request returns, its data is stored in the corresponding PEs context memory and the context is marked as ready.

Decoding radix-tree nodes to determine the next node while traversing the ERT and control operations in the SMEM algorithm are the most time consuming compute steps in ERT-based seeding. Prior to designing custom functional units for these steps, we explored the RISC-based Xilinx MicroBlaze softcore. However, on the MicroBlaze, node decoding resulted in 10–16× higher latency based on ERT node type and required 1.7× higher LUTs and 3.2× higher flip-flops compared to a custom node decoder. When the custom node decoder is combined with a MicroBlaze-based controller, we observed 7.3×–16.6× higher latency for implementing the SMEM algorithm compared to a custom node decoder coupled with a custom controller implementation.

B. Processing Elements

Index Fetcher: The Index Fetcher is responsible for initiating a walk by converting a k-mer string to an index table address and requesting the corresponding entry from the ERT index table. These requests immediately trigger a context switch, swapping out the current context until the requested data is returned. If the path terminates at the index table (entry type = `EMPTY`), the results are returned to the control processor to determine how to proceed. If the radix tree for that k-mer exists, the index fetcher issues a request for the root of the radix tree.

Tree Walker: The Tree Walker is responsible for traversing the ERT, decoding nodes, and reporting the end result of a walk. Each node in the tree is decoded using the corresponding base-pair in the read to calculate the next ERT node address. If the Tree Walker ever detects that it needs more of the ERT data structure to continue its traversal, it requests the data from the Data Fetcher and triggers a context switch. During decode, the Tree Walker computes the address of the next tree node based on the types and content of existing child nodes and the read characters or ends the traversal. Each ERT node takes a variable number of cycles to decode depending on node complexity. For example, `UNIFORM` nodes require an exact match string comparison to compare each DNA base-pair in the `UNIFORM` string with the read string. This comparison is accomplished using parallel XOR gates and priority encoders over three cycles. Leaf nodes that are "early path compressed" also require string comparison hardware (Section III-A2). Implementing these comparisons using custom parallel hardware is an important feature of the specialized processor versus implementation in software on a general purpose CPU.

Leaf Gatherer: If a tree walk hits an `EMPTY` node (i.e., match cannot be extended further) all remaining leaves in the parent sub-tree must be gathered in order to identify all possible reference locations of the current match. We refer to this as *Leaf Gathering*, and accomplish it using depth-first search (DFS) on the ERT sub-tree. This DFS is accomplished by considering and decoding each base-pair (A,T,G,C) path in the ERT and maintaining a stack of ERT node indices that need to be explored. Nodes are decoded and traversed just as in the Tree Walker, however, the Leaf Gatherer does not need to perform string matching (required for early path compression and `UNIFORM` nodes), and does not include string comparison hardware.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on March 06,2022 at 19:59:37 UTC from IEEE Xplore. Restrictions apply.

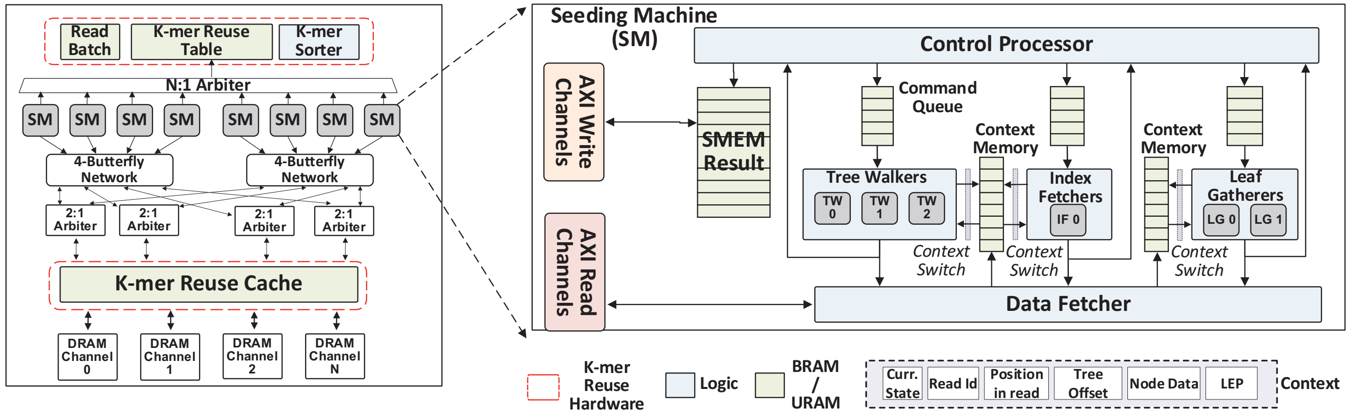

Fig. 10. **Seeding Accelerator architecture.** Each Tree Walker (TW) is responsible for scanning a read, walking ERT Trees, and computing candidate SMEMs. Each Tree Walker can switch between multiple contexts to help hide memory latency. The Data Fetcher (DF) is responsible for serving ERT and reference fetch requests to DRAM. The Control Processor (CP) coordinates read fetch, and k-mer reuse phases.

C. *Control Processor*

The SMEM search algorithm consists of several input-dependent conditional branches that are hard to predict in general-purpose processors. Our control processor overcomes this by implementing the high-level algorithm for SMEM search in hardware. For example, if a forward walk finishes, the control processor looks at the start and end point, determines the condition of the finished walk, and issues a new command (e.g., get the leaves associated with the walk if the walk produced an SMEM, or start a new backward search if the walk failed to produce an SMEM) to the corresponding processing element command queue. To simplify tree walking hardware, walker PEs do not have separate hardware for forward or backward walks; the control processor issues a forward or backward walk command by providing a start index and the forward read (for forward searches) or reverse complemented read (for backward searches). The control processor maintains a queue of pending tree walks to deal with variable tree traversal times and schedules walks from other reads to ensure good compute utilization. Our accelerator is designed to be flexible enough to also implement other algorithms based on the FMD-index. This would require adding new control FSMs to the Control Processor, while all other hardware structures (index fetchers, tree walkers, leaf gatherers, crossbar, and I/O) can be reused.

D. *K-mer Reuse Metadata Storage and Sorting*

In order to perform k-mer reuse (Section III-C, Figure 6), all backward search LEPs for a forward match in a read must be exported to the k-mer metadata table. Backward searches that share the same k-mer are grouped together using the parallel hardware sorter [49] to group entries for the same k-mer (Phase 2 in Figure 6). We also implement a specially designed cache structure–*the k-mer reuse cache*–to cache index table lookups, ERT root node accesses, and other ERT accesses. It is sized conservatively using high-coverage human reads (details in Section V, Table III, batch size = 1000), taking into account potential high-reuse use-cases e.g., high coverage input reads and reads from other repetitive genomes like wheat. We saw little reuse benefit from increasing batch size beyond 1000 and reuse cache size beyond 4 MB. The k-mer reuse cache is also direct-mapped. We settled on a direct mapped cache, since the observed hit rate was within 1.2% of a fully-associative cache. Since k-mer reuse forces the algorithm to generate MEMs out-of-order for a particular read, we must also store all MEMs for each read in intermediate on-chip storage, to perform MEM containment checks and finally produce SMEMs in a final reconciliation step.

E. *System Integration and Programming API*

```
/* Create batch of 2-bit encoded reads in host */
void encode(uint8_t* h_buf, char** reads, int* len);
/* PCIe DMA transfer of 'sz' bytes from host buffer
   to accelerator memory at offset d_off */
int writeData(uint8_t* h_buf, int sz, int d_off);
/* Write configuration register
   on accelerator to begin processing batch */
int startCompute();
/* Read status register
   on accelerator to determine completion */
int waitForFinish();
/* Read 'sz' bytes from offset d_off
   in accelerator memory to host buffer o_buf */
int readResult(uint8_t* o_buf, int d_off, int sz);
```

Listing 1. C Programming API for accelerator

Host-Accelerator Interface: We adopt a system configuration similar to that in AWS EC2 F1, with both the host and the accelerator having their own physical memories (DRAM). We assume the existence of two communication channels from the host to the accelerator, similar to the AWS-F1 shell interface; one for data transfer to/from the accelerator custom logic (CL), for example, using 512-bit AXI-4 DMA transactions (XDMA) on PCIe Gen3 $\times16$ links; and another for issuing control commands and accessing memory-mapped status registers using an interface such as the 32-bit AXI4-Lite interface (OCL).

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on March 06,2022 at 19:59:37 UTC from IEEE Xplore. Restrictions apply.

Listing 1 shows the C API for programming our accelerator which builds on the AWS FPGA management libraries. The libraries include APIs for reading/writing large chunks of data to the accelerator via PCIe DMA and handling interrupts. Memory-mapped registers on the accelerator are used for configuration and status monitoring. We also implement overflow handling in the accelerator for reads with too many SMEMs that overflow the on-chip MEM result buffers. Our accelerator flushes these results into a designated overflow region in the accelerator DRAM to be later processed in the host.

Runtime System: We extend the multi-threading model in BWA-MEM to provide a separate worker thread, one each for managing our seeding and optional seed extension accelerators interfaced over PCIe. These worker threads communicate via non-blocking producer-consumer queues.

The main CPU thread first allocates a buffer for the ERT-index and copies it to the accelerators' DRAM using XDMA transactions. Reads are then pre-processed in a worker-CPU thread which allocates a 64-byte buffer for each read, encodes each base using 2-bits (reads with ambiguous bases such as `N` are processed on the host) and copies the buffer to the accelerators DRAM. This thread also acquires a lock to one of the accelerator status registers using the OCL interface and signals the accelerator to begin computation. It continues to monitor the status of this register, till the accelerator updates it with a *done_seeding* command. At this point, another CPU thread retrieves SMEMs from the accelerator DRAM using XDMA transactions and processes any result-buffer overflows. These SMEMs pass through a chaining step and can optionally be processed similarly using a seed-extension accelerator.

We implement double buffering on our accelerator, so that memory transfers to/from the accelerator over PCIe can be overlapped with computation. Seeding results are encoded in the same format as the baseline prior to chaining (i.e., (seed start position in read, seed length, list of seed hits in the reference genome)) to eliminate overheads due to additional data structures for format conversion.

V. Methodology

Reference genome and input reads: ERT was built using the latest build of the reference human genome assembly (GRCh38) from the UCSC genome browser [33]. Decoy contigs and mitochondrial DNA are filtered out and only chromosomes 1-22, X and Y are used to build the ERT index. For the input reads, we choose the single-ended Illumina Platinum Genomes benchmark dataset (`ERR194147_1.fastq`) [22] consisting of 787,265,109 reads of 101 `bp` length also used in prior work [25]. Reads containing ambiguous base pairs (non-A/C/G/T) are processed on the host-CPU and ambiguous base pairs in the reference genome are converted to one of the standard nucleotides (A/C/G/T) using the same procedure as [43], [44].

Experimental Setup: We compare the software and FPGA/ASIC versions of ERT against BWA-MEM (v0.7.17 release) and BWA-MEM2 (commit `ebc2378`) (refer Table II). ERT-PM adds the prefix-merging optimization while ERT-KR includes both prefix-merging and k-mer reuse. All software comparisons were performed on one of the best-available CPU instances from AWS EC2, `c5n.18xlarge` running 72 threads. BWA-MEM, BWA-MEM2 and software ERT scale well with thread count, given sufficient memory bandwidth. The detailed system configuration is shown in Table I. CPU power was estimated using Intel's RAPL interface. In software, the k-mer reuse optimization resulted in a 1.2% slowdown over prefix-merged ERT. This comes from the overheads of sorting k-mers prior to backward search, maintaining backward searches by k-mer in a metadata table and querying the software managed k-mer reuse cache on each index table/tree access. All reported results consider the three stages of seeding computation in BWA-MEM2: SMEM generation, reseeding, and LAST. We verified that our implementation produces identical seeds as BWA-MEM2 for the complete Illumina Platinum genomes dataset. To estimate the performance of different configurations of ASIC-ERT, we developed a cycle-accurate model using our software implementation and generated memory traces from the corresponding software runs for a representative set of 1 million reads from ERR194147, containing ∼80 % perfect matching reads and ∼20 % non-perfect matching reads similar to the full ERR194147 dataset. Ramulator [38] (commit `7ce65d`) was used to estimate performance and DRAMPower [16] (commit `6c5ebe`) was used to estimate DRAM power and energy.

c5n.18xlarge (AWS EC2 instance)	Intel Xeon Platinum 8124M 3 GHz; 2 sockets; 36 cores; 72 threads
L1 I&D cache	18 x 32KB Instruction; 18 x 32KB Data
L2 cache	18 x 1MB
L3 cache	18 x 1.375MB
Memory	192 GB DRAM

TABLE I
Baseline system configurations.

Configuration	Description
CPU-BWA-MEM	Baseline BWA-MEM: 72 threads
CPU-BWA-MEM2	Baseline BWA-MEM2: 72 threads
CPU-ERT	Best configuration of ERT: 72 threads
ERT	Baseline ERT
ERT-PM	ERT with prefix merging
ERT-KR	ERT with prefix merging and k-mer reuse

TABLE II
Comparison Candidates for Evaluation

ASIC Configuration, Synthesis, and Frequency: Our RTL model for the seeding accelerator was synthesized using Synopsis Design Compiler 2018.2, HPC 28nm process, LVT standard cell library and 12t cells, at 1V (Table III). The seeding processor achieves a 1.38 GHz clock frequency, and is limited by the operating frequency of SRAMs used for context memories. Each SRAM structure in our ASIC was compiled separately: considering word size, number of words, single/dual port requirement. TSMC's 28nm memory compiler is used for power/area estimation.

FPGA Prototype: We prototyped and verified our seeding accelerator on Amazon's EC2 F1 FPGA cloud environment. We chose the `f1.4xlarge` instance with 2 FPGAs and equivalent bandwidth as the CPU configuration (64 GB/s peak

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on March 06,2022 at 19:59:37 UTC from IEEE Xplore. Restrictions apply.

Component	Configuration	SRAM (total)	Area (mm^2)	Power (mW)
Seeding Machines Total	16×	2.72 MB	9.598	11,768.38
K-mer Sorter + Metadata Table	1×	8.26 MB	14.94	9,593.87
K-mer Reuse Cache	1×	4.02 MB	6.99	1,526.76
Seeding Accelerator Total	—	—	31.53	22,889.01
DRAM Power	8 channels	—	NA	2,185.7
Total System Power	—	—	—	25,074.71

TABLE III
ASIC CONFIGURATION AND SYNTHESIS RESULTS.

Component	Configuration	LUT (%)	BRAM (%)	URAM (%)
Index FU	1 × 8	0.32	0	0
Walker FU	3 × 8	13.76	0	0
Leaf Gathering FU	2 × 8	3.36	0	0
Command Queues	0.72 KB x 8	1.92	6.08	0
Context Memories	17.6 KB x 8	0	15.04	3.28
Control Processors	1 × 8	0.56	0	0
Data Fetcher	1 × 8	3.68	0	0
SMEM Result Buffer	2.3 KB x 8	0	0	13.28
MISC.		1.12	0	0
Seeding Machines Total	1 × 8	24.72	21.12	16.56
K-mer Sorter	—	1.95	0.3	26.77
K-mer Reuse Cache	4.01 MB	10.04	5	18.33
Seeding Accelerator Total	1	36.71	26.42	61.66
AWS Shell	–	19.74	12.63	12.20
Total	–	56.45	39.05	73.86

TABLE IV
PER-FPGA CONFIGURATION AND SYNTHESIS RESULTS.

bandwidth per FPGA). Each FPGA in the F1 instance is a Xilinx XCVU9P with 2,586K logic cells, 36.1 Mbits of Block RAM and 270 Mbits of UltraRAM. The accelerator is implemented in System Verilog, placed-and-routed at 250 MHz. System configuration and synthesis results along with the overheads of the AWS Shell interface are shown in Table IV.

VI. RESULTS

Seeding Performance: Figure 11 shows the performance of the seeding step expressed as Million reads/s across different configurations. It can be seen that the software version of ERT provides 2.1× speedup over the state-of-the-art BWA-MEM2 baseline running on 72 threads. This is because ERT greatly reduces the amount of data fetched per read leveraging multi-character lookup and optimizations for spatial locality.

Overall, ASIC-ERT achieves 8.1× improvement in seeding throughput over multi-threaded BWA-MEM2. ASIC-ERT-Baseline utilizes 256 contexts to saturate memory bandwidth and achieves a 2.05× throughput improvement over the CPU-version of ERT. Using prefix-merged radix trees, allows us to reduce the number of backward extensions and further improve throughput by 1.23×. By leveraging temporal locality in the backward search pass, k-mer reuse further improves the overall seeding throughput by 1.56×.

FPGA-ERT achieves a throughput of 3.6 Million reads/s resulting in a speedup of 3.3× over baseline CPU BWA-MEM2. Our FPGA-ERT prototype inherits some limitations of the AWS FPGA-memory interface for ERT-style accesses, not present in the ASIC configurations. For instance, we could not customize the 3rd-party memory-controller IP to return subsequent memory requests to the same DRAM page with lower latency, unless AXI burst transactions with large burst lengths (>64 B) were used. However, always using large burst-length transactions for ERT accesses leads to data wastage. Also, large burst lengths increase datapath complexity and on-chip storage on the FPGA. By issuing 128 B requests when possible, we observed ∼5–8 GB/s per-channel for ERT accesses, although peak channel-bandwidth is 17 GB/s.

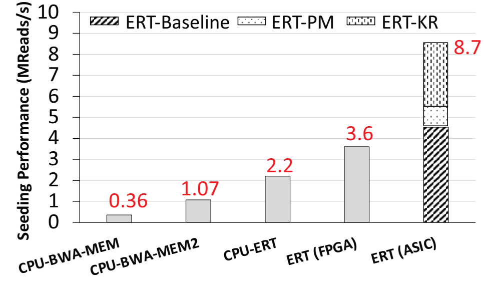

Fig. 11. Seeding performance in Million reads/s.

Memory Access Characteristics: To further understand the reasons for improvement in seeding throughput, we discuss the memory access characteristics of different configurations. Figure 12 shows the average number of memory requests and the data fetched for BWA-MEM and the different configurations of ERT. Compared to BWA-MEM (BWA-MEM2), ERT makes 6.7× (4.5×) fewer memory requests per read. This is because ERT nodes are tightly packed into cache lines to improve spatial locality. On average, ∼3 ERT nodes are traversed per 64 B, utilizing 50% of the data. Also, ERT-KR leverages the k-mer reuse cache to further reduce the number of memory requests by ∼2×. This leads to low data requirements per read (15.1 KB).

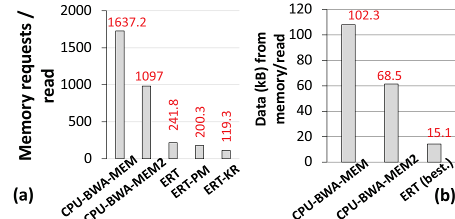

Fig. 12. (a) Memory requests per read (b) Data requirements per read (in KB)

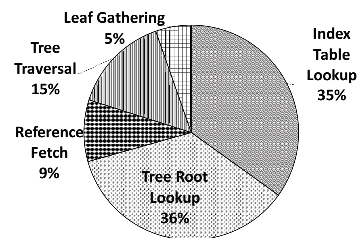

Fig. 13. DRAM page open breakdown for ERT-KR

Seeding Performance Breakdown and Efficiency: Figure 13 shows the distribution of DRAM page opens in ERT for the different steps in seeding. Tree traversal and leaf gathering only contribute to a small number of page opens

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on March 06,2022 at 19:59:37 UTC from IEEE Xplore. Restrictions apply.

indicating high spatial locality in these steps (15% and 5% respectively). Furthermore, the multi-level index table in ERT reduces the number of node traversals by allowing multi-character lookups. In the baseline ERT, the penalty for index table and radix tree root lookup must be paid for almost every k-mer. Since these accesses are random, they contribute to 71% of the DRAM row buffer misses.

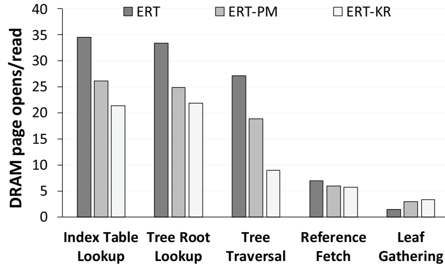

Fig. 14. DRAM page opens per read across optimizations.

Figure 14 shows how prefix-merging and k-mer reuse can be leveraged to reduce the number of row buffer misses in each of these steps. Prefix-merged radix trees reduce the work done during backward search and reduce index table lookups by 24.4%, tree root lookups by 25.5% and tree traversal by 30.4%. In addition, k-mer reuse amortizes the cost of backward searches across several k-mers and leverages temporal locality using the k-mer reuse cache to reduce index table lookups by 37.9%, tree root lookups by 34.3% and tree traversal by 66.7% compared to baseline ERT. Note that reference fetch to obtain the complete strings stored at leaf nodes accounts for 9% and incurs nearly the same cost for all three configurations. Since k-mer reuse does not impose the right-to-left order for the backward extensions of a given read it cannot take advantage of the early termination of backward searches as described in (Section III-F). This results in a slight increase in DRAM page opens for leaf gathering over baseline ERT.

System	Area Efficiency (KReads/s/mm^2)	Energy Efficiency (Reads/mJ)
BWA-MEM (CPU)	0.38	2.89
BWA-MEM2 (CPU)	1.13	8.59
CPU-ERT (best.)	2.32	17.56
ASIC-GenAx [25]	24.23	379.16
ASIC-ERT (best.)	276.36	347.51

TABLE V
SEEDING EFFICIENCY.

Table V compares the area efficiency and energy efficiency of CPU-BWA-MEM, CPU-BWA-MEM2, ASIC-GenAx [25] and the best configuration for ASIC-ERT. ASIC-GenAx is a recent sequence alignment accelerator that leverages CAM-based intersections to perform seeding. When compared to ASIC-GenAx [25] which use large on-chip SRAMs, ASIC-ERT uses lightweight tree walker units and improves area efficiency by 11.4×.

Overall Read Alignment Performance: To match the seeding throughput of FPGA-ERT, we augment ERT with 8 seed-extension accelerator lanes from state-of-the-art seed-extension accelerator SeedEx [26]. Each seed-extension accelerator lane consists of 3 banded Smith-Waterman units (each with 41 PEs, `band-size=41`) and 1 edit-distance unit. Table VI compares the overall read alignment performance of the software versions of BWA-MEM, BWA-MEM2 and our FPGA accelerated read alignment system. When integrated into BWA-MEM2, our FPGA accelerated read-alignment system provides 2.1× higher throughput compared to the software version of BWA-MEM2. Table VII shows the resource consumption of both the seeding and seed-extension accelerators on one FPGA.

System	Instance	Throughput (Mreads/s)
BWA-MEM	c5n.18xlarge	0.216
BWA-MEM2	c5n.18xlarge	0.43
FPGA-ERT (best.) + [26]	f1.4xlarge	0.903

TABLE VI
OVERALL READ ALIGNMENT PERFORMANCE ON AWS EC2

Component	LUT (%)	BRAM (%)	URAM (%)
Seeding Accelerator Total	36.71	26.42	61.66
Seed-Extension Accelerator Total	17.32	2.38	0.68
AWS Shell	19.74	12.63	12.20
Total	73.77	41.43	74.54

TABLE VII
ESTIMATED PER-FPGA RESOURCE UTILIZATION FOR FPGA-ERT + [26]

VII. RELATED WORK

CPU-/GPU-based Seeding: FMD-index based seeding [24], [42] involves many irregular memory accesses and has been found to be bottlenecked by LLC and TLB misses on CPUs [15], [58]. Prior work has explored reordering memory accesses [58] and performing n-character lookups on an n-step FMD-index [15] to improve the locality and data requirements of FMD-index based seeding. However, these implementations focus on exact-match search and do not natively support SMEM computation.

Data-parallel architectures such as GPUs have also been leveraged to accelerate FMD-index search [14] by virtue of high available memory bandwidth and memory level parallelism. However, ERT traversal is inherently not data-parallel and causes significant memory divergence in GPU's SIMD units. When compared to GPU-based aligner SOAP3-dp [45] which achieves 0.48 million reads/s on a Tesla V100 GPU, FPGA-ERT coupled with seed-extension accelerator, SeedEx [26] can improve read alignment throughput by 1.9x×. While this work focuses on FMD-index based seeding, there exists a rich body of work that uses hash-tables for seeding [6], [34], [56] and have optimized its cache behavior [29], [30]. However, hash-based seeding coupled with filtration algorithms [9], [10], [37], [55] are less effective in FMD-index mappers such as BWA-MEM that already produce fewer seeds prior to seed-extension.

Read Alignment Accelerators: Seeding accelerators based on the FMD-index use custom bit-wise operations to traverse the index and improve memory parallelism [17], [20], [53]. However, these implementations soon hit the memory-bandwidth roofline because of character-by-character processing of reads. As a result, the performance of prior FMD-index based FPGA-accelerated solutions for read alignment [7] are on par with

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on March 06,2022 at 19:59:37 UTC from IEEE Xplore. Restrictions apply.

BWA-MEM2 [47]. Several hardware accelerators have also been proposed for read alignment [1], [11], [25], [46], [48], [52]. For instance, Edico Genome's FPGA-based DRAGEN achieves 2.3 million reads/s on a `f1.4xlarge` instance. But since the algorithms used are proprietary, and the output produced is different from BWA-MEM, a direct comparison is difficult. Our FPGA implementation of ERT maintains binary equivalency with BWA-MEM and achieves 3.6 million reads/s for seeding. ASIC-GenCache [48] is a recent sequence alignment accelerator that improves upon ASIC-GenAx [25] seeding by leveraging in-cache operations and reduces redundant work using Bloom filters. When compared to ASIC-GenAx [25] and ASIC-GenCache [48], ASIC-ERT has 11.4× and 2.27× higher iso-area throughput respectively. Darwin [52] is a recent work which demonstrates impressive throughput for long read alignment. However this work focuses on short reads, since long read technology has higher error rates and their seeding algorithms are still evolving.

Radix Tree Applications and Tree Traversal Acceleration: When compared to conventional radix trees and suffix trees [21]: **(1)** ERT eliminates long singleton-path tails and is space-efficient (<64 GB for the human genome). **(2)** ERT is customized for MEM-based seeding. For instance, ERT encodes prefix information at leaf nodes to re-use work across multiple tree traversals and exposes temporal locality for re-using trees across a batch of reads. **(3)** ERT is asymmetric, i.e., supports both forward and backward search by leveraging the reverse complementary nature of reference DNA strands.

Radix trees have also found wide utility as a general-purpose indexing structure for main-memory database systems. ERT leverages several optimizations used in modern database indices [12], [39], [41] to accelerate tree traversal. For instance, cache-line based sub-tree organization to improve spatial locality [27], [35] and leaf node pointer elimination (structural reduction [57]). ERT removes pointers to leaf nodes and uses variable-width pointers (2–4 B) to reduce node size. Internal node pointers are retained because ERT-nodes have variable size. Tree-walking hardware in graph accelerators [8], [31] may also be re-purposed to traverse ERT. However, they cannot leverage all the locality opportunities specific to ERT-based seeding. Our seeding accelerator features custom hardware to support bandwidth-efficient SMEM computation using ERT. For instance, we add hardware sorters and k-mer reuse cache to take advantage of re-use across reads.

VIII. Conclusion

Seeding is an important bottleneck of industry standard DNA alignment genomics pipelines. Current state-of-the-art FMD-Index based seeding algorithms are bandwidth inefficient, and have little spatial or temporal locality. This paper demonstrates a hardware-software co-design approach to accelerate seeding by optimizing for memory bandwidth, rather than memory capacity. The proposed seeding accelerator implemented on AWS F1 cloud combined with state-of-the-art seed extension accelerators can achieve 2.1× speedup over BWA-MEM2 while maintaining binary compatibility. This work can potentially serve as a case study for domain specific acceleration of memory-bound algorithms.

Acknowledgment

We thank the anonymous reviewers for their insightful suggestions which helped improve this paper. We also thank Sanchit Misra and Md. Vasimuddin from Intel Parallel Computing Lab for discussions on earlier versions of the paper and for assistance in integrating ERT with BWA-MEM2.

References

[1] "Accelerate precision medicine with microsoft genomics," https://www.azure.microsoft.com/en-us/resources/accelerate-precision-medicine-with-microsoft-genomics/.

[2] "Genome analysis toolkit: Variant discovery in high-throughput sequencing data," https://github.com/gatk-workflows/gatk4-genome-processing-pipeline.

[3] "Illumina accounted for 74% of global sequencing instrument market in 2014," http://www.gosreports.com/illumina-accounted-for-74-of-global-sequencing-instrument-market-in-2014/.

[4] "Illumina remains ngs leader but competitors expected to gain, survey finds," https://www.genomeweb.com/sequencing/illumina-remains-ngs-leader-competitors-expected-gain-survey-finds.

[5] "Run the germline gatk best practices pipeline for $5 per genome," https://software.broadinstitute.org/gatk/blog?id=11415.

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

[7] N. Ahmed, V.-M. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars, "Heterogeneous hardware/software acceleration of the bwa-mem dna alignment algorithm," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2015, pp. 240–246.

[8] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 105–117, 2016.

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

[10] M. Alser, H. Hassan, A. Kumar, O. Mutlu, and C. Alkan, "Shouji: a fast and efficient pre-alignment filter for sequence alignment," *Bioinformatics*, vol. 35, no. 21, pp. 4255–4263, 2019.

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

[12] M. Boehm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner, "Efficient in-memory indexing with generalized prefix trees."

[13] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand *et al.*, "Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 951–966.

[14] A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, "Boosting the fm-index on the gpu: effective techniques to mitigate random memory access," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 12, no. 5, pp. 1048–1059, 2015.

[15] A. Chacón, J. C. Moure, A. Espinosa, and P. Hernández, "n-step fm-index for faster pattern matching," *Procedia Computer Science*, vol. 18, pp. 70–79, 2013.

[16] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens., "Drampower: Open-source dram power & energy estimation tool." [Online]. Available: http://www.drampower.info

[17] M.-C. F. Chang, Y.-T. Chen, J. Cong, P.-T. Huang, C.-L. Kuo, and C. H. Yu, "The smem seeding acceleration for dna sequence alignment," in *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*. IEEE, 2016, pp. 32–39.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on March 06,2022 at 19:59:37 UTC from IEEE Xplore. Restrictions apply.

[18] Y.-T. Chen, J. Cong, J. Lei, and P. Wei, "A novel high-throughput acceleration engine for read alignment," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 2015, pp. 199–202.

[19] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Cache-conscious structure layout," in *ACM SIGPLAN Notices*, vol. 34, no. 5. ACM, 1999, pp. 1–12.

[20] J. Cong, L. Guo, P. Huang, P. Wei, and T. Yu, "Smem++: A pipelined and time-multiplexed smem seeding accelerator for dna sequencing," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, p. 206.

[21] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, "Alignment of whole genomes," *Nucleic acids research*, vol. 27, no. 11, pp. 2369–2376, 1999.

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

[23] E. Fernandez, W. Najjar, and S. Lonardi, "String matching in hardware using the fm-index," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011, pp. 218–225.

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

[25] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "Genax: a genome sequencing accelerator," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 69–82.

[26] D. Fujiki, S. Wu, N. Ozog, K. Goliya, D. Blaauw, S. Narayanasamy, and R. Das, "Seedex: A genome sequencing accelerator for optimal alignments in subminimal space," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 937–950.

[27] G. Graefe and P.-A. Larson, "B-tree indexes and cpu caches," in *Data Engineering, 2001. Proceedings. 17th International Conference on*. IEEE, 2001, pp. 349–358.

[28] S. Greenstein, J. Holt, and L. McMillan, "Short read error correction using an fm-index," in *2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2015, pp. 101–104.

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

[30] F. Hach, I. Sarrafi, F. Hormozdiari, C. Alkan, E. E. Eichler, and S. C. Sahinalp, "mrsfast-ultra: a compact, snp-aware mapper for high performance sequencing applications," *Nucleic acids research*, vol. 42, no. W1, pp. W494–W500, 2014.

[31] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[32] X. Jiang, X. Liu, L. Xu, P. Zhang, and N. Sun, "A reconfigurable accelerator for smith–waterman algorithm," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 12, pp. 1077–1081, 2007.

[33] D. Karolchik, A. S. Hinrichs, and W. J. Kent, "The ucsc genome browser," *Current protocols in bioinformatics*, vol. 40, no. 1, pp. 1–4, 2012.

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

[35] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "Fast: fast architecture sensitive tree search on modern cpus and gpus," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 339–350.

[36] D. Kim, L. Song, F. P. Breitwieser, and S. L. Salzberg, "Centrifuge: rapid and sensitive classification of metagenomic sequences," *Genome research*, vol. 26, no. 12, pp. 1721–1729, 2016.

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

[38] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[39] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner, "Kiss-tree: smart latch-free in-memory indexing on modern architectures," in *Proceedings of the Eighth International Workshop on Data Management on New Hardware*. ACM, 2012, pp. 16–23.

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

[41] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 38–49.

[42] H. Li, "Exploring single-sample snp and indel calling with whole-genome de novo assembly," *Bioinformatics*, vol. 28, no. 14, pp. 1838–1844, 2012.

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

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

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

[46] R. McMillen and M. Ruehle, "Bioinformatics systems, apparatuses, and methods executed on an integrated circuit processing platform," https://www.google.com/patents/US9014989, Apr. 21 2015, uS Patent 9,014,989.

[47] V. Md, S. Misra, H. Li, and S. Aluru, "Efficient architecture-aware acceleration of bwa-mem for multicore systems," in *Proceedings of the Thirty-Third IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2019.

[48] 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 *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 334–346.

[49] S. H. Pugsley, A. Deb, R. Balasubramonian, and F. Li, "Fixed-function hardware sorting accelerators for near data mapreduce execution," in *Computer Design (ICCD), 2015 33rd IEEE International Conference on*. IEEE, 2015, pp. 439–442.

[50] J. T. Simpson and R. Durbin, "Efficient de novo assembly of large genomes using compressed data structures," *Genome research*, vol. 22, no. 3, pp. 549–556, 2012.

[51] W. Tang, W. Wang, B. Duan, C. Zhang, G. Tan, P. Zhang, and N. Sun, "Accelerating millions of short reads mapping on a heterogeneous architecture with fpga accelerator," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012, pp. 184–187.

[52] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 199–213.

[53] Y. Wang, X. Li, D. Zang, G. Tan, and N. Sun, "Accelerating fm-index search for genomic data processing," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: ACM, 2018, pp. 65:1–65:12. [Online]. Available: http://doi.acm.org/10.1145/3225058.3225134

[54] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2009.

[55] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating read mapping with fasthash," in *BMC genomics*, vol. 14, no. 1. BioMed Central, 2013, p. S13.

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

[57] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory oltp databases with hybrid indexes," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1567–1581.

[58] J. Zhang, H. Lin, P. Balaji, and W.-c. Feng, "Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 2013, pp. 377–384.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on March 06,2022 at 19:59:37 UTC from IEEE Xplore. Restrictions apply.