Reconfigurable Acceleration of Short Read Mapping with Biological Consideration

Ho-Cheung Ng[1], Izaak Coleman[2], Shuanglong Liu[3] and Wayne Luk[1]

[1] Department of Computing, Imperial College London, UK {h.ng16, w.luk}@imperial.ac.uk

[2] Department of Systems Biology, Columbia University, USA ic2465@cumc.columbia.edu

[3] School of Physics and Electronics, Hunan Normal University, China liu.shuanglong@hunnu.edu.cn

ABSTRACT

Existing FPGA accelerators for short read mapping often fail to utilize the complete biological information in sequencing data for simple hardware design, leading to missed or incorrect alignment. Furthermore, their performance may not be optimized across hardware platforms. This paper proposes a novel alignment pipeline that considers all information in sequencing data for biologically accurate acceleration of short read mapping. To ensure the performance of the proposed design optimized across different platforms, we accelerate the memory-bound operations which have been a bottleneck in short read mapping. Specifically, we partition the FM-index into buckets. The length of each bucket is equal to an optimal multiple of the memory burst size and is determined through data-driven exploration. A tool has been developed to obtain the optimal parameters of the design for different hardware platforms to enhance performance optimization. Experimental results indicate that our design maximizes alignment accuracy compared to the state-of-the-art software Bowtie, mapping reads 4.88 × as fast. Compared to the previous hardware aligner, our achieved accuracy is 97.7 % which reports 4.48 M more valid alignments with a similar speed.

ACM Reference Format:
Ho-Cheung Ng, Izaak Coleman, Shuanglong Liu and Wayne Luk. 2021. Reconfigurable Acceleration of Short Read Mapping with Biological Consideration. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '21), February 28-March 2, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3431920.3439280

1 INTRODUCTION

Technological innovation and the declining cost of next-generation sequencing (NGS) have led to an explosion in the quantity of genomic data. A critical prior step of lots of downstream analysis of genomic data is short read mapping, where millions of short DNA fragments generated by an NGS machine are aligned to a reference genome [16]. However, given that the throughput of NGS technology is outstripping Moore's Law, the time required for existing software aligners to map NGS data is becoming prohibitive [21, 28]. For example, in the identification of oncogenesis (cancer cells) for cancer diagnosis, software aligners are unable to meet the demand for large scale clinical adaption, where patient turnaround time is critical. Therefore, their acceleration would bridge the gap between alignment research and practice, allowing these diagnosis techniques to become part of routine clinical procedures [22].

FPGA technology is a promising candidate to accelerate short read mapping [24, 27]. Its highly-parallel bit-oriented architecture has been leveraged to accelerate different mapping algorithms. In particular, suffix-trie based alignment, which often uses the FM-index, has been accelerated in many previous efforts due to its popularity in leading software aligners such as Bowtie [16] and BWA [20].

However, FPGA-accelerated short read mapping is rarely adopted by genomic and medical scientists due to two reasons:

- Most of them are **platform-dependent**, i.e. their design methodology and optimizations target a single platform. As a result, **when migrating across platforms, the speedup may not hold**. Because of the cost and privacy concerns, sometimes local clusters are preferable [18]. The lack of performance portability reduces the attraction of FPGA technology to genomic scientists, allowing its expense and requirement of specialized knowledge to outweigh its potential benefits. For example, Arram *et al.* [2] propose an FM-index-based accelerator that is 18.1 × faster than software. However, the design targets Maxeler MaxWorkstation and the performance analysis is unlikely to generalize across platforms. Other accelerators for short read mapping such as [6, 9] also suffer from this issue.
- Existing accelerators have failed to utilize the complete biological information to simplify the hardware design. Thus far, only A, T, C, G characters have been handled by accelerators. **Ambiguous characters (N characters) and quality metric are usually not processed** for hardware simplicity. Moreover, their implementations may not be strictly consistent with the algorithms in popular software aligners. The under-utilization of the available information and their inconsistent modeling of short read alignment limit the biological validity and reproducibility of the alignments they output. This in turn **diminishes the confidence of employing hardware aligners in real-life applications, particularly in cancer diagnosis as an invalid alignment result can be disastrous for the cancer patients.**

To address the above issues, we propose a novel approach that employs the complete biological information including quality metric and ambiguous characters to accelerate suffix-trie based short read alignment. We exploit a **multi-configuration pipeline** that

FPGA '21, February 28-March 2, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8218-2/21/02...$15.00
https://doi.org/10.1145/3431920.3439280

aligns reads with different edit-distance-based filters using **run-time reconfiguration**. These filters, which include exact-match, one-mismatch, two-mismatch, and three-mismatch filters, are arranged in ascending edit distance order. Unlike previous designs targeting an individual platform, we accelerate the memory access in computations to aid performance portability, by partitioning the suffix-trie into buckets with the size equal to multiple of the memory burst size. This optimization approach covers FPGA platforms in different generations since memory burst size is a property of many memory-FPGA systems. Finally, we design a fully automated tool that predicts the optimal bucket size - and other parameters - to customize the proposed aligner for platforms beyond this work. The main contributions of this work are the following:

- A novel optimization technique which employs the suffix-trie data structure, particularly FM-index to improve alignment speed across platforms (Section 4);
- A novel and customizable architecture that consists of a multi-filter alignment pipeline to accelerate short read alignment. It utilizes the complete biological information present in sequencing data to maximize alignment accuracy (Section 3 and Section 5);
- An automatic tool that determines the optimal parameters for aligner customization while maintaining maximum alignment accuracy (Section 6).

2 BACKGROUND AND RELATED WORK

This section provides an overview of short read mapping by demonstrating the alignment process with an example read file and reference. We also elaborate suffix-trie based alignment, particularly FM-index as it is chosen for acceleration. This information will help clarify the reconfigurable architecture and algorithm optimizations presented in the later section.

2.1 Alignment Overview

Short read mapping is the process of ordering the nucleotide bases (A, T, C, G) of a DNA sequence within the query cells. When the query cells are given to the NGS machine, the sequenced data generated are comprised of millions of short DNA fragments. These short fragments are known as short reads, where the position and orientation of the reads with respect to the original, long DNA strand are lost. Based on the assumption that DNA sequences within species are highly similar, the sequenced DNA can be reconstructed by identifying the location of reads in a known reference genome. Figure 1 displays an example of aligning reads to a short reference.

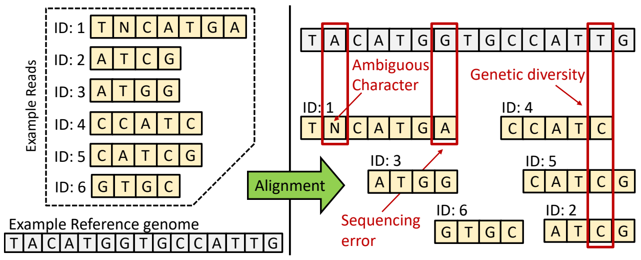
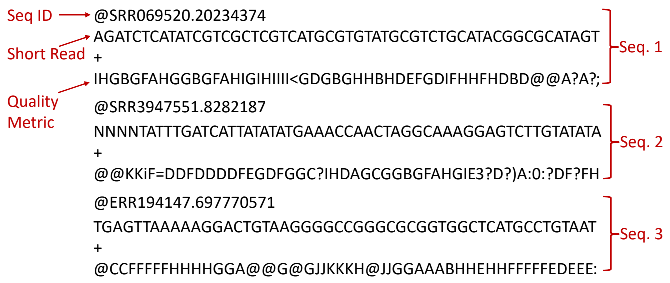


Figure 1: Example of aligning reads to a reference.


Seq ID
Short Read
Quality Metric
Seq. 1
Seq. 2
Seq. 3

```
@SRR069520.20234374
AGATCTCATATCGTCGCTCGTCATGCGTGTATGCGTCTGCATACGGCGCATAGT
+
IHGBGFAHGGBGFAHIGIHIIII<GDGBGHHBHDEFGDIFHHFHDBD@@A?A?;
@SRR3947551.8282187
NNNNTATTTGATCATTATATATGAAACCAACTAGGCAAAGGAGTCTTGTATATA
+
@@KKiF=DDFDDDDFEGDFGGC?IHDAGCGGBGFAHGIE3?D?)A:0:?DF?FH
@ERR194147.697770571
TGAGTTAAAAAGGACTGTAAGGGGCCGGGCGCGGTGGCTCATGCCTGTAAT
+
@CCFFFFFHHHHGGA@@G@GJJKKKH@JJGGAAABHHEHHFFFFFEDEEE:
```

Figure 2: Snippet of an example FASTQ file.

The mapping process is intrinsically a pattern matching problem which can be accelerated using a well-established method like indexing a reference genome. Although the time needed to create an index for the human reference genome can be up to an hour, the reference human genome version changes infrequently. This time can be amortized over the abundant amount of alignment jobs.

The read data, on the other hand, is normally stored in FASTQ format using ASCII encoding. Figure 2 shows a snippet of an example FASTQ file which consists of several short reads. Basically, a FASTQ file uses four lines per sequence. The first line is the sequence identifier while the second line is the DNA short read. The third line begins with a '+' character and this line only stores optional information. The fourth line carries important information that encodes the quality metric for the short read. Each quality value is associated with a nucleotide in the same position of line 2, and each value has a range between 33 ('!' in ASCII) denoting the lowest quality and 75 ('K') denoting the highest quality.

The quality metric is critical to producing correct alignment results, as a lower quality value indicates a higher likelihood of a sequencing error. The alignment policy should filter out or indicate reads with sequencing errors by analyzing the quality metric. However, the quality metric is usually neglected by previous hardware designs for performance consideration and hardware simplification. Since each quality value requires 6 bits while each nucleotide only needs 2 bits to represent, the consideration of quality metric tremendously increases the amount of data to process per nucleotide. Without this metric, previous FPGA designs can produce biologically incorrect alignment results which are unacceptable in real-life sequencing applications. Moreover, ambiguous characters (N characters), which are illustrated in the short read of the second sequence, are common in read data. Yet for the same reason, reads consisting of N characters are generally discarded to improve the alignment speed.

2.2 FM-index

FM-index [10] is a commonly used indexing algorithm in state-of-the-art alignment software such as Bowtie. It combines the properties of suffix array with the Burrows-Wheeler transform (BWT) [4]. This data structure provides an efficient mechanism to perform substring matching of a pattern P in a long reference sequence R.

The generation of FM-index begins with the computation of BWT of the reference genome R, i.e. $BWT(R)$. First, R is terminated with a unique character: '\$', which is lexicographically the smallest value. Then, all the rotations of the text are obtained and sorted correspondingly (sorted rotations). The suffix array can be obtained by considering the characters before '\$' in each entry of the rotation list. $BWT(R)$ can be formed by extracting and concatenating the last characters of all the entries on the sorted list. Table 1a demonstrates the derivation of BWT with an example reference genome R = GCTAT. The string preceding the '\$' sign in the sorted rotations forms the suffix array (SA), which indicates the position of each possible suffix in the original string.

Table 1: (a) Example of deriving the suffix array and BWT of reference sequence *R*. (b) $i(x)$ and $c(n, x)$ functions for the sequence *R*.

(a)

R = GCTAT\$		
Index n	SA	Sorted Rotations
0	5	\$GCTAT
1	3	**AT**\$GCT
2	1	**CTAT**\$G
3	0	**GCTAT**\$
4	4	**T**\$GCTA
5	2	**TAT**\$GC
$BWT(R)$ = TTG\$AC		

(b)

$c(n, x)$

Index n	A	C	G	T
0	0	0	0	0
1	0	0	0	1
2	0	0	0	2
3	0	0	1	2
4	0	0	1	2
5	1	0	1	2
6	1	1	1	2

$i(x)$ $\{1, 2, 3, 4\}$

After having the suffix array, R and $BWT(R)$ are used to form the i and c tables. For each element x of the alphabet of R, $i(x)$ is defined as the index of its first occurrence in sorted R. For each index n in $BWT(R)$ and for each character x in the alphabet, $c(n, x)$ stores the number of occurrences of x in $BWT(R)$ in the range $[0, n-1]$. Table 1b illustrates the $i(x)$ and $c(n, x)$ tables for the sequence R.

```
Input: Pattern $P$, $c()$, $i()$, $SA$ of Reference $R$
Result: The starting locations $Loc$ of $P$ in $R$
1  $l = |P| - 1$
2  $(top, bottom) = (0, c(|R|, P[l]))$
3  for $j = l$ to 0 do
4      $top = c(top, P[j]) + i(P[j])$
5      $bottom = c(bottom, P[j]) + i(P[j])$
6  end
7  if $top < bottom$ then
8      for $j = 0$ to $top - bottom$ do
9          $Loc[j] = SA[top + j]$
10     end
11 end
```

Algorithm 1: Exact substring matching using FM-index.

Algorithm 1 describes the procedure of searching a pattern P in the reference R. The pointers *top* and *bottom* are first initialized with the first and last indices of the $c(n, x)$ table respectively. The search begins with processing a character at a time, starting with the last character of P. The final results of *top* and *bottom* are the range of indices in SA that contains P as the prefix, which can be subsequently converted into the coordinates in R. If $top \geq bottom$, P does not occur in R. Note that the time complexity of locating a pattern in the reference genome is linear with respect to $|P|$ instead of $|R|$.

Finally, FM-index can be extended to support mismatch alignment with backtracking [16]. Essentially, a stack is maintained to store the current search state when a mismatch happens. A different character is then attempted to align with the reference genome. The state is restored when the number of mismatches exceeds the permitted value. Another untested character is used and attempted to perform matching again.

2.3 Related Work

Previous efforts on FPGA acceleration of short read alignment are elaborated comprehensively in surveys [24, 27]. Essentially, many of the existing accelerators are partly or solely based on the suffix-trie method. For example, Fernandez *et al.* [9] introduced the use of FM-index to accelerate exact and approximate string matching on Convey HC-1. Arram *et al.* [2] also proposed an FM-index-based aligner on Maxeler MaxWorkstation to support alignment up to two mismatches. Some FPGA designs which adopted the seed-and-expansion technique such as [1, 26] employed FM-index to identify the location of the seed. However, their underlying solutions to resolve the memory bottleneck target a specific platform, and it is unclear how their solutions benefit other platforms. Other accelerators for short read mapping such as [6, 7] also suffer from this issue.

Existing accelerators also suffer from the problem of biological validity and reproducibility, as the quality metric and ambiguous characters in the read data are usually neglected. There are a few accelerators that work around this problem by proposing a software/hardware co-design to an existing software framework. Pico Computing [7] developed a BWT accelerator that ties into existing BWA software, which enables the processor to perform tasks that it is optimized for. The authors in [6] introduced a similar approach where the seeding stage of BWA-MEM [19] runs on the FPGA for acceleration while retaining the expansion stage on CPU. A similar approach is adopted by related efforts [13] and [14]. Despite the speedup provided by these accelerators, the complete biological information in the read data and biological significance of the alignment results are not covered or discussed.

Our work differs from previous ones by providing a general method that accelerates alignment with suffix-trie. A detailed investigation of memory throughput against bucket size and index structure enables accelerator optimization from the memory access perspective. Complete biological information is considered and hence the alignment results are comparable to Bowtie to ensure biological validity and reproducibility.

3 RECONFIGURABLE HARDWARE ARCHITECTURE

We propose a general architecture for alignment acceleration that exploits the reconfigurability of an FPGA. Distinct hardware implementations are executed on the FPGA in a pipeline, where each implementation is composed of a homogeneous array of ***computational modules***. Each module is functionally equivalent to ***one specific alignment algorithm***. Runtime reconfiguration is used to load individual implementation onto the FPGA in order, and data from the previous configuration (or initial data from the host) are processed concurrently by the module array.

3.1 Motivation for Reconfiguration

Previous efforts that accelerate alignment with FPGA usually rely on a ***static architecture***. Essentially, the target device is configured with an implementation that is functionally equivalent to ***multiple alignment algorithms***. For example, the FPGA implementation in [9] is composed of different alignment modules to perform filtered search with FM-index, where reads unaligned by one algorithmic step (***filter***) are directed to the following one. Respective modules are grouped to form an exact match, one-mismatch, and two-mismatch filter and are connected to form one implementation. Although a static configuration of these filters as a single implementation eliminates the time for reconfiguration, this approach has its limitations which reduce the overall performance and subsequently nullify the benefit of discarding reconfiguration.

Significant Amount of Data Hazards — A typical alignment workflow is composed of multiple algorithmic steps where data from the current step rely on the results from the previous one. In software such as Bowtie, a read is only handled by the mismatch subroutine if an exact match is failed, i.e. a ***filtered search*** with exact match then mismatches. This incurs numerous data hazards for a statically configured circuit. Since all the modules for different steps are mapped onto FPGA, data hazards reduce the FPGA efficiency because some modules are left idle occasionally.

Distinct Module Latency and Limited Resources — Different alignment modules require a distinct number of cycles to finish processing a read. To maintain a balanced pipeline in a static configuration, some modules are replicated to match the latency. For example, when different filters in Bowtie are mapped onto FPGA with a static configuration, the mismatch module must be duplicated more to even out the throughput of the exact match modules. This can be challenging and even impossible due to the limited resources available on FPGA.

Flexibility of Reconfigurable Architecture — Alignment parameters can be different depending on the experimental requirements. A runtime reconfigurable architecture can provide users better control over these parameters where filters can be re-arranged, removed, or added straightforwardly. It also improves performance portability in which the module counts within a single implementation of a filter can increase/decrease subject to the availability of FPGA resources.

3.2 Proposed Architecture

To guarantee the biological validity and reproducibility of the alignment results, the proposed architecture follows a similar workflow compared to state-of-the-art software Bowtie. FM-index is used to implement filtered search on FPGA with runtime reconfiguration. As illustrated in Figure 3, the filters are arranged in a pipeline with the order: exact match, one-mismatch, two-mismatch, and three-mismatch where each filter is composed of a homogeneous array of modules. The FM-index, which is of a few GB, is stored on the external DDR memory attached to the FPGA. The number of times a module can be replicated is given by

$$M_j = \min\left(\frac{R}{r_j}, N_{DIMM}\right) \quad (1)$$

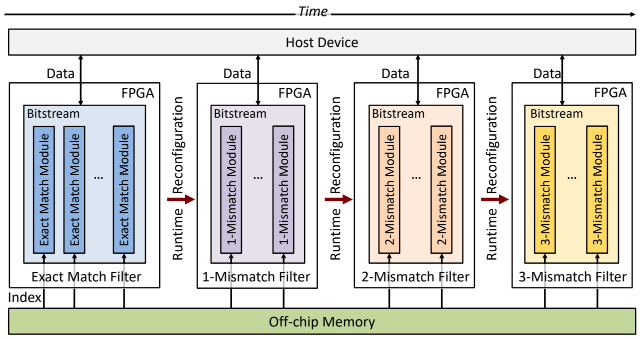


Figure 3: Alignment pipeline of the proposed architecture with runtime reconfiguration. Each filter is composed of homogeneous array of modules. The module counts within a filter is given by Equation (1)**.**

where M_j is the module counts for filter j, R is the total available resources on the target FPGA, and r_j is the amount of resources required by the module. N_{DIMM} represents the number of DIMMs available in external memory. Since a memory channel is connected to a module within the filter, the number of DIMMs can also restrict the number of modules that can be replicated on FPGA. Therefore, in the next section, we introduce novel techniques such as bucketing to optimize memory access and maximize alignment performance.

The overall performance of this architecture can be modeled by (2), where T is the alignment time for all filters, N_j is the number of reads to process and t_j is the time to process a read by a particular filter. The overhead of this architecture is the reconfiguration time t_{re} and the data communication overhead t_c between the host. For a typical alignment process with more than a million reads to align, these two numbers are negligible as the alignment time is the dominant factor. Note that the alignment can start as soon as the first read arrives from the host, so t_c can be hidden by the alignment latency.

$$T = \sum_j \left(t_{re} + \max\left(t_c, \frac{N_j t_j}{M_j}\right)\right) \quad (2)$$

4 FM-INDEX OPTIMIZATION FOR FPGA

According to Algorithm 1, the bottleneck of FM-index based alignment is memory access where a character search involves access to a random memory location. Therefore, in this section, we detail the techniques that utilize the complete memory bandwidth and improve the performance of FM-index.

4.1 Bucketing

Index Restructuring — Although FM-index is a space-efficient data structure that permits fast substring matching, when indexed, the human reference genome is around 51 GB. This is often far larger than the capacity of off-chip memory. To reduce the memory footprint, we store a subset of $c(n, x)$ of FM-index, while substituting the remaining entries with a portion of the original BWT. Specifically, we sample every d entry of $c()$ and pack the BWT in the range of every d and $d - 1$ alongside, forming a ***bucket***. During a character search, the missing entries in $c()$ can be recalculated on-the-fly using the BWT B. Figure 4 shows the final restructured FM-index F and Algorithm 2 demonstrates the corresponding procedure for substring matching.

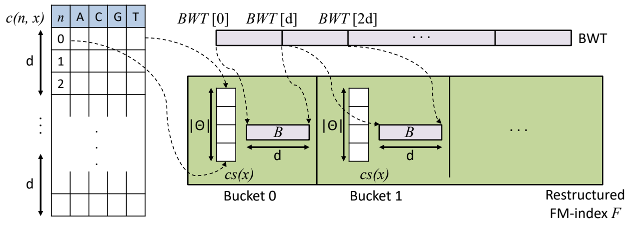


Figure 4: The proposed restructured FM-index.

```
Input: Pattern $P$, Restructured FM-index $F$, $SA$ of Reference $R$
Result: The starting locations $Loc$ of $P$ in $R$
1  $l = |P| - 1$
2  $(top, bottom) = (0, c(|R|, P[l]))$ //Assume we always store $c(|R|, x)$
3  for $j = l$ to 0 do
4      $top = F[top/d].cs(P[j]) +$
5          $Count(P[j],\ F[top/d].B,\ top\%d) + i(P[j])$
6      $bottom = F[bottom/d].cs(P[j]) +$
7          $Count(P[j],\ F[bottom/d].B,\ bottom\%d) + i(P[j])$
8  end
9  if $top < bottom$ then
10     for $j = 0$ to $top - bottom$ do
11         $Loc[j] = SA[top + j]$
12     end
13 end
14 Function Count($char, B, position$):
15     $cnt = 0$
16     for $j = 0$ to $position$ do
17         if $char == B[j]$ then
18             $cnt{+}{+}$
19         end
20     end
21     return $cnt$
```

Algorithm 2: Algorithm for exact substring matching using restructured FM-index. Note some notations are specified in Figure 4.

Given that the human reference genome R_{human} consists of 3 billion characters, the storage of each $c()$ entry requires 32 bits. Hence, subset storage of $c()$ table can significantly decrease the index size, especially when each BWT character only requires 2 bits for alphabet $\Theta = \{A, T, C, G\}$. The memory usage required by the customized FM-index is reduced to:

$$\begin{aligned} &\text{Sampled } c(n,x) \text{ Size} + \text{Sampled } BWT \text{ Size} \\ =&\frac{|R_{human}| \times (32bit)}{d} \times |\Theta| + |R_{human}| \times (\log_2(|\Theta|)bit) \\ =&\frac{3.2GB \times 4}{d} \times 4 + 3.2GB \times \frac{2}{8} = (\frac{51.2}{d} + 0.8)GB \end{aligned} \tag{3}$$

Sampling Distance d — The value of d plays an important role in the memory access efficiency. First, it affects the final size of the restructured index. More importantly, it changes the bucket size and influences alignment performance. According to Algorithm 2, every character search of P requires streaming of a bucket from the external memory. To enable better utilization of the memory bandwidth, d should be set based on Equation (3) such that the final bucket size is a multiple of the memory burst size. For typical DDR4 memory with a burst size 64 B, $d = 192$. The final index size becomes 1.06 GB.

4.2 n-step FM-index

Based on Algorithm 1, the search of a pattern P takes $|P|$ steps to obtain the final interval in the SA. Chacón *et al.* [5] introduce an alternative arrangement of FM-index that reduces the steps, hence improving the execution time. Essentially, this arrangement alters the index structure such that n symbols in P are matched per step during the backward search. Subsequently, the number of steps required to obtain the final interval between top and $bottom$ become $\frac{|P|}{n}$. For more information about n-step FM-index, please refer to [5].

This reduction of search steps, however, comes at the expense of larger bucket and index size, as well as increased computational complexity per step. The index is now adjusted to store information about n concatenated symbols each time instead of 1 symbol. For example, if $n == 2$, each row in $i()$ and $c()$ table contains the occurrence values for alphabets $\Theta = \{AA, AC, ..., GC, GG\}$. This increases the entries count and consequently the index and bucket size, and also d.

In spite of the larger bucket size where more data are read per memory access, reading more bursts generally increases the effectiveness of memory transfer. In Section 6.1, we provide an exploration of the step n and bucket size versus the memory bandwidth to understand their implications on alignment performance.

5 DESIGN OF THE ALIGNMENT ACCELERATOR

This section presents the FPGA accelerated aligner where details and collaboration between modules are described. In particular, the modules for handling the quality metric and ambiguous characters are elaborated. The techniques used to further improve the alignment performance are also discussed.

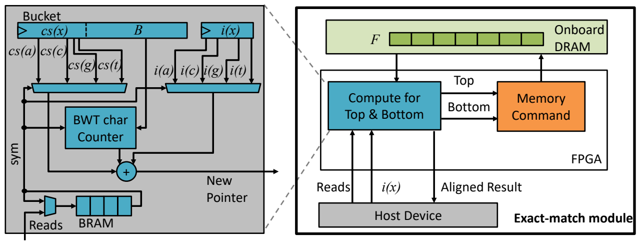


Figure 5: Simplified top-level diagram for exact-match module. The diagram on the left shows the details of the compute block for ***Top*** **&** ***Bottom*****. Note that for readability it only displays the logic for one pointer calculation.**

5.1 Module Designs and Optimization

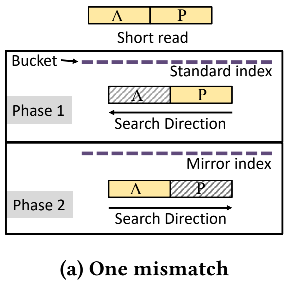
As mentioned, our work is inspired by Bowtie where a read is handled by the mismatch subroutine if exact matching is failed, i.e. a filtered search with the exact match then mismatches. Therefore, the hardware filters are arranged in an alignment pipeline with runtime reconfiguration. Given by Equation 1, modules are replicated on FPGA to increase parallelism. Each module within the filter is also fully pipelined to maximize the matching throughput.

All filters utilize our restructured FM-index to perform the alignment. Figure 5 shows a simplified top-level diagram that implements Algorithm 2 for exact-match filter. The process of alignment within all the filters begins with streaming the reads from the host to FPGA while the restructured FM-index F is preloaded onto the onboard memory. The new pointers for *top* and *bottom* are then calculated based on the symbol and the correlated buckets from the onboard memory. A command block is responsible for sending memory requests according to the new pointer values. Results are streamed back to the host once the reads are matched, or when the reads are unaligned and needed to redirect to the subsequent filter.

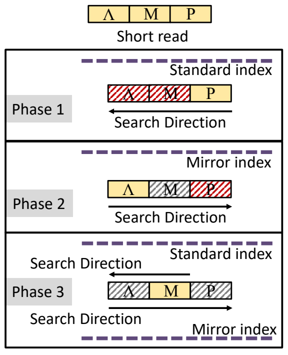
To support approximate matching, additional logic is included in the exact-match filter for one and two mismatch filters to support backtracking. Since FM-index suffers from exponential scaling with the number of permitted mismatches, bi-directional FM-index is used in the one-mismatch and two-mismatch filter to reduce the search space. Details about bi-directional FM-index can be found in [16]. In addition, our three-mismatch filter employs a seed-and-compare strategy. Seeds are exactly matched with the FM-index and the mismatch locations are identified in a subsequent direct comparison stage.

Bi-directional FM-index — In the one-mismatch filter, we use two different indices of the reference. The first one contains the standard restructured FM-index of R, while the second one contains the index of the reversed of R, which is called the mirror index. This enables a character search from both the start or the end of a read. Figure 6a illustrates the mechanism of alignment using bi-directional FM-index. During the search, the mismatch character can either fall onto the left or right half of the read. Therefore, our one-mismatch filter proceeds in two phases to handle these two cases: Phase 1 uses our restructured FM-index F to begin the search from the end of the read, with the constraint that one mismatch occurs only in the left half of the read. Phase 2 uses the mirror index to begin the search from the start of the read, with the constraint that one mismatch occurs only in the right half. By constraining the mismatch position, a long segment of the read can be exactly matched to the reference initially. This can largely reduce the search space size without sacrificing any sensitivity.

The two-mismatch filter also utilizes the standard and mirror index to perform the search. To reduce the search space, the short read is partitioned into three parts (Left Λ, Middle M, Right P) with the same length. A reportable alignment falls into one of the three cases as illustrated in Figure 6b: (case I) 2 mismatches in ΛM; (case II) $\leq$ 1 mismatch in M and $\geq$ 1 mismatch in P; (case III) 1 mismatch in Λ and 1 mismatch in P. Therefore, the two-mismatch filter proceeds in three phases corresponding to these three cases: Phase 1 begins with exact matching from the end of P, and attempts two-mismatch alignment once it reaches M, using the standard index. Phase 2 uses the mirror index to perform a similar search on the short reads, with the constraint of having at most one mismatch character in M. Phase 3 corporates the first 2 phases and find partial alignment with one mismatch in Λ using the standard index. Then it extends the



(a) One mismatch



(b) Two mismatches

Figure 6: One and two-mismatch alignment using bi-directional FM-index. The segments shaded in grey are regions for testing one mismatch, while segments shaded in red are regions for testing two mismatches. The non-shaded areas are exact-match regions.

partial alignment with the mirror index to perform one mismatch alignment in P.

Seed-and-compare — In the three-mismatch filter, it first splits each read into four seeds - equal length non-overlapping substrings. Then, the position of each seed is computed using the exact-match logic and the positions are directed to the compare step on the processor. Consequently, this filter avoids using a backtracking algorithm and is instead able to exploit the efficient exact-match logic. The compare step is performed on the processor for two reasons. First, it involves a direct and consecutive comparison of nucleotides which exhibit spatial locality. Second, this step is a constant time operation, and is not computationally intensive enough to justify running on FPGA when accounting for reconfiguration overhead.

Concurrent Processing for Multiple Reads — Although bucketing improves the utilization of memory throughput, the alignment performance still suffers from the latency of memory access. According to Algorithm 2, the new pointers for *top* and *bottom* are calculated based on the values from F, which in turn depends on *top* and *bottom* from existing iteration. This incurs a vast amount of stall.

We tackle this problem by proposing an interleaving scheme to process multiple reads concurrently. Displayed on the left of Figure 5, a block memory (BRAM) is used to store a few short reads during the mapping process. In each clock cycle, a read is selected from BRAM and the next symbol is processed. This enables full utilization of the FPGA as almost one nucleotide can be processed and a memory request can be sent every clock cycle. Note that we apply this technique to all the filters.

5.2 Consideration for Complete Biological Information

Another major component of this work is the capability of recognizing the complete biological information in the alignment process. One important information is the quality value, also known as Phred quality [8], a metric that is usually neglected in previous

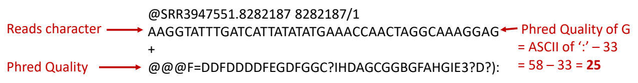

Reads character
Phred Quality
@SRR3947551.8282187 8282187/1
AAGGTATTTGATCATTATATATGAAACCAACTAGGCAAAGGAG
+
@@@F=DDFDDDDFEGDFGGC?IHDAGCGGBGFAHGIE3?D?):
Phred Quality of G
= ASCII of ':' – 33
= 58 – 33 = 25

Figure 7: Explanation on Phred quality using an example.

works. Its values are ranged from 0 to 42 and are presented in ASCII character (with addition of 33). Figure 7 displays an example snippet of a read file in which the second line is the read and the fourth line is Phred quality. Each character in the read is associated with a quality score in the same position and a larger value represents a better quality.

Phred quality is a critical factor for producing biologically correct alignment when **mismatches** are encountered. Essentially, the **quality sum** for an alignment is defined by the sum of Phred quality values at all mismatch positions. Since errors can happen when cell samples are collected, or when sequenced by NGS machine, it is necessary to rank and filter low-quality alignment for biological validity.

Challenges — Previous accelerators usually neglect Phred quality because of design challenges and performance issues. To start, Phred quality requires 6 bits per value, which not only increases the communication overhead between the host and FPGA but also the resource usage. This results in prolonged placement & routing time, more immense fan-out, and lower clock frequency. Furthermore, a read can have various combinations of mismatch locations that contribute to several aligned results per read. For example, given reference R = ACACGT and read pattern P = ACC in Figure 8, the last character of P can be replaced with A or G, forming the results ACA or ACG, for successful alignment with one mismatch. The consideration of Phred quality complicates the design logic and lengthens the alignment time, as we need to sort and rank these results based on the quality sum of each alignment.

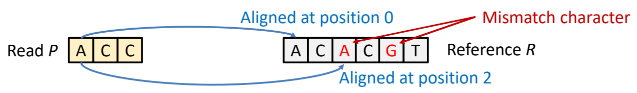


Figure 8: Example of one-mismatch alignment with two aligned results.

Separated Computation & Minimal Ports — In our proposed accelerator, the compute block for the quality sum is separated so that the original alignment throughput and latency are not affected. A 7-bit register is used to buffer the quality sum for one aligned result. Multiple of these registers form a cascade of shift registers which contain the **quality sums for all aligned results of a single read**. To minimize the number of ports at the interface, only the necessary information is directed to this block such as read ID and mismatch locations so that fan-out is reduced.

By default, 32 registers where each of them is 7-bit are cascaded to form the required shift register so as to cache the quality sums for a read. From our observation with the use of Bowtie, successful alignment with mismatches exhibits less than 32 possible outcomes, given that the reads are 50 to 150 base pairs (*bp*).

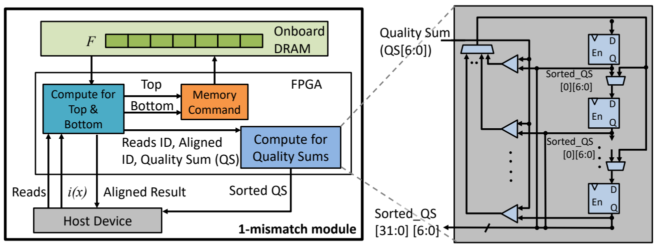


Figure 9: Simplified top-level diagram for one-mismatch module which contains the compute block for sorting quality sums of a read. For readability, some data and control paths are omitted.

On-the-fly Sorting — As shown in Figure 9, a set of comparators is used so that results with the highest quality values are always preserved. When a new alignment is found for particular mismatch location(s), these comparators match the quality sum of this aligned result with the existing ones from the shift registers. Accordingly, an insertion point can be obtained and the quality sums smaller than the current one are shifted in the shift register. Note the underlying logic is fully pipeline so that it can receive the aligned result from the compute block for *top* & *bottom* every cycle, despite the comparison and the shifting operations can take a few cycles.

Three-mismatch Filter & Reads containing Ambiguous Characters — Finally, Phred quality values are also considered in the alignment with the three-mismatch filter. The quality sum for each alignment is calculated during the compare step on the processor, and only the ones with the highest values are reported. Additionally, reads containing the ambiguous characters (N characters) are handled by the processor as soon as the exact-match hardware starts processing. According to Bowtie, only alignments involving ambiguous characters in the read are legal, and ambiguous characters in the read mismatch all other characters. When an ambiguous character is seen in the input, the reads are processed directly on the processor to obtain the possible alignments with no more than three mismatches. Given that only a small portion of reads contains N characters, this step can be completely overlapped by the alignment on FPGA.

6 EXPLORATION FOR AUTOMATIC TOOL

As mentioned, n-step FM-index can align more efficiently if the memory bandwidth improves as the bucket size increases. In this section, we explore the relationship between step n and bucket size versus the alignment speed. Based on this information, we can optimize the FPGA from memory access perspective and provide an insight into the automatic tool.

6.1 Exploration of step n versus Alignment Speed

The memory bandwidth plays a major role in alignment performance, especially when the increase in n contributes to a larger bucket size. Table 2 shows the final bucket and index size for n == 1 to 3.

Table 2: The bucket and index size for different step n.

Step n	1	2	3
Bucket / Index Size	64 B / 1.06 GB	128 B / 3.2 GB	320 B / 12.15 GB

In this experiment, we use Maxeler's MAX5C DFE which is equipped with Xilinx Virtex UltraScale+ VU9P and three DIMMs of 16 GB onboard memory. The FPGA runs at 250 MHz while the host runs with Intel Xeon CPU E5-2643 at 3.4 GHz and 64 GB DDR4-2400 memory. Centos 7.0 is installed on the host. MaxCompiler 2018.2 and Vivado 2017.4 are used for synthesis and implementation. PCI-e 2.0 is used to transfer the data between the host and FPGA. As the computation is bottleneck by the onboard memory on the FPGA, PCIe 2.0 is already sufficient for the data transfer.

GRCh38 [11] is used as the reference and single-end reads from human_100_300M in [17] are used as the input. This dataset contains 300×10^6 reads with $101bp$, and was originally used to evaluate Bowtie on a multi-core system to investigate the alignment performance. It is composed of reads generated by the Illumina HiSeq2000 instrument from various genome projects.

Figure 10 shows the alignment speed against different n for the exact-match, one-mismatch, and two-mismatch filters. For each filter, the corresponding computational module is populated on the FPGA by one and three times to investigate the performance difference. Also displayed in the graph is the measured throughput of accessing onboard memory from FPGA for different n, resulting in different number of bursts transfer. The graph on the right elaborates on the corresponding resource consumption in percentage required by each filter. The resource usage for DSP is not shown in the figures as less than 1 % is used by all the filters.

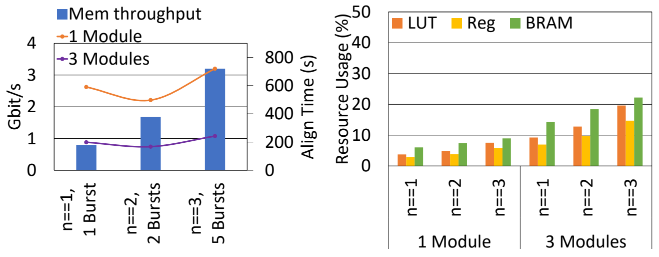
On the target platform, the alignment speed is optimal when n is 2 where the bucket size is 128 B (two bursts). Since the memory bandwidth increases by 2.2 × when two bursts are transferred, the alignment speed can slightly improve when 2 characters are matched using 2-step FM-index. However, when n equals to 3, five memory bursts are needed to transfer a bucket, while the memory bandwidth only increases by 4 × compared to one-burst transfer. Thus, 3-step FM-index doesn't provide any performance advantages on VU9P. Moreover, when the number of modules is tripled, the alignment performance is improved by 3 ×. This showcases the scalability where the performance scales linearly with the module counts. Finally, the critical resource for each module is BRAM ($\sim$ 20 – 50% usage) because of the circular buffer for concurrent processing. There remains an adequate amount of space for module replications should more memory DIMMs are given.

6.2 Automatic Tool

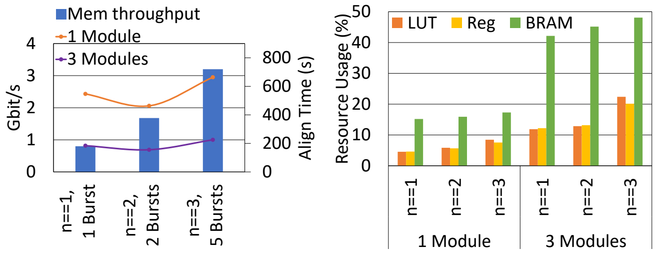
From the exploration, we notice that the alignment performance can be affected by factors such as memory throughput, step, etc. Therefore, the automatic tool, written in Python, consists of three major stages: Memory Performance Measurement, Hardware Customization, and Hardware Compilation.

Memory Performance Measurement — Since FM-index based alignment is bottleneck by the memory, the first stage accepts the information about the FPGA, and generates hardware that measures the throughput and latency of the onboard memory. This stage relies on the Memory and PCI-e IP core from Vivado. A pre-written hardware template that utilizes these IP cores is used to access the onboard memory. When the hardware is run on the target FPGA, the memory access latency L and throughput T_n with respect to different number of bursts are obtained accordingly. We set the frequency as 250 MHz when speed grade ≤ -2 and 150 MHz when speed grade $= -1$. Optionally, the user can provide the memory performance information and skip this stage.

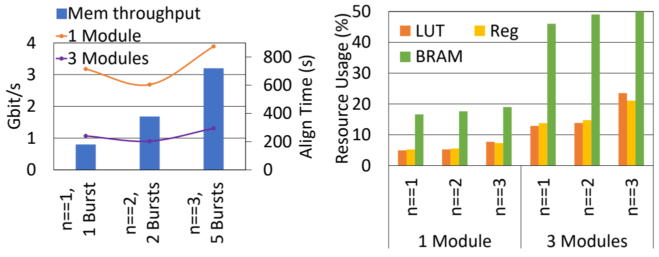
Hardware Customization — Based on the exploration, we understand that an increasing n does not necessarily improve the alignment speed, unless the memory throughput is improved by more than its own burst counts. We also observe that the BRAM usage often dominates the resource consumption. With these properties, the hardware aligner can be optimized based on the following analysis and different equations. First, given a step n, we calculate the number of bursts Cnt_{burst_n} needed for a bucket. Then we can calculate which n provides the optimal alignment performance with:



(a) Exact-match filter



(b) One-mismatch filter



(c) Two-mismatch filter

Figure 10: The alignment speed against n for the exact-match, one-mismatch, and two-mismatch filters (Left). The corresponding resource consumption with respect to the available resources is represented in percentage (Right).

$$T_1 < \frac{T_n}{Cnt_{burst\_n}}, \text{for } n = \{2, 3\} \tag{4}$$

If there exist one or more n satisfying Equation (4), we select the n which contributes to $max(\frac{T_n}{Cnt_{burst\_n}})$. We also need to consider two more equations that restrict the design space smaller than the available BRAM:

Usage of BRAM for Exact-match Filter

$$= \frac{Size_{var} \times N_{iread} \times M_j + Size_{bucket} \times M_j \times 2}{BRAM_{size}} < Cnt_{BRAM}, \tag{5}$$

Usage of Block Memory for Filter with Mismatches

$$= \frac{(Size_{var} + Size_{backtrack}) \times N_{iread} \times M_j + Size_{bucket} \times M_j \times 2}{BRAM_{size}} < Cnt_{BRAM}, \tag{6}$$

where $Size_{var}$ and $Size_{backtrack}$ are defined as the bit-width required by the variables and backtracking variables respectively within the module. N_{iread} is the number of interleaved reads based on L. Additionally, since a FIFO buffer is used to temporarily cache the bucket from the onboard memory, $Size_{bucket}$ is included in the equations which stands for the bucket size. Cnt_{BRAM} is the memory resources available on the chip. Moreover, as the number of modules M_j is limited by the number of DIMMs based on Equation (1), we also need to satisfy the following:

$$M_j \leq N_{DIMM}, \text{where } j \text{ is the filter} \tag{7}$$

Lastly, we use `scipy` in Python to solve Equations (5)/(6) and (7) with the objective of maximizing M_j for different filters j. If no solution is found, we select a smaller n and solve the above linear equations again until we exhaust all possible n.

6.3 Hardware Compilation

When the hardware customization is finished, we direct the hardware to Vivado for synthesis and implementation. For $M_j \geq 3$, other issues such as congestion or large fan-out might arise during compilation, particularly on a small FPGA. This is because the aligner now requires more than 50 % of BRAM on-chip for moderate size FPGA such as Xilinx VU9P. Therefore, the automatic tool provides an option to concurrently compile the hardware with a smaller amount of M_j.

7 EXPERIMENTAL RESULTS AND DISCUSSION

To evaluate the performance and limitations of our proposed design, we initialize the hardware with the maximum number of modules, $M_j = 3$, using 2-step FM-index (bucket size = 128 B) based on the results in Section 6.1. All other settings for our accelerator are identical to the previous section.

7.1 Alignment Accuracy

As our proposed hardware design is modeled upon the algorithms implemented in Bowtie, this subsection evaluates the alignment accuracy, with results from Bowtie acting as the ground truth. Furthermore, we compare the performance difference between our aligner and software. We also select the accelerator proposed by Arram *et al.* [2] for comparison as it is based on a similar methodology to perform runtime alignment. We manage to download [12] and re-implement their design on our experimental device with a frequency of 250 MHz.

Given the relatively small workload (~7 % of a full alignment workload), the reconfiguration time (9-12 s per configuration) for VU9P does not have a large impact on the overall alignment time. Moreover, different datasets and alignment parameters are used in different previous works, and some accelerators use more than one FPGA to perform the alignment. To allow a fair comparison, we define a normalized metric, base pairs aligned per second per device (bps^{-1}), which is given by:

$$bps^{-1} = \frac{read\ length \times read\ count}{alignment\ time \times no.\ of\ device} \tag{8}$$

Table 3: A comparison between set-up and alignment results of Arram *et al.*, the proposed design and Bowtie.

	Arram *et al.*	**Proposed Aligner**	**Bowtie**
Device	Xilinx Virtex UltraScale+ VU9P		Intel Silver 4110 (16 threads)
Frequency	250 MHz		2.1 GHz
Lithography	TSMC 16 nm		Intel 14 nm
Align time	638 s	683 s	3330 s
bps^{-1} (million)	47.5	44.4	9.10
Accuracy	96.3 %	97.7 %	–

Table 3 displays the comparison between the set-up and alignment results of the hardware and software. Accuracy is defined as the fraction of correct alignment and un-alignment from the proposed aligner among all of the reads. Correct alignment refers to reads matched to the same locations as Bowtie. Correct un-alignment means that if the reads are not aligned by Bowtie, they should not be aligned by the proposed aligner also.

The 2.3 % discrepancy in accuracy, between our accelerator and Bowtie (100 % accuracy for the ground truth), is due to two reasons. First, some of the reads are successfully aligned to the reference using FPGA but fail to map using Bowtie. Bowtie imposes a backtracking restriction to limit effort spent finding valid alignments. This causes Bowtie to miss some legal mismatch alignments. This clearly demonstrates the biological validity of the proposed aligner, since our design can exhaust all the possible mismatch locations without putting a backtracking limit. This is important for alignment accuracy because statically around 20 % of reads from the NGS machine can be aligned with one and two-mismatch filters. Second, Bowtie employs a heuristic to determine the mapping locations, especially when the reads have multiple aligned results. Some of the reads that can be successfully aligned by Bowtie are mapped to a different location in our proposed aligner. The accelerator proposed by Arram *et al.*, on the other hand, is unable to reflect a comparable accuracy. Despite the performance, the accuracy drops to 96.3 % because of the overlooking of the quality metric, as well as the missing stage of the three-mismatch filter. Compared to Arram *et al.*, our proposed aligner can report **4.48 M** more correct alignments. The better performance of Arram *et al.* is also due to

Table 4: Performance comparison with previous hardware accelerators that are based on similar algorithms.

Year	Work	Algorithm & Method	Platform	Device	Read Count (Million)	Read Length (Base Pair)	Align Time	Mbps^{-1}
2012	[9]	FM-index	Convey HC-1	Virtex-5 LX330×4	18	101	73.3 s	6.20
2013	[1]	FM-index + Smith-Waterman	Maxeler MAX3	Virtex-6 SX475T	82	90	49.0 s	151
2013	[2]	FM-index	Maxeler MAX3	Virtex-6 SX475T	18	75	13.8 s	97.8
2015	[3]	FM-index	Maxeler MPC-X1000	Stratix-V×8	10	75	11.3 s	8.30
2019	[15]	Bowtie2 + Smith-Waterman	Maxeler MAX5C	Virtex Ultrascale+ VU9P	60	100	5830 s	1.03
2020	This work	FM-index	Maxeler MAX5C	Virtex Ultrascale+ VU9P	300	100	683 s	44.4

the missing three-mismatch filter, where short reads unaligned by the two-mismatch filter are left unprocessed and redirected to the host straight away. If the three-mismatch filter is omitted in our proposed aligner, our implementation needs around 562 s to finish processing all the reads which is similar to Arram's work.

Finally, the proposed aligner can achieve a reasonable speedup of 4.88 × where the alignment time is decreased from **56 minutes** in software to around **12 minutes**. With the assurance of biological validity and reproducibility of the alignment results, the proposed aligner provides an opportunity to bridge the gap between alignment research and practice, allowing applications such as cancer diagnosis to become part of routine clinical procedures.

7.2 Performance Comparison with Existing Accelerators

Table 4 demonstrates the performance comparison of the proposed aligner with different accelerators. Since different designs conduct their evaluations using different datasets, we do not compare the alignment accuracy in this evaluation. Note that we only select the hardware aligners that are based on Bowtie, Bowtie2, or FM-index, as they are relatively similar to ours. Accelerators built upon other algorithms such as Minimap2 are not considered as they use different techniques or target long reads alignment.

The million bps^{-1} values in Table 4 indicate that our aligner outperforms most of the existing designs apart from [1] and [2]. This showcases the benefits of our design where the addition of computation logic for quality sums does not affect alignment time. On the other hand, our aligner is slower than [1] and [2] mainly because of the availability of DIMMs on the respective devices. Their platforms are based on Maxeler MAX3 where there exist seven memory DIMMs onboard. With the number of memory channels more than double compared to Xilinx VU9P, the alignment speed is therefore more significant in their work. It is also important to note that their evaluation methods are based on the theoretical upper bound estimation on each respective device. Hence, the actual performance of their implemented designs might be less significant, as the routing congestion and fan-outs are not taken into consideration. Finally, [1] only consists of two stages in the alignment pipeline, where the approximate matching is performed with the Smith-Waterman algorithm. Therefore, the alignment speed can be faster than the proposed aligner as we exhaust all possible mismatch locations using backtracking FM-index.

Table 5: Area cost of the final design on VU9P. Percentage values are relative to the available resources on target FPGA.

	LUT	Register	BRAM	DSP
Exact Match	151272(13%)	229014(10%)	973(18%)	9(1%)
1-Mismatch	152111(13%)	311544(13%)	2385(45%)	9(1%)
2-Mismatch	163464(14%)	348755(15%)	2590(49%)	9(1%)

7.3 Resource Consumption

Based on the proposed reconfigurable architecture using 2-step FM-index and $M_j = 3$, Table 5 indicates the corresponding resource consumption for the implementation of each filter on VU9P, with the inclusion of the PCI-e and memory controller. With an adequate area remaining on the FPGA, more modules can be populated on the FPGA if the number of DIMMs increases. Note these numbers can only serve as a reference, as many factors can affect the final resource computation when migrated onto other platforms.

8 CONCLUSION

In this work, we propose a novel, general reconfigurable architecture to address the memory bottleneck of suffix-trie based alignment. Our architecture is based on FM-index where we use different optimization techniques, such as bucketing, n-step FM-index, bi-directional FM-index, and concurrent processing of reads to improve alignment performance. Complete biological information is also considered in the alignment pipeline to maximize accuracy. With a guarantee of biological validity and reproducibility, we anticipate the proposed hardware aligner can promote the adoption of FPGA in short read mapping among biological and clinical scientists. Further research includes decreasing the area cost by leveraging an automatic design analyzer and merger [25], investigating alternative memory access scheme [23], supporting paired-end reads and handling indels.

ACKNOWLEDGMENT

The support of the Lee Family Scholarship, the Great Britain-China Educational Trust, the Hong Kong Scholarship For Excellence Scheme, the UK EPSRC (EP/L016796/1, EP/N031768/1, EP/P010040/1 and EP/L00058X/1), the National Natural Science Foundation of China (62001165), Maxeler, Intel and Xilinx is gratefully acknowledged.

REFERENCES

[1] J. Arram et al. 2013. Reconfigurable Acceleration of Short Read Mapping. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. 210–217.

[2] J. Arram et al. 2013. Reconfigurable filtered acceleration of short read alignment. In *2013 International Conference on Field-Programmable Technology (FPT)*. 438–441.

[3] J. Arram et al. 2015. Ramethy: Reconfigurable Acceleration of Bisulfite Sequence Alignment. In *2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 250–259.

[4] M. Burrows and D. Wheeler. 1994. *A Block-sorting Lossless Data Compression Algorithm*. Technical Report. Digital Equipment Corporation.

[5] A. Chacón et al. 2013. n-step FM-Index for Faster Pattern Matching. *Procedia Computer Science* 18 (2013), 70 – 79.

[6] M. C. F. Chang et al. 2016. The SMEM Seeding Acceleration for DNA Sequence Alignment. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 32–39.

[7] P. Draghicescu et al. 2012. *Inexact Search Acceleration on FPGAs Using the Burrows-Wheeler Transform*. Technical Report.

[8] B. Ewing et al. 1998. Base-Calling of Automated Sequencer Traces Using Phred. I. Accuracy Assessment. *Genome Research* 8, 3 (1998), 175–185.

[9] E. Fernandez et al. 2012. Multithreaded FPGA acceleration of DNA Sequence Mapping. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–6.

[10] P. Ferragina and G. Manzini. 2001. An Experimental Study of an Opportunistic Index. In *12th Annual ACM-SIAM Symposium on Discrete Algorithms*. 269–278.

[11] National Center for Biotechnology Information. 2020. Genome Reference Consortium Human Build 38. https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.26/

[12] P. Grigoras et al. 2017. dfesnippets:An Open-Source Library for Dataflow Acceleration on FPGAs. In *13th International Symposium on Applied Reconfigurable Computing*. 299–310.

[13] E. J. Houtgast et al. 2015. An FPGA-based Systolic Array to Accelerate the BWA-MEM Genomic Mapping Algorithm. In *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 221–227.

[14] E. J. Houtgast et al. 2016. Power-Efficiency Analysis of Accelerated BWA-MEM Implementations on Heterogeneous Computing Platforms. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–8.

[15] K. Koliogeorgi et al. 2019. Dataflow Acceleration of Smith-Waterman with Traceback for High Throughput Next Generation Sequencing. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 74–80.

[16] B. Langmead et al. 2009. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology* 10, R25 (2009).

[17] B. Langmead et al. 2019. Scaling read aligners to hundreds of threads on general-purpose processors. *Bioinformatics* 35, 3 (2019), 421–432.

[18] B. Langmead and A. Nellore. 2018. Cloud computing for genomic data analysis and collaboration. *Nature Reviews Genetics* 19 (2018), 208–219.

[19] H. Li. 2013. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997v2* (2013).

[20] H. Li and R. Durbin. 2009. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics* 25, 14 (2009), 1754–1760.

[21] G. Lightbody et al. 2019. Review of applications of high-throughput sequencing in personalized medicine: barriers and facilitators of future progress in research and clinical application. *Briefings in Bioinformatics* 20, 5 (2019), 1795–1811.

[22] N. A. Miller et al. 2015. A 26-hour system of highly sensitive whole genome sequencing for emergency management of genetic diseases. *Genome Medicine* 7, 100 (2015), 16 pages.

[23] H. Ng et al. 2013. Direct Virtual Memory Access from FPGA for High-productivity Heterogeneous Computing. In *2013 International Conference on Field-Programmable Technology (FPT)*. 458–461.

[24] H.-C. Ng et al. 2017. Reconfigurable Acceleration of Genetic Sequence Alignment: A Survey of Two Decades of Efforts. In *27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8.

[25] H.-C. Ng et al. 2018. ADAM: Automated Design Analysis and Merging for Speeding up FPGA Development. In *2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 189–198.

[26] H.-C. Ng et al. 2020. Acceleration of Short Read Alignment with Runtime Reconfiguration. In *2020 International Conference on Field-Programmable Technology (FPT)*. 7 pages.

[27] S. Salamat and T. Rosing. 2020. FPGA Acceleration of Sequence Alignment: A Survey. *arXiv preprint arXiv:2002.02394v2* (2020).

[28] B. Schmidt and A. Hildebrandt. 2017. Next-generation sequencing: big data meets high performance computing. *Drug Discovery Today* 22, 4 (2017), 712–717.