# ALPHA: A Novel Algorithm-Hardware Co-design for Accelerating DNA Seed Location Filtering

Fazal Hameed , Asif Ali Khan, and Jeronimo Castrillon, *Senior Member, IEEE*

**Abstract**—Sequence alignment is a fundamental operation in genomic analysis where DNA fragments called reads are mapped to a long reference DNA sequence. There exist a number of (in)exact alignment algorithms with varying sensitivity for both local and global alignments, however, they are all computationally expensive. With the advent of high-throughput sequencing (HTS) technologies that generate a mammoth amount of data, there is increased pressure on improving the performance and capacity of the analysis algorithms in general and the mapping algorithms in particular. While many works focus on improving the performance of the aligner themselves, recently it has been demonstrated that restricting the mapping space for input reads and filtering out mapping positions that will result in a poor match can significantly improve the performance of the alignment operation. However, this is only true if it is guaranteed that the filtering operation can be performed significantly faster. Otherwise, it can easily outweigh the benefits of the aligner. To expedite this pre-alignment filtering, among others, the recently proposed GRIM-Filter uses highly-parallel processing-in-memory operations benefiting from light-weight computational units on the logic-in-memory layer. However, the significant amount of data transferring between the memory and logic-in-memory layers quickly becomes a performance and energy bottleneck for the memory subsystem and ultimately for the overall system. By analyzing input genomes, we found that there are unexpected data-reuse opportunities in the filtering operation. We propose an algorithm-hardware co-design that exploits the data-reuse in the seed location filtering operation and, compared to the GRIM-Filter, cuts the number of memory accesses by 22-54%. This reduction in memory accesses improves the overall performance and energy consumption by 19-44% and 21-49%, respectively.

**Index Terms**—Genome sequencing, seed location filtering, processing-in-memory, DNA sequence alignment.

## 1 Introduction

The recent developments in second-generation and third-generation sequencing technologies have enabled the production of DNA sequence data in an unprecedented volume and with extremely low cost. The Illumina NovaSeq 6000 sequencer, for instance, can produce up to 6 terabases of data in less than two days, sequencing 20 billion *reads* (i.e., DNA fragments of length 150 base-pairs each) and costing only 1000$ per human gemone [1], [2]. This has enabled the usage of these technologies in new domains such as genome sequencing of new species, sequencing of individual genomes, and single-cell sequencing, which were previously either too costly or limited by technology. In most of these applications, reads are typically mapped and aligned to reference sequences or genomes to determine similarities or differences between them.

Sequence alignment is a fundamental yet computationally expensive operation in genome analysis. It starts from *read mapping* where each read is checked for alignment with one or more locations in the reference genome based on the similarity between the read and the reference sequence at that location. Subsequently, reads are aligned at the mapped locations using compute-intensive dynamic-programming based algorithms such as Smith-Waterman and Needleman-Wunch [3], [4]. The innovations in sequencing platforms and generation of a mammoth volume of data put tremendous pressure on expediting the alignment operation. This necessitates further improvements in the algorithms and the underlying architectures.

To accelerate the alignment operation, many recent efforts have been ramped up towards algorithmic transformations, development of new algorithms, and architectural improvements [2], [5], [6], [7], [8], [9]. At the same time, promising developments have been reported in *pre-alignment filtering*, an operation that efficiently determines if a particular location in the reference genome would result in a good or a poor match for a given read without actually performing the alignment operation [10], [11], [12], [13]. A poor match indicates that the read and the selected location in the reference genome will not match even after alignment, hinting at skipping the complex alignment step. This reduction in the number of candidate locations (up to 59% [11]) in the reference sequence for alignment operation can significantly reduce the overall end-to-end alignment time.

Several pre-alignment filters have been proposed of late [10], [11], [12], [14], [15], [16], [17], [18], [19], [20]. While most of these filter implementations are highly parallel and compute-efficient, they require significantly high memory bandwidth to classify candidate locations of the reference genome into good and poor matches. The recently proposed GRIM-Filter [13] hides this bottleneck by leveraging the high memory bandwidth and processing-in-memory capabilities of 3D-stacked memory architectures. Although authors report considerable improvements in performance and accuracy compared to the state-of-the-art, the GRIM-Filter requires a large number of memory lookups. Our analysis shows that the probability of repetitive tokens in

- *The authors are with the Chair for Compiler Construction and the Center for Advancing Electronics Dresden (cfaed) at Technische Universität Dresden, 01069 Dresden, Germany.*
  *E-mails: {fazal.hameed, asif_ali.khan, jeronimo.castrillon}@tu-dresden.de*
- *F. Hameed is also affiliated with the institute of space technologies (IST), Islamabad 44000, Pakistan.*

*Manuscript received July 2020; revised May 2021.*

human genomes is up to 24-60%. We show that there is a potential opportunity for this data reuse and propose a new algorithm and an architectural extension that leverages this to reduce the number of memory accesses and arithmetic operations.

The major contributions of this paper are:

- An improved algorithm that, unlike GRIM-Filter, compares only distinct tokens of the input reads to the reference genome. This results in considerable performance improvement.
- We propose necessary changes to the underlying hardware and introduce low overhead extensions. Concretely, we use a preprocessing unit that scans input reads for extracting tokens repetition information and a small *token count table* that stores this information for later processing.
- An efficient scheduling unit that processes the data-reuse in a way to reduce the number of memory lookups and arithmetic operations.
- A detailed design space analysis to evaluate the impact of architectural parameters on the performance, energy consumption, and hardware overhead.

Our design achieves simultaneous performance and energy improvements of 19-44% and 21-49%, respectively, compared to the baseline GRIM-Filter.

The paper is organized as follows. The background and related work is briefed in Section 2 followed by motivation in Section 3. In Section 4, the proposed algorithm-hardware co-design of seed location filtering is elaborated. Experimental results and analysis are reported in Section 5 followed by conclusion in Section 6.

## 2 BACKGROUND AND RELATED WORK

This section provides an overview of the sequence alignment operation and seed location filtering and explains the implementation and functionality of the GRIM-Filter [13].

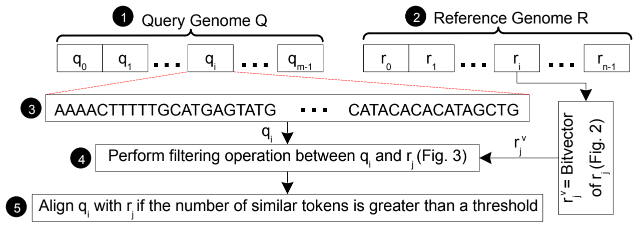
### 2.1 Sequence alignment

A genome is an organism's entire set of DNA, covering all of its genes. Genomic analysis extracts meaningful information from an uncharacterized genome by analyzing its function and structure. Sequence alignment or sequence mapping is a fundamental operation in this analysis [14], [15], [16], [17], [18], [19], [20]. The sequencing machine outputs the sequenced genome in millions of short DNA sequences called *reads*. These reads carry no information about their actual location in the genome. The sequence alignment operation searches for and aligns these reads to their actual position in a reference genome. Classical alignment algorithms based on dynamic programming are prohibitively slow at aligning typical genomes that range in the billions of bases. State-of-the-art aligners use the *seed-and-extend* approach to tackle this problem [2], [21], [22], [23], [24], [25], [26], [27].

The seed-and-extend alignment operation consists of four steps: seed generation, seed mapping, seed extension, and read alignment. A *seed* is a substring of a DNA read that matches, exactly or near-exactly, to one or more locations in the reference genome. In the first step, the aligner searches for potential seeds in a read using different techniques. For instance, k-mer seeds are generated by a sliding window over a read [28], [29]. During the seed mapping, seeds are mapped to their candidate locations in the reference genome. In the seed extension step, the matched seeds are extended to the right and left directions of the matched locations by taking into account constraints such as maximum mismatches and length of insertion-deletion mutations. After that, the read is aligned to the extended region around the location of the seed using compute-intensive dynamic programming based algorithms [3], [4]. Seed extension and read alignment typically account for more than 30% of the runtime in the genomic analysis [2], [30]. Therefore, seed location filters (cf. Section 2.2) are employed to proactively determine and skip seed locations in the reference genome that would result in poor alignment, without actually performing the complex seed extension and alignment operations.

### 2.2 Seed location filtering

The seed location filtering problem can be formulated as follows: Given a query genome $Q = \{q_0, q_1, \ldots, q_{m-1}\}$ (❶ in Fig. 1) and a reference genome $R = (r_0, r_1, ..., r_{n-1})$ (❷), determine and filter out candidate seed map locations that would result in a poor match, without actually performing the expensive alignment operation. Each $q_i$ and $r_j$ are short segments of the query and reference genomes respectively and are referred to as *reads* and *bins*. Each read and bin is typically several hundred base-pair long. Each base pair represents one of the four nucleotide bases, i.e., A, C, G, and T. For instance, ❸ in Fig. 1 shows an illustrating example of $q_i$ in the query genome containing a sequence of base pairs. The inputs to the seed location filter (❹) are a read from the query genome ($q_i$) and a bit vector representing a bin from the reference genome ($r_j^v$) (Fig. 2). The filter accumulates the values in ($r_j^v$) and compares it to a predefined threshold (cf. Fig. 3). If the accumulator value is greater than the threshold, this implies that many tokens in the input read $q_i$ are also present in the corresponding bin $r_j$, the filter forwards $q_i$ to the aligner to perform an alignment between $q_i$ and $r_j$ (❺). Otherwise, the alignment step is skipped.

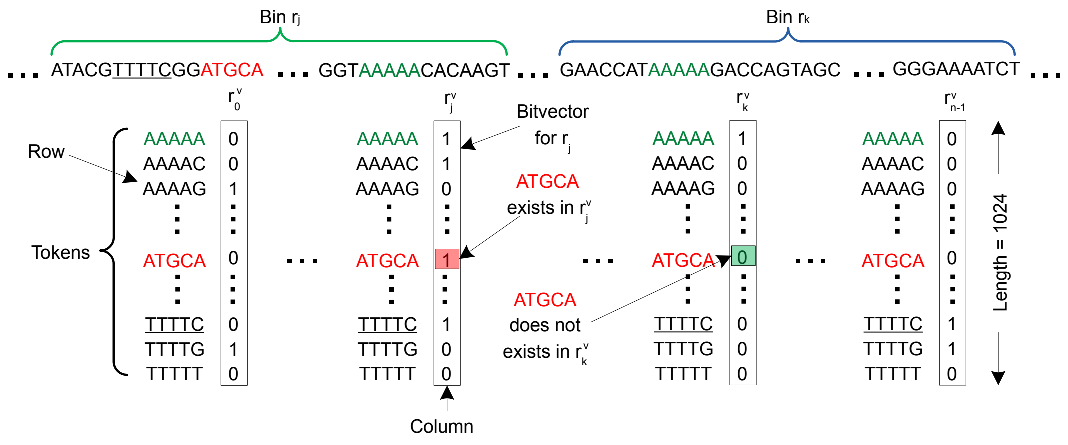


1 Query Genome Q
2 Reference Genome R
q0 q1 ... qi ... qm-1
r0 r1 ... ri ... rn-1
3 AAAACTTTTTGCATGAGTATG ... CATACACACATAGCTG
qi
4 Perform filtering operation between qi and rj (Fig. 3)
rjv
rjv = Bitvector of rj (Fig. 2)
5 Align qi with rj if the number of similar tokens is greater than a threshold

Fig. 1. Seed locating filtering between reads of the query genome and bins of the reference genome

### 2.3 GRIM-Filter

GRIM-Filter is a recent algorithm for seed location filtering which is optimized for a processing-in-memory architecture (cf. Section 2.4). It divides each bin into *tokens* of size 5 base pairs, ranging from $AAAAA$ to $TTTTT$. For the reference genome, the GRIM-Filter maintains a per-bin *bitvector* where each bit (referred to as the *presence* bit) of the bit vector



Bin rj
Bin rk
... ATACGTTTTCGGATGCA ... GGTAAAAACACAAGT ... GAACCATAAAAAGACCAGTAGC ... GGGAAAATCT ...
Row
Tokens
Bitvector for rj
ATGCA exists in rj
ATGCA does not exists in rk
Column
Length = 1024

Fig. 2. Metadata organization of the GRIM-Filter where each bit at [Row, Column] specifies if a token (corresponding to a row) exists in the relevant bin (indicated by the column) of the reference genome. Each column represents the metadata associated with a particular bin of the reference genome. Token size is 5 in this figure.

indicates whether a particular token exists in a bin or not, as shown in Fig. 2. The bit vectors of all bins in the reference genome are created only once and are used to find similarities between the reference and query genomes. Fig. 2 presents an illustrative example showing the bit vectors associated with selected bins (i.e., $r_0$, $r_j$, $r_k$, and $r_{n-1}$) of the reference genome. The bit vector of the bins $r_j$ and $r_k$ is denoted as $r_j^v$ and $r_k^v$ respectively. The presence bit of token $ATGCA$ is set in $r_j^v$ because this token is present in $r_j$. Similarly, the presence bit of token $ATGCA$ is unset in $r_k^v$ as $ATGCA$ does not exist in $r_k$.

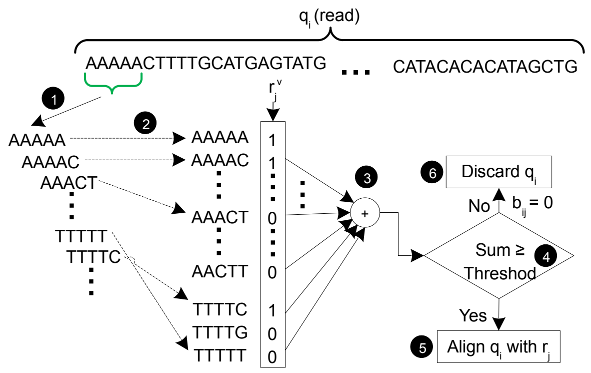


Fig. 3. Steps involved in the GRIM-Filter to determine a good or bad match between a read of the query genome ($q_i$) and a bin of the reference genome ($r_j$) for a token size = 5

For a given read ($q_i$) of the query genome, the GRIM-Filter examines whether this matches to a certain bin of the reference genome (say $r_j$) or not. To do this, it reads $q_i$ token by token (❶ in Fig. 3), and for each token, it accesses the presence bit of the corresponding token in $r_j^v$ (❷). The GRIM-Filter then sums all these presence bits together (❸) and compares the result with a predefined threshold (❹). If the accumulated sum is greater than the threshold, the filter forwards this read to the alignment step to actually map $q_i$ to $r_j$ (❺). Otherwise, it skips aligning $q_i$ to $r_j$ (❻) and moves to the next bin. Only the $r_j$s that share enough tokens with a given $q_i$ pass the GRIM-Filter. By doing so, the GRIM-Filter ensures that no $r_j$ that results in a correct matching with $q_i$ is incorrectly rejected, thereby maintaining a 0% false rejection.

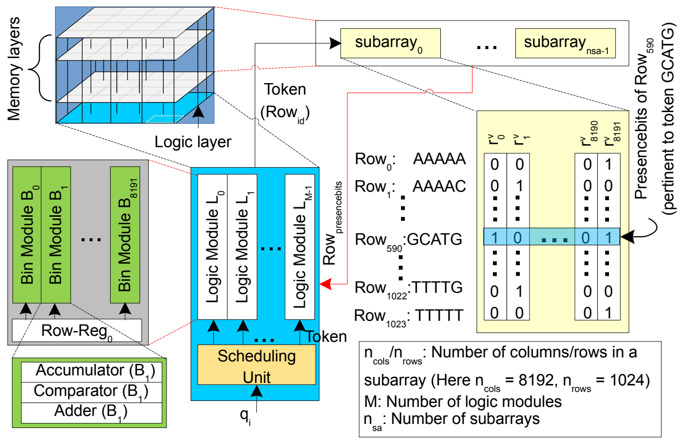


Fig. 4. Processing-in-memory based GRIM-Filter architecture

## 2.4 PIM implementation of the GRIM-Filter

This subsection describes the implementation of the GRIM-Filter using 3D-stacked memory architecture. Processing-in-memory (PIM) is a novel concept that allows performing computation inside the memory subsystem. By employing small computation units inside the memory, PIM reduces the amount of data that has to be moved from memory to external processors, thereby improving the performance and energy efficiency of computing systems. The GRIM-Filter is a potential candidate to be implemented using PIM due to its memory intensity as it requires simple operations involving additions and comparisons (cf. Fig. 3). The PIM implementation of the GRIM-Filter is illustrated in Fig. 4 and is realized using a 3D-stacked DRAM memory, consisting of several memory layers and a single logic layer. The memory layers are hierarchically decomposed into subarrays (i.e., subarray$_0$ to subarray$_{nsa-1}$), rows (i.e., row$_0$ to row$_{nrows-1}$), and columns (i.e., col$_0$ to col$_{ncols-1}$).

In Fig. 4, we assume that each subarray consists of 1024 rows (i.e., n$_{rows}$ = 1024). Each row corresponds to a unique token i.e., $AAAAA$ and $TTTTT$ are mapped to row$_0$ and row$_{1023}$ respectively. Each row is indexed by its corresponding token and stores the presence bit information of the same token for multiple bins/bitvectors (8192 in the figure) of the reference genome. For instance, row$_{590}$ of subarray$_0$ stores the presence bit of token $GCATG$ relevant to 8192 bins (i.e., $r_0$ to $r_{8191}$) of the reference genome. As illustrated in the figure, the GRIM-Filter stores the bit vectors of each bin of the reference genome in a separate column. For instance, the bit vectors of bins $r_0$ and $r_1$ are stored in the first and second column of subarray$_0$.

To carry out filtering between $q_i$ and $r_j$, the GRIM-Filter provides a bin module that operates on $r_j^v$. The bin module is composed of an accumulator (initialized to zero), an adder, and a comparator. The module scans each token in $q_i$ and add the presence bits into the accumulator. Once all the tokens in $q_i$ are exhausted, the comparator associated with $r_j$ is used to compare the value of the accumulator with a set threshold (cf. Fig. 3). If the value is greater than the threshold, the read is forwarded to the aligner; otherwise, it is dropped.

At a particular instance, the GRIM-Filter examines the presence of a single token of $q_i$ in $M \times n_{cols}$ bins of the reference genome. The parameter n$_{cols}$ refers to the number of columns in a subarray while $M$ is the number of logic modules (cf. Fig. 4). For the same token of $q_i$, the GRIM-Filter fetches $M$ rows from $M$ different subarrays in parallel and load them in $M$ row registers (i.e., Row-Reg$_0$ to Row-Reg$_{M-1}$). These $M$ row registers store the presence bits of $M \times n_{cols}$ bins of the reference genome pertinent to the same token. After the presence bits are loaded into $M$ row registers, the GRIM-Filter processes them in parallel using $M \times n_{cols}$ bin modules. For instance, in Fig. 4, the bin module B$_1$ adds the second bit of Row-Reg$_0$ to its accumulator.

Fig. 5 illustrates the sequence of token accesses (i.e., memory row) in the GRIM-Filter when comparing an input $q_i$ read with 8192 bins of the reference genome ($R$). For the sake of explanation, we assume that n$_{cols}$ = 8192, $M$ = 1, and the bit vectors of these 8192 bins (i.e., $r_0^v$ to $r_{8191}^v$) are stored in subarray$_0$. For the example $q_i$ read in Fig. 5, row$_0$ (for token $AAAAA$) of subarray$_0$ is fetched at time $t_0$ and loaded into the row register of the logic module. Subsequently, the 8192 presence bits in the row register are processed by 8192 bin modules in parallel and the relevant accumulators are computed accordingly. Similarly, at time $t_1$, row$_1$ (for token $AAAAC$) is fetched in the row register to examine the presence of $AAAAC$ in 8192 bins of $R$. When all tokens in $q_i$ are exhausted, all the comparators associated with 8192 bin modules are used to compute the seed location filter outcome of 8192 bins in $R$. The GRIM-Filter then moves on to other subarrays and compares $q_i$ with a new set of bins in $R$. When $q_i$ finishes comparing with all bins of $R$, then the same sequence of operations are performed on the remaining reads of the query genomes (i.e., $q_{i+1}$, $q_{i+2}$, ..., $q_{m-1}$).

## 3 Data Access Analysis

This section presents a motivating example that shows how the GRIM-Filter incurs a large number of memory accesses which deteriorates both the performance and the energy efficiency. As depicted in Fig. 5, at a certain time period, the GRIM-Filter compares a read of the query genome (say $q_i$) to a subset of bins in the reference genome. For the sake of explanation, we stick to the same illustrative example (cf. Fig. 5) and consider the same assumptions made in Section 2.4. For the example $q_i$ read from Fig. 5, the GRIM-Filter loads row$_{590}$ (for token $GCATG$) three times (i.e., at time $t_{10}$, $t_{113}$, and $t_{152}$) into the row register of the logic module (cf. Fig. 4). After performing computation on the row register of row$_{590}$, the accumulators associated with bin modules $B_0$ and $B_{8191}$ are incremented three times because the presence bit of token $GCATG$ is one in $r_0^v$ and $r_{8191}^v$. We classify the row accesses into the following categories: 1) the first access to a row belonging to a particular token of $q_i$ is referred to as *compulsory* access. For instance, the access to row$_{590}$ at time $t_{10}$ for token $GCATG$ is compulsory. 2) the following accesses to the same row are referred to as *redundant* accesses because they could be avoided, as explained in the next section. For the given example, the accesses to row$_{590}$ for token $GCATG$ at time $t_{113}$ and $t_{152}$ are redundant.

By analyzing different query genomes (listed in Table 1), we found that the majority of the reads exhibit a high percentage of repeated tokens. Repeated tokens are tokens that are accessed more than once in a given read. Fig. 6 shows the percentage of repeated tokens in reads of the query genome ERR240730_2 in Table 1. The bars show the overall percentage of the repeated tokens for each read length, and the different colors indicate the breakdown for

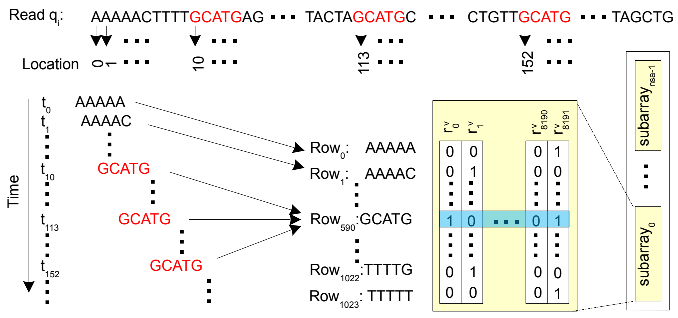


Fig. 5. Memory row accesses required in the GRIM-Filter for different tokens in an example $q_i$ read. $M$ = 1 and n$_{cols}$ = 8192 in this figure

TABLE 1
Benchmark data obtained from the 1000 Genome Project [31].

| Benchmark | No. of reads | Sample |
|---|---|---|
| ERR240726_1 / ERR240726_2 | 4031354 / 4389429 | NA20753 |
| ERR240727_1 / ERR240727_2 | 4082203 / 4013341 | NA20754 |
| ERR240728_1 / ERR240728_2 | 3894290 / 4013341 | NA20759 |
| ERR240729_1 / ERR240729_2 | 4013341 / 4082472 | NA20761 |
| ERR240730_1 / ERR240730_2 | 4082472 / 4082472 | NA20766 |

FTP link to all benchmarks/samples sources[1]. Individual benchmark's details and data can be accessed at[2] where *id* corresponds to the sample numbers provided in the right-most column of the table.

1. http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/
2. https://www.internationalgenome.org/data-portal/sample/id

different count values, e.g., "2" shows the percentage of times the tokens are repeated 2 times.

For evaluation, we took ten pair-end input genomes, basically the same as the GRIM-Filter, from the 1000 Genome Project [32], available at [31]. Table 1 lists the benchmark details that are relevant to this work. For additional information on the dataset, e.g., Biosample ID, populations, sex, cell line source, accession numbers, etc., we provide links to their origins. By analyzing different query genomes (listed in Table 1), we found that the majority of the reads exhibit a high percentage of repeated tokens. Repeated tokens are tokens that are accessed more than once in a given read. Fig. 6 shows the percentage of repeated tokens in reads of the query genome ERR240730_2 in Table 1. The bars show the overall percentage of the repeated tokens for each read length, and the different colors indicate the breakdown for different count values, e.g., "2" shows the percentage of times the tokens are repeated 2 times.

On average, for a 400 read size, the percentage of repeated tokens is 24.2%. As shown, this increases as we increase the read size. As explained above, the GRIM-Filter always fetches a row independent of whether it is a compulsory or a redundant row access. We exploit this repetition of tokens in the query reads to completely eliminate redundant row accesses as explained in the next section.

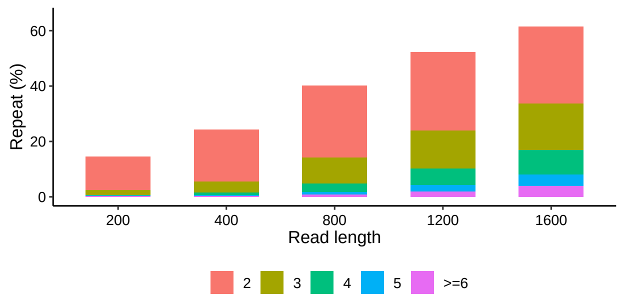


Fig. 6. Percentage of repeated tokens across all reads of the query genome ERR240730_2 in Table 1. Token size = 5 base pairs, and read length is varied from 200 to 1600 base pairs in this figure.

## 4 The ALPHA Filter

This section introduces our proposed Architecture for Low-Power and Hasty seed location filtering based on novel Algorithm (ALPHA).

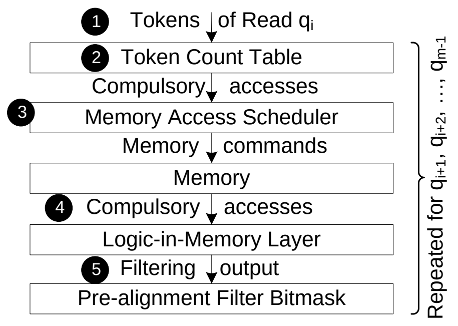


Fig. 7. Sequence of operations in ALPHA filter.

### 4.1 Overview

Fig. 7 shows the sequence of operations in ALPHA filter. Similar to the GRIM-Filter [13], ALPHA is implemented as a processing-in-memory (PIM) architecture, providing advantages over other seed location filters [10], [11], [12]. As mentioned in Section 3, a major drawback of the GRIM-Filter is that it unnecessarily incurs a large number of redundant memory accesses which deteriorates both performance and energy efficiency. To eliminate the redundant memory accesses, ALPHA makes algorithmic transformations to the GRIM-Filter and changes the underlying architecture as depicted in Fig. 8.

The ALPHA filtering commence with scanning the tokens of a read $q_i$ (❶ in Fig. 7) followed by updating the token repetition information (❷) in a newly proposed *Token Count Table* (TCT). ALPHA introduces a preprocessing unit that extracts tokens repetitions (i.e., which ultimately translate to row access information) from the reads of the query genome (Section 4.2). In step ❸, the memory access scheduler avoids redundant memory accesses by only scheduling compulsory accesses based on the repetition information in the TCT. Section 4.3 describes the hardware support to further reduce the number of memory accesses and enable parallel processing of multiple reads of the query genome. Once all compulsory rows are accessed (❹) pertinent to the tokens contained in a given read, ALPHA generates the pre-alignment filter bitmask (❺). This bitmask determines the filtering outcome (i.e., a logical 1 for a potential match and logical 0 for a mismatch based on the threshold) and transmit it to the alignment step. It is worth mentioning that the filtering outcome of ALPHA and the GRIM-Filter is the same. However, compared to the GRIM-Filter, ALPHA changes the way in which the data is accessed from the memory and processed (details in Section 4.2). Similar to the GRIM-Filter, ALPHA maintains a 0% false rejection ensuring that no candidate location in $R$ that results in a correct matching with a particular read in $Q$ is incorrectly rejected by the filter.

### 4.2 Preprocessing of the query genome

The GRIM-Filter first scans the tokens of a read $q_i$ one by one and then compares each token with all bins in $R$. In contrast, ALPHA preprocesses all the tokens of $q_i$ and updates its repetition information in the TCT before comparing them with the bins in $R$.

Fig. 9 depicts the organization of the Token Count Table (TCT), consisting of $2k$ columns. For illustration purposes, $k$ is set to 3 in Fig. 9. Each column in TCT stores the repetition information of exactly one read of the query genome and is identified by a $\lceil log_2(k) \rceil + 1$ bits $q_{tag}$ field. For a given read of the query genome (say $q_i$), each column entry of the TCT is indexed with the tokens (i.e., ranges from $AAAAA$ to $TTTTT$) contained in $q_i$. The column entry $count_{ri}$ records how many times the r$_{th}$ token appears in the read $q_i$. The TCT tracks the counts of the tokens relevant to few reads (i.e., $2k$) of the $m$ total reads in the query genome (i.e., $2k << m$).

For each read/column in the TCT, the preprocessing unit maintains two overhead bits named *scancomplete* and *processed* (cf. Fig. 9). A column with a set *scancomplete* bit

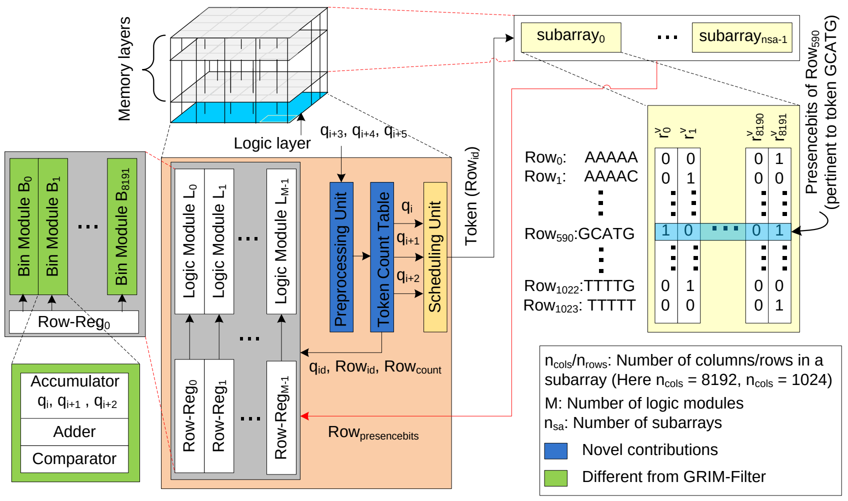


Fig. 8. A high-level overview of ALPHA highlighting our novel contributions

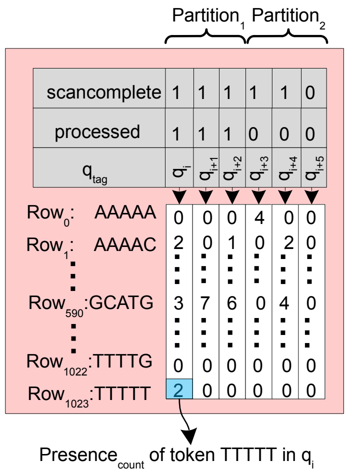


Fig. 9. Organization of the proposed Token Count Table (TCT) with $2k$ columns. In the figure $k = 3$

indicates that the preprocessing unit has scanned all tokens of the relevant read in the query genome, and the token repetition information is updated in the TCT. The *processed* bit shows whether the content of a column is consumed by the filtering unit or not.

Similar to the GRIM-Filter, we process query genome on a read-by-read basis. However, before performing the actual filtering operation, we first preprocess each read to extract the token repetition information. For a given read $q_i$, the preprocessing unit allocates a column in the TCT and clears its associated *processed* and *scancomplete* bits. It then reads $q_i$ token-wise and populates its corresponding TCT entry with the token count. Once all tokens in $q_i$ are exhausted, the *scancomplete* bit associated with $q_i$ is set. As for the replacement of the previous contents, the preprocessing unit only replaces a column in TCT if its contents are already consumed by the filtering step. The *processed* bit, associated with each column, provides this information.

For the example $q_i$ sequence in Fig. 5, the $count_{ri}$ of token $GCATG$ is three after scanning all tokens in $q_i$. For comparison with the reference genome, our filtering algorithm scans only distinct tokens in $q_i$ and adds their corresponding token counts in the TCT. This means that for $q_i$, the row$_{590}$ for token $GCATG$ (cf. Fig. 8) is accessed only once in ALPHA instead of thrice in the GRIM-Filter. In this case, three is added to the accumulator of those bins in the reference genome whose presence bit relevant to $GCATG$ is set. The *processed* bit of $q_i$ in the relevant column is set when $q_i$ is compared with a subset of bins (i.e., $M \times$ n$_{cols}$) selected as candidate locations in the reference genome. This implies that the column can now be allocated to a new unprocessed read of the query genome.

The implementation of the preprocessing unit requires a FIFO buffer and an adder. In the first step, the tokens of a particular read are scanned token wise and stored in the FIFO. The token at the head of the FIFO is used to access a particular TCT entry with the relevant q$_{tag}$ field. An adder is used to increment the value of that entry which is stored back in the TCT. Parallel to the preprocessing unit, the scheduling unit reads the count information of each token from the TCT in a top-to-bottom way and only forwards that row requests to the memory for which the corresponding token entry has a non-zero value. It is worth mentioning that the scheduling unit sends the data to memory only for those reads which are preprocessed.

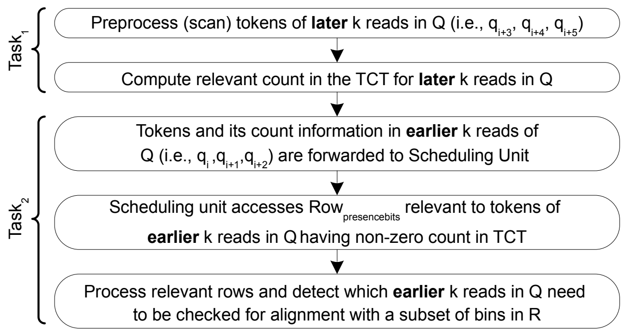

Task1
Preprocess (scan) tokens of later k reads in Q (i.e., qi+3, qi+4, qi+5)
Compute relevant count in the TCT for later k reads in Q
Task2
Tokens and its count information in earlier k reads of Q (i.e., qi,qi+1,qi+2) are forwarded to Scheduling Unit
Scheduling unit accesses Rowpresencebits relevant to tokens of earlier k reads in Q having non-zero count in TCT
Process relevant rows and detect which earlier k reads in Q need to be checked for alignment with a subset of bins in R

Fig. 10. Overlapping tasks (Task$_1$ and Task$_2$) in the improved ALPHA to determine where $k$ reads of the query genome $Q$ needs to be checked for alignment with a subset of bins in the reference genome R

### 4.3 Improved query genome processing

Memory accesses can be further reduced by processing multiple reads of the query genomes at the same time. To enable this, ALPHA preprocesses the counts of the tokens relevant to $k$ reads of the query genome where $2k$ columns of TCT are divided into two equal partitions. These partitions are referred to as Partition$_1$ and Partition$_2$ in Fig. 9. While the scheduling unit is busy in handling counts of one partition belonging to earlier $k$ reads of the query genome (i.e., $q_i$, $q_{i+1}$, $q_{i+2}$ for $k$ = 3), the preprocessing unit computes the counts of the later $k$ reads (i.e., $q_{i+3}$, $q_{i+4}$, $q_{i+5}$ for $k$ = 3) stored in another partition. This implies that Partition$_1$ stores the relevant counts of the tokens belonging to earlier or later reads of the query genome at alternate time intervals.

Fig. 10 shows the sequence of events in ALPHA. The preprocessing unit computes the counts of one partition by scanning all tokens of later $k$ reads of the query sequence (referred to as Task$_1$ in Fig. 10). Once the counts of one partition is computed (i.e., all *scancomplete* bits in that partition are set), the scheduling unit performs necessary memory operations on that partition (referred to as Task$_2$ in Fig. 10). As Task$_1$ preprocesses the later reads of the query sequence, Task$_2$ performs necessary memory operations on the earlier reads. Compared to Task$_1$, Task$_2$ is the performance critical task as the latter accesses many rows storing bit vectors of the reference genome. On the other hand, Task$_1$ is lightweight because it scans the tokens of the relevant bins in the query sequence followed by updating the counts in the small TCT.

In Task$_2$, the scheduling unit only accesses the Row$_{presence}$ bits of those tokens for which the token entry in TCT has a non-zero value. In the example in Fig. 9 we assume that the first, second and third column of the Partition$_1$ contains the counts of the tokens relevant to three reads of the query sequence namely $q_i$, $q_{i+1}$, and $q_{i+2}$. As illustrated, the tags of these reads are stored in the $q_{tag}$ field of the respective column. We further assume that all tokens of the aforementioned reads are scanned completely in Task$_1$ (i.e., *scancomplete* bits are set for these bins). This implies that the scheduling unit can now process the counts of Partition$_1$. As depicted in Fig. 9, the count of token $GCATG$ is 3, 7, and 6 for $q_i$, $q_{i+1}$, and $q_{i+2}$ respectively. In this example, the GRIM-Filter requires 16 (i.e., 3 + 7 + 6 = 16) accesses to row$_{590}$ while ALPHA with $k$ = 1 requires 3 accesses. In contrast, ALPHA with $k$ = 3 only requires a single access to row$_{590}$. However, this reduction in the number of row accesses for ALPHA with $k$ = 3 comes at an additional cost. The conventional GRIM-Filter requires a small bin module (cf. Fig. 4) consisting of one accumulator, one adder and one comparator. In contrast, ALPHA requires a total of $k$ accumulators to hold the accumulated sum of the matched tokens of $k$ reads of the query genome with a particular bin of the reference genome (cf. Fig. 8 for $k$ = 3). Therefore, ALPHA requires additional $k$ - 1 accumulators. It is worth mentioning that the same adder and comparator is used in a serial fashion by processing the counts of additional $k$ - 1 reads of the query genome compared to the GRIM-Filter.

### 4.4 Overhead analysis

The Token Count Table (TCT) tracks the counts of the tokens relevant to recently accessed $2k$ reads of the query genome. The TCT comprises 1024 rows and $2k$ columns where the storage requirement of each column is 1024 $\times$ $\lceil log_2(RL) \rceil$ bits where $RL$ refers to the read length. In addition, the storage overhead for the *scancomplete* and *processed* fields are $2k$ bits each. Also, the bits required for $q_{tag}$ field are $2k \times \lceil log_2(k) \rceil + 1$. For our experiments, we vary $k$ from 1 to 4. So the total number of bits required for $q_{tag}$ adds up to ($2k \times 3$). Considering the above overheads, the total storage requirement of the TCT amounts to $k \times 2.25\,\mathrm{kB}$ assuming that each read consists of 400 base pairs.

The preprocessing unit requires a FIFO buffer to hold the recently scanned tokens belonging to the query genome. For a 32-entry buffer, the storage overhead amounts to 320 bits (32 $\times$ 10-bit/token = 320 bits). In addition, a $\lceil log_2(RL) \rceil$ bit adder is required to update the relevant entry in the TCT. As depicted in Fig. 8, ALPHA also requires $k$ - 1 more accumulators in each bin module. Therefore, the total number of additional accumulators in ALPHA are ($k$ - 1) $\times$ n$_{cols}$ $\times$ $M$. The parameter n$_{cols}$ refers to the number of columns in a subarray while $M$ is the number of logic modules (cf. Fig. 8). The area overhead analysis of ALPHA compared to the GRIM-Filter is provided in Section 5.4 (cf. Table 3 and Fig. 15) considering the overhead of the TCT, the preprocessing unit, and the bin modules.

## 5 Evaluation

This section provides the details of the experimental methodology and the simulation infrastructure that are used for qualitative and quantitative comparison of the proposed pre-alignment filtering solution with the GRIM-Filter [13].

### 5.1 Experimental Methodology

Table 1 lists the name and number of reads of each data set with a read length of 100 base pairs. The longer reads are generated by combining multiple short reads. For instance, a read length with 800 base pairs is generated by combining eight adjacent reads of 100 base pairs. The details of important parameters of our pre-alignment filter (i.e., read and token size) and the memory system used in our evaluations are shown in Table 2. Our reference genome consists of $450 \times 2^{16}$ bins. The total memory requirement to store the bit vectors of the reference genome for a token size of 5 is $450 \times 2^{16} \times 4^5$ bits. This results in a total memory footprint of approximately $3.5\,\mathrm{GB}$ which can be easily stored in today's 3D-stacked memories [33], [34], [35], [36].

The energy and area numbers for the various components on the logic-in-memory layer are estimated using McPat [37] and CACTI [38] which are provided in Table 3. All of our simulations model an HBM2 [34] memory system with latency and energy numbers taken from [39] and are listed in Table 4.

For comparison, we do not consider the time required to generate the metadata of the reference genome because it is a one-time process and is the same for both the GRIM-Filter and ALPHA.

We instead measure the performance of ALPHA and the GRIM-Filter by comparing the actual time of the filtering operation which include the following timing parameters:

1) $t_{memoryaccess}$: The memory access time required to retrieve the bins of the reference genome and load them in the row registers.
2) $t_{preprocessing}$: For ALPHA only, the time spent on preprocessing, which includes the latency overheads of the buffer, the TCT, and the adder.
3) $t_k$: For ALPHA only, the additional latency overheads when the adders and comparators are used in a serial fashion in the bin module for $k > 1$.
4) $t_{forward}$: The time spent by the logic-in-memory layer to forward the result of the filtering operation to the sequence alignment step.

To estimate the time spent on the filtering operation, we used a trace-based simulation environment based on NVMain [40], which faithfully models HBM2 [34] timing parameters highlighted in Table 4, and the above-mentioned timing parameters.

## 5.2 Result Overview

The comparison of ALPHA with the GRIM-Filter in terms of runtime and overall energy consumption is shown in Fig. 11 and Fig. 12 respectively. Our results clearly indicate that each variant of ALPHA outperforms the GRIM-Filter in terms of runtime and energy consumption. As depicted, ALPHA reduces the average runtime by 19.3%, 30.8%, 38.6%, and 43.8% compared to the GRIM-Filter for $k$ = 1, $k$ = 2, $k$ = 3, and $k$ = 4, respectively. Similarly, the energy improvement translates to 21.1%, 33.6%, 42.7%, and 49%, respectively. The runtime and energy benefits primarily come from the reduction in number of DRAM accesses. The energy breakdown in Fig. 12 shows that the overall energy consumption is dominated by the energy consumed by the DRAM and that the energy consumed by the logic components, i.e., adders, accumulators, and comparators, is less prominent. This figure also shows that ALPHA introduces a very small overhead in terms of energy consumption. As $k$ increases, the overhead energy incurred by ALPHA increases, but its impact on the overall energy consumption is largely compensated by the reduction in the energy consumed by the DRAM and the logic-in-memory layer. A detailed energy breakdown of the DRAM and the components on the logic-in-memory layer is presented in the following subsection.

## 5.3 Energy breakdown

For different variants of ALPHA and the GRIM-Filter, we breakdown the energy consumption of the DRAM and the components on the logic-in-memory layer, as shown in Fig. 13 and Fig. 14 respectively. Fig. 13 highlights that with an increasing value of $k$, the DRAM dynamic energy as well as background energy is substantially reduced. This is due to the fact that the probability of the occurrence of repeated tokens is high when more reads of the query genome are analyzed. The increase in repeated tokens reduces the number of DRAM accesses which result in a reduction in dynamic energy while the background energy gain is due to shorter runtime. The overhead energy incurred by ALPHA includes the leakage energy of the additional accumulators, the overall energy consumption of the Token Count Table (TCT) and the preprocessing unit (cf. Section 4.4 for overhead analysis). The energy breakdown in Figure 14 indicates that the overhead energy linearly increases with an increasing value of $k$. However, this increase in the overhead energy is largely offset by the drop in the dynamic and the leakage energy of the logic module. The decrease in the dynamic energy of the logic module with an increasing $k$ is due to the fact that less number of additions are required in ALPHA compared to the GRIM-Filter (cf. Section 4.3).

TABLE 2
Parameters details of the data set and memory

| Parameter | Value |
|---|---|
| Token size | 5 base pairs |
| Read size | 400 base pairs |
| Number of bins in the reference genome | 450 x $2^{16}$ bins |
| Memory size | 4 GB |
| Number of logic modules (M) | 4 |
| Number of memory banks | 64 banks |
| Number of subarrays in a memory bank | 64 |
| Number of subarrays | 64 x 64 = 4096 |
| Number of columns in a subarray ($n_{cols}$) | 8192 bits |
| Number of rows in a subarray ($n_{rows}$) | 1024 |

TABLE 3
Energy and area for different logical components estimated from [37], [38].

| Component | Energy (Dyn) | Power (Leak) | Area [$\mu m^2$] |
|---|---|---|---|
| Adder | 7.62 [fJ]* | 18.0 [nW]* | 0.65/bit |
| Comparator | 5.12 [fJ]* | 12.2 [nW]* | 0.49/bit |
| Accumulator | 10.80 [fJ]* | 21.5 [nW]* | 0.70/bit |
| Buffer (32 entries) | 2.175 [pJ] | 0.5332 [mW] | 588 |
| Row-Reg | 4.84 [pJ] | 2.16 [mW] | 5419 |
| TCT (k = 1) | 6.93 [pJ] | 1.70 [mW] | 14916 |
| TCT (k = 2) | 10.37 [pJ] | 2.61 [mW] | 23563 |
| TCT (k = 3) | 14.81 [pJ] | 4.17 [mW] | 41763 |
| TCT (k = 4) | 25.70 [pJ] | 6.44 [mW] | 69230 |

*Energy/power consumption per bit.

## 5.4 Area breakdown

Although increasing the value of $k$ reduces the runtime and energy consumption of ALPHA compared to the GRIM-Filter, this improvement comes at the cost of additional area overhead (i.e., extra accumulators in the logic-in-memory layer, preprocessing unit and the TCT) as depicted in Fig. 15. For larger $k$, the area overhead becomes more significant on

TABLE 4
HBM2 DRAM energy and latency taken from [39]

| Operation | Value |
|---|---|
| Activation and precharge | 909 [pJ] |
| Access energy | 1.17 [pJ]/bit |
| I/O energy | 0.80 [pJ]/bit |
| Background power | 27.5 [mW] |
| $t_{RC}$-$t_{RCD}$-$t_{RP}$-$t_{RAS}$-$t_{CL}$ | 45-16-16-29-16 [ns] |
| $t_{RRD}$-$t_{WR}$-$t_{FAW}$-$t_{WTRl}$-$t_{WTRs}$ | 2-16-12-8-3 [ns] |
| $t_{CCD_L}$-$t_{tCCD_S}$-$t_{BURST}$ | 4-2-2 [ns] |

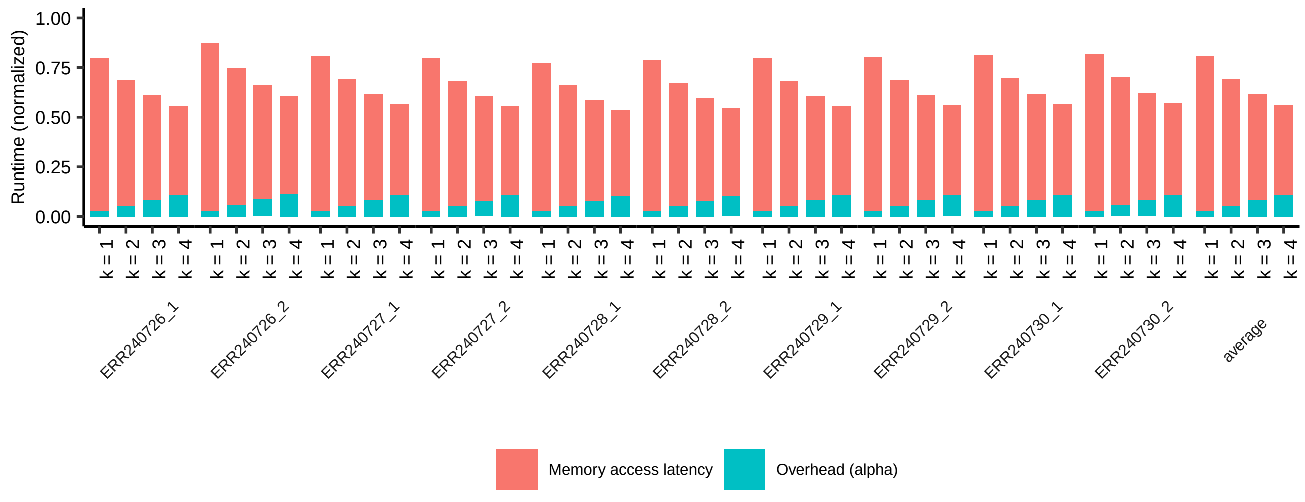


Fig. 11. Runtime comparison of ALPHA with the GRIM-Filter. All results are normalized to the GRIM-Filter.

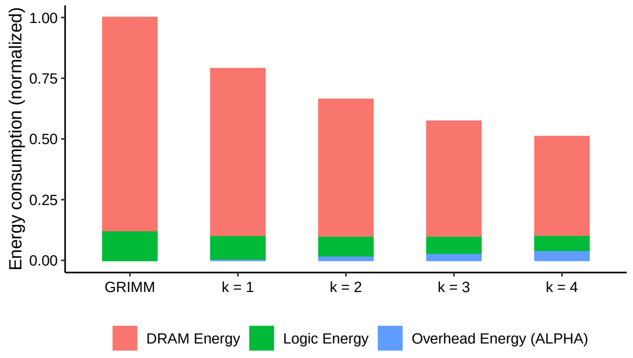


Fig. 12. Overall energy breakdown

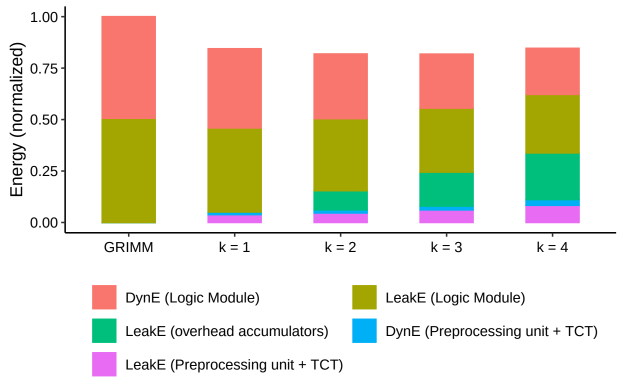


Fig. 14. Energy breakdown of the components on the logic-in-memory layer

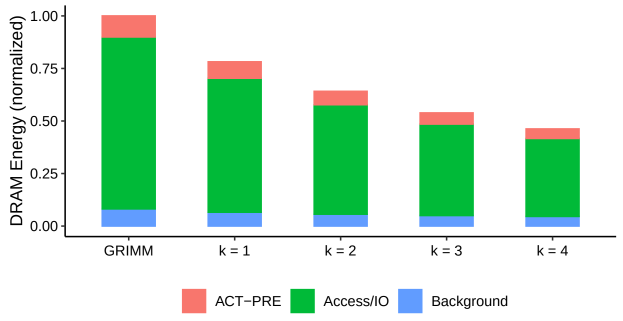


Fig. 13. DRAM energy breakdown

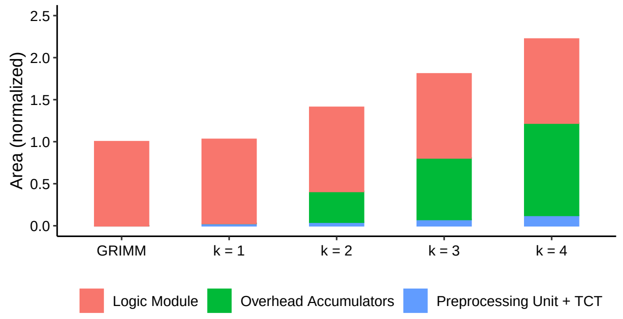


Fig. 15. Area breakdown of the components on the logic-in-memory layer

the logic-in-memory layer. For $k$ = 4, the area of the logic-in-memory layer nearly doubles compared to the GRIM-Filter. For 3D-stacked memories, the logic-in-memory layer has the same area footprint as the memory layers for manufacturing reasons [41]. Therefore, the logic-in-memory layer introduces unused area that can easily accommodate the extra overhead of ALPHA. Similarly, the latency and energy overhead of the extra logic on the logic-in-memory layer increases with increasing the value of $k$. Despite this increase in the overheads for a larger $k$, the overall runtime and energy consumption decreases compared to a smaller $k$ (cf. Fig. 11 and Fig. 12). It is worth mentioning that $k$ = 1 introduces very small area overhead compared to the GRIM-Filter. For $k$ = 1, the preprocessing unit and the TCT occupy an additional 2.75% of area.

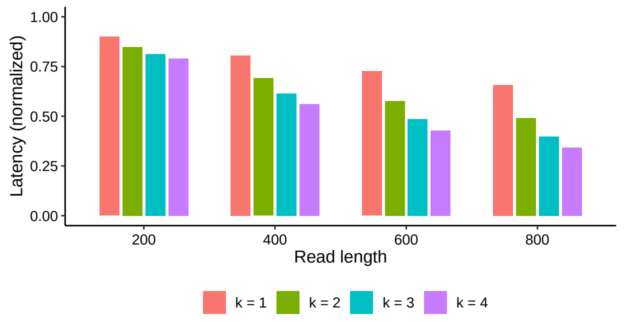


Fig. 16. Runtime for different read lengths averaged over all data sets listed in Table 1. All results are normalized to the GRIM-Filter.

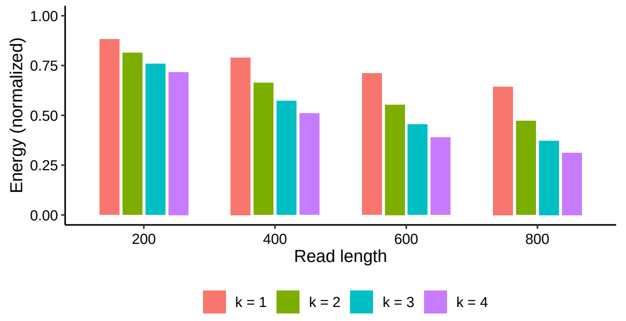


Fig. 17. Energy consumption for different read lengths averaged over all data sets listed in Table 1. All results are normalized to the GRIM-Filter.

### 5.5 Sensitivity to read length

Sequencing technologies produce read lengths in the range of 100 to several million base pairs [42], [43]. It is predicted that read lengths greater than 100,000 basepairs will be required not only to generate high quality genome assemblies but also to detect and resolve structural variation [44]. In order to determine the impact of read length on the overall runtime and energy consumption, we carried out some experiments by sweeping the read length from 200 to 800. Fig. 16 and Fig. 17 show how varying the read length affects the normalized runtime and energy consumption, respectively. These results indicate that ALPHA consistently outperforms the GRIM-Filter for all ranges of read lengths. As the probability of the occurrence of repeated tokens is high for the longer read lengths compared to the shorter ones, ALPHA provides significant improvements in both performance as well as energy consumption for the longer reads, while the benefits are less prominent for shorter reads.

For the previously reported results, ALPHA is tested on the benchmarks with short read lengths (i.e., 200 to 800 base pairs), which are the same benchmarks used in the evaluation of the GRIM-filter. This allows a direct comparison of the approaches. To highlight the benefits of ALPHA for the longer read lengths, we tested ALPHA on a genome with around 12K base pairs. Specifically, we used "m54238_180901_011437.Q20"[3]. As expected, the longer reads result in a considerably higher percentage of token repetition. On average, the tokens repetition in each read is around 92%, compared to around (14%, 24%, 40%, 52%, 61%) for read lengths of (200, 400, 800, 1200 and 1600) respectively. Therefore, we expect proportional benefits in the latency and the energy results for the longer reads compared to the shorter ones.

3. https://github.com/genome-in-a-bottle/giab_data_indexes/blob/master/AshkenazimTrio/sequence.index.AJtrio_PacBio_CCS_15kb_10022018.HG002

## 6 Conclusions

This paper presents an efficient algorithm-hardware co-design of a pre-alignment filter that outperforms the state-of-the-art GRIM-Filter in terms of energy and runtime. We propose a low overhead preprocessing unit that scans the input reads to find repetitive tokens and their count in an input genome. Our analysis of 10 human genomes showed that input reads most often have repetitive tokens. We exploit this fact to reduce the number of memory accesses and arithmetic operations in the filtering process. For a given token that is repeated $n$ times, the GRIM-Filter requires $n$ memory accesses and $n$ increment operations, while our proposed approach requires single memory access and a single add operation, thanks to the token count table that stores the repetition information. This significant reduction in the number of memory accesses and the number of arithmetic operations improves the runtime and energy consumption of the pre-alignment filter by an average of 19-44% and 21-49%, respectively. Note that these improvements are on top of the end-to-end read mapping improvement reported by the GRIM-Filter, i.e., $1.81 - 3.65\times$.

We believe the energy consumption of ALPHA can be further reduced by replacing the power-hungry DRAM with emerging memory technologies such as the racetrack memory [45]. However, since these technologies have their limitations, appropriate changes are required to the ALPHA architecture to address them. We are planning to explore this from different perspectives in future research, aiming at further reducing energy consumption.

## Acknowledgments

This work was partially funded by the German Research Council (DFG) through the DART-HMS project HA 9123/1-1 (project number 437232907) and the TraceSymm project CA 1602/4-1 (project number 366764507).

## References

[1] Illumina Inc., "Hiseq X$^{TM}$ series of sequencing systems," https://emea.illumina.com/content/dam/illumina-marketing/documents/products/datasheets/datasheet-hiseq-x-ten.pdf, accessed: 2020-06-19.
[2] M. Vasimuddin, S. Misra, H. Li, and S. Aluru, "Efficient architecture-aware acceleration of bwa-mem for multicore systems," 2019, pp. 314–324.
[3] T. F. Smith and M. S. Waterman, "Identification of Common Molecular Sub sequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, March 1981.
[4] S. B. Needleman and C. D. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443 – 453, 1970.
[5] H. Li, "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences," *Bioinformatics*, vol. 32, no. 14, pp. 2103–2110, 2016.
[6] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 05 2018.

[7] C. Jain, S. Misra, H. Zhang, A. Dilthey, and S. Aluru, "Accelerating sequence alignment to graphs," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 451–461.

[8] N. Ahmed, J. Lévy, S. Ren, H. Mushtaq, K. Bertels, and Z. Al-Ars, "Gasal2: a gpu accelerated sequence alignment library for high-throughput ngs data," *BMC Bioinformatics*, vol. 20, 2019.

[9] Y. Turakhia, K. Zheng, G. Bejerano, and W. Dally, "Darwin: A hardware-acceleration framework for genomic sequence alignment," *bioRxiv*, 2017.

[10] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "GateKeeper: a New Hardware Architecture for Accelerating Pre-alignment in DNA Short Read Mapping," *Bioinformatics*, vol. 33, no. 21, pp. 3355–3363, 2017.

[11] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating Read Mapping with FastHASH," *BMC Genomics*, vol. 14, no. S1, 2013.

[12] 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, 2016.

[13] J. Kim, D. Senol Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping using Processing-in-memory Technologies," *BMC Genomics*, vol. 19, no. 2, 2018.

[14] A. Nag, C. N. 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*, ser. MICRO '52, 2019, p. 334346.

[15] S. Banerjee, M. El-Hadedy, J. Lim, Z. Kalbarczyk, D. Chen, S. Lumetta, and R. Iyer, "ASAP: Accelerated Short-Read Alignment on Programmable Hardware," *IEEE Transactions on Computers*, vol. 68, no. 3, p. 331346, Mar. 2019.

[16] M. Alser, T. Shahroodi, J. Gmez-Luna, C. Alkan, and O. Mutlu, "Sneakysnake: a fast and accurate universal genome pre-alignment filter for cpus, gpus and fpgas," *Bioinformatics*, 12 2020.

[17] P. Chen, C. Wang, X. Li, and X. Zhou, "Accelerating the next Generation Long Read Mapping with the FPGA-Based System," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 11, no. 5, p. 840852, Sep. 2014.

[18] D. Cali, G. Kalsion, Z. Bingöl, C. Firtina, L. Subramanian, J. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, A. Nori, A. Scibisz, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, "Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis," in *Proceedings - 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020*, ser. Proceedings of the Annual International Symposium on Microarchitecture, MICRO, October 2020, pp. 951–966.

[19] 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*, ser. ISCA '18, 2018, p. 6982.

[20] M. Alser, Z. Bingöl, D. S. Cali, J. S. Kim, S. Ghose, C. Alkan, and O. Mutlu, "Accelerating genome analysis: A primer on an ongoing journey," *IEEE Micro*, vol. 40, no. 5, pp. 65–75, 2020.

[21] D. Kim, J. Paggi, C. Park, C. Bennett, and S. Salzberg, "Graph-based Genome Alignment and Genotyping with HISAT2 and HISAT-genotype," *Nature Biotechnology*, vol. 37, no. 8, pp. 907–915, 8 2019.

[22] F. Hach, I. Sarrafi, F. Hormozdiari, C. Alkan, E. Eichler, and C. Sahinalp, "Mrsfast-ultra: A compact, snp-aware mapper for high performance sequencing applications," *Nucleic acids research*, vol. 42, no. W1, pp. W494–W500, 05 2014.

[23] 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, 2012.

[24] C. Alkan, J. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. Kitzman, C. Baker, M. Malig, O. Mutlu, C. Sahinalp, R. Gibbs, and E. Eichler, "Personalized Copy Number and Segmental Duplication Maps using Next-generation Sequencing," *Nature genetics*, vol. 41, no. 10, pp. 1061–1067, October 2009.

[25] S. Rumble, P. Lacroute, A. Dalca, M. Fiume, A. Sidow, and B. Michael, "SHRiMP: Accurate Mapping of Short Color-space Reads," *PLoS Computational Biology*, vol. 5, no. 5, 2009.

[26] F. Hormozdiari, F. Hach, C. Sahinalp, E. Eichler, and C. Alkan, "Sensitive and Fast Mapping of Di-base Encoded Reads," *Bioinformatics*, vol. 27, no. 14, pp. 1915–1921, 05 2011.

[27] D. Weese, A.-K. Emde, T. Rausch, A. Döring, and K. Reinert, "RazerS – Fast Read Mapping with Sensitivity Control." *Genome research*, vol. 19, no. 9, pp. 1646–1654, 2009.

[28] K. Brinda, M. Sykulski, and G. Kucherov, "Spaced Seeds Improve k-mer-based Metagenomic Classification," *Bioinformatics*, vol. 31, no. 22, pp. 3584–3592, 07 2015.

[29] S. Girotto, M. Comin, and C. Pizzi, "Efficient Computation of Spaced Seed Hashing with Block Indexing," *BMC Bioinformatics*, vol. 19, 11 2018.

[30] G. Auwera, M. Carneiro, C. Hartl, R. Poplin, G. del Angel, A. Levy-Moonshine, T. Jordan, K. Shakir, D. Roazen, J. Thibault, E. Banks, K. Garimella, D. Altshuler, S. Gabriel, and M. DePristo, "From fastq data to high-confidence variant calls: The genome analysis toolkit best practices pipeline," *Current protocols in bioinformatics*, vol. 11, pp. 11.10.1–11.10.33, 10 2013.

[31] "Igsr: The international genome sample resource," https://www.internationalgenome.org/, accessed: 2020-06-19.

[32] D. Altshuler, G. Abecasis, D. Bentley, A. Chakravarti, A. Clark, P. Donnelly, E. Eichler, P. Flicek, S. Gabriel, R. Gibbs, E. Green, M. Hurles, B. Knoppers, J. Korbel, E. Lander, C. Lee, H. Lehrach, E. Mardis, and G. Consortium, "An integrated map of genetic variation from 1,092 human genomes," *Nature*, vol. 491, pp. 56–65, 11 2012.

[33] C. Weis, M. Jung, and N. Wehn, "3d stacked dram memories," in *Handbook of 3D Integration: Design, Test, and Thermal Management*, P. D. Franzon, E. J. Marinissen, and M. S. Bakir, Eds., 2016, vol. 4, ch. 8.

[34] J. Kim and Y. Kim, "HBM: Memory Solution for Bandwidth-hungry Processors," in *2014 IEEE Hot Chips 26 Symposium (HCS)*, Aug 2014, pp. 1–24.

[35] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong, "25.2 a 1.2v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 432–433.

[36] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-efficient DRAM Caching via Software/Hardware Cooperation," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 1–14.

[37] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469–480.

[38] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 3–14.

[39] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, "Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 41–54.

[40] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 140–143, July 2015.

[41] G. Loh, "A Register-file Approach for Row Buffer Caches in Die-stacked DRAMs," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 351–361.

[42] S. Jünemann, F. Sedlazeck, K. Prior, A. Albersmeier, U. John, J. Kalinowski, A. Mellmann, A. Goesmann, A. von Haeseler, J. Stoye, and D. Harmsen, "Updating Benchtop Sequencing Performance Comparison," *Nature Biotechnology*, vol. 31, pp. 294–296, 04 2013.

[43] M. Quail, M. Smith, P. Coupland, T. Otto, S. Harris, T. Connor, A. Bertoni, H. Swerdlow, and Y. Gu, "A Tale of Three Next Generation Sequencing Platforms: Comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers," *BMC genomics*, vol. 13, p. 341, 07 2012.

[44] M. Chaisson, R. Wilson, and E. Eichler, "Genetic Variation and the

De Novo Assembly of Human Genomes," *Nature Reviews. Genetics*, vol. 16, no. 11, pp. 627–640, 10 2015.

[45] R. Bläsing, A. A. Khan, P. C. Filippou, C. Garg, F. Hameed, J. Castrillon, and S. S. Parkin, "Magnetic racetrack memory: From physics to the cusp of applications within a decade," *Proceedings of the IEEE*, vol. 108, no. 8, pp. 1303–1321, 2020.

**Fazal Hameed** Fazal Hameed received his Ph.D. (Dr.-Ing.) degree in computer science from the Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, in 2015. He joined the chair for Compiler Construction at the TU Dresden (Dresden, Germany) as Post-doctoral researcher in March 2016. Before, he worked on a similar position at the Chair of Dependable and Nano Computing (CDNC) Karlsruhe Institute of Technology (KIT), Germany. He is currently affiliated with Institute of Space Technology, Islamabad, Pakistan. He mainly works in the architecture group with a focus on memories. Mr. Hameed was a recipient of the CODES+ISSS 2013 Best Paper Nomination for his work on DRAM cache management in multicore systems. He has served as an External Reviewer for major conferences in embedded systems and computer architecture.

**Asif Ali Khan** Asif Ali Khan is currently pursuing his Ph.D. at the Chair for Compiler Construction in the Computer Science Department of the TU Dresden, Germany. His research interests include Computer architecture, heterogeneous memories, and compiler support for the memory system. Currently, Asif's research mainly focuses on exploring the emerging nonvolatile memory technologies in the memory subsystems and their optimizations for various metrics.

**Jeronimo Castrillon** Jeronimo Castrillon is a professor in the Department of Computer Science at the TU Dresden, where he is also affiliated with the Center for Advancing Electronics Dresden (CfAED). He is the head of the Chair for Compiler Construction, with research focus on methodologies, languages, tools and algorithms for programming complex computing systems. He received the Electronics Engineering degree from the Pontificia Bolivariana University in Colombia in 2004, the master degree from the ALaRI Institute in Switzerland in 2006 and the Ph.D. degree (Dr.-Ing.) with honors from the RWTH Aachen University in Germany in 2013. Prof. Castrillon served in the executive committee of the ACM "Future of Computing Academy" from 2017 to 2019.