Ramethy: Reconfigurable Acceleration of Bisulfite Sequence Alignment

James Arram
Department of Computing
Imperial College
jma11@imperial.ac.uk

Wayne Luk
Department of Computing
Imperial College
wl@imperial.ac.uk

Peiyong Jiang
Department of Chemical Pathology
The Chinese University of Hong Kong
jiangpeiyong@cuhk.edu.hk

ABSTRACT

This paper proposes a novel reconfigurable architecture for accelerating DNA sequence alignment. This architecture is applied to bisulfite sequence alignment, a stage in recently developed bioinformatics pipelines for cancer and non-invasive prenatal diagnosis. Alignment is currently the bottleneck in such pipelines, accounting for over 50% of the total analysis time. Our design, Ramethy (Reconfigurable Acceleration of METHYlation data analysis), performs alignment of short reads with up to two mismatches. Ramethy is based on the FM-index, which we optimise to reduce the number of search steps and improve approximate matching performance. We implement Ramethy on a 1U Maxeler MPC-X1000 dataflow node consisting of 8 Altera Stratix-V FPGAs. Measured results show a 14.9 times speedup compared to soap2 running with 16 threads on dual Intel Xeon E5-2650 CPUs, and 3.8 times speedup compared to soap3-dp running on an NVIDIA GTX 580 GPU. Upper-bound performance estimates for the MPC-X1000 indicate a maximum speedup of 88.4 times and 22.6 times compared to soap2 and soap3-dp respectively. In addition to runtime, Ramethy consumes over an order of magnitude lower energy while having accuracy identical to soap2 and soap3-dp, making it a strong candidate for integration into bioinformatics pipelines.

Categories and Subject Descriptors

J.3 [**LIFE AND MEDICAL SCIENCES**]: Biology and genetics

General Terms

Design, Performance, Experimentation

Keywords

bioinformatics, alignment, reconfigurable hardware, next-generation-sequencing

FPGA'15, February 22–24, 2015, Monterey, California, USA.

ACM 978-1-4503-3315-3/15/02 ...$15.00.
http://dx.doi.org/10.1145/2684746.2689066.

1. INTRODUCTION

DNA methylation is a biochemical process which involves the addition of a methyl group to cytosine or adenine nucleotides. It commonly occurs on cytosines belonging to a CG pair (cytosine followed by guanine), which serves as a mechanism for gene regulation by turning certain genes off. Studies have shown that methylation is vital to healthy growth and development, and aberrant methylation has been associated with the development of cancer. With the advent of next-generation sequencing (NGS), it is now possible to conduct whole-genome methylation analysis at single base resolution. This has recently been used in developing methods for cancer [6] and non-invasive prenatal diagnosis [16]. However, a major challenge impeding such methods is the data analysis, which can take an inordinate amount of time, and is performed by a multitude of dissociated programs.

Methy-Pipe [10] is a recently developed integrated pipeline for whole-genome methylation analysis. It not only fulfils the core data analysis requirements such as sequence alignment and differential methylation analysis, but also provides useful tools for methylation data annotation and visualisation. When compared to previous related efforts, Methy-Pipe has greater functionality and user-friendliness, making it an invaluable tool for researchers and clinical scientists. However, its usefulness is limited by the time it takes to transform raw sequenced data into appropriate information to enable diagnosis.

The bottleneck of Methy-Pipe is sequence alignment, in which short sequences of DNA letters (called reads) are mapped to locations in a known reference genome. Alignment of 300M short reads to the Human genome takes roughly 5 hours when running on a system with dual 12-core Intel Xeon processors and 100GB of RAM. Consequently, Methy-Pipe is still yet to meet the requirements for large-scale clin-

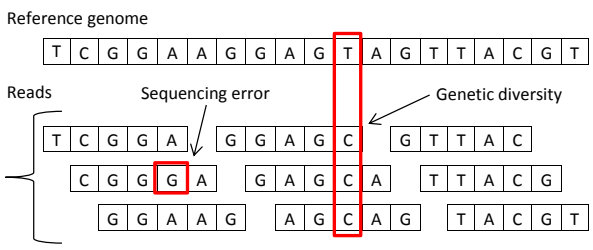

Figure 1: Alignment of reads to a reference genome.

ical adoption, in which patient turnaround time is critical. Accelerating alignment would shorten the diagnosis time, thereby allowing faster responses and increasing the number of patient samples that can be analysed per day. This achievement would facilitate bridging the gap between research and practice, enabling the diagnosis techniques developed to become part of routine clinical procedures.

There currently exist multiple freely available software tools for alignment, including Soap [14], BWA [13], and Bowtie [12]. These tools utilise the latest pattern matching algorithms and hardware technologies to perform the alignment process quickly. However, there is reliance on extensive computing resources to deliver this performance. For example, the 1000 genome project uses a 1192-processor cluster to align reads, while the BGI Bio-cloud computing platform has a current total of 14774 processors delivering 157T flops of performance. Such approaches are short-sighted, as simply scaling across more machines cannot keep up with the projected growth of sequenced data which far exceeds Moore's Law.

Reconfigurable hardware, such as the field programmable gate array (FPGA), is a promising candidate for accelerating alignment. The multiple levels of exploitable parallelism can provide substantial speed-up, whilst the low operational clock frequencies allow reduced energy consumption and high rack unit densities. Several papers have been published which report the use of FPGAs to accelerate the alignment process [7, 19, 8]. Whilst these designs outperform most of the software tools, the speed comes at the cost of accuracy, functionality, and platform independence. As a result, few hardware-based alignment tools have been fully integrated into a bioinformatics pipeline.

In this paper we present a novel reconfigurable architecture for accelerating alignment, which is applied to bisulfite sequencing alignment – the specific type of alignment used in methylation data analysis. Our aim is to show how reconfigurable hardware can be used to deliver substantial improvements in runtime, energy consumption and form factor, making it an ideal candidate for integration into bioinformatics pipelines. The contributions of this work include:

- A novel architecture for accelerating alignment consisting of a multi-configuration alignment pipeline. This architecture exploits the reconfigurability of FPGAs to allow for highly efficient, yet flexible alignment designs.
- An application of this architecture to accelerate bisulfite sequence alignment, which is the bottleneck in whole-genome methylation analysis. Our design, referred to as Ramethy, is based on accelerating the FM-index, a pattern matching algorithm.
- A novel optimisation to the FM-index which improves the pattern matching performance. We propose a new index structure which reduces the number of search steps and computational complexity compared to previous efforts.
- An implementation of Ramethy on a Maxeler MPC-X1000 dataflow node. The runtime, energy consumption and alignment accuracy is evaluated and compared to the fastest CPU, GPU and FPGA-based alignment programs currently available.

The rest of this paper is organised as follows: Section 2 provides background information on the ideas and algorithms used throughout the paper. Section 3 gives an overview of related efforts in this field. Section 4 presents the reconfigurable architecture for accelerating alignment. Section 5 covers the algorithm optimisation developed to improve the pattern matching performance of the FM-index. Section 6 gives an overview of the design of Ramethy. Section 7 evaluates the performance of Ramethy and provides comparison with the fastest CPU, GPU and FPGA-based alignment programs currently available. Finally, Section 8 concludes the paper.

2. BACKGROUND

In this section, background information on the algorithms used to perform alignment is presented. Particular attention is paid to the FM-index, the chosen algorithm for acceleration in this work. This information will help clarify the proposed architecture and algorithm optimisations presented later in this work.

2.1 Alignment Algorithms Overview

The algorithms used by currently available alignment software tools can be categorised into two groups: 1) suffix-trie algorithms, such as the FM-index [9], in which reads are aligned using an index generated from the suffixes of the reference genome, and 2) seed-and-extend algorithms, such as the Smith-Waterman algorithm [20], in which subsequences (seeds) of the read are aligned to the reference genome, and any candidate locations are extended using a scoring matrix. The alignment parameters of an experiment typically dictate which group is chosen. For reads with less than 150 bases and small edit distances, suffix-trie algorithms provide the best performance. Conversely, for reads with hundreds to thousands of bases and large edit distances, seed-and-extend algorithms provide the best performance. The objective of this work is to speedup Methy-Pipe, a bisulfite sequencing data analysis pipeline, in which reads of 75 bases are aligned to the Human genome with up to two mismatches. As a result, suffix-trie algorithms, specifically the FM-index, are chosen for hardware acceleration.

2.2 FM-index

Indexing a reference sequence is a well established method for accelerating pattern matching. For the Human genome, the time spent on creating an index can take several hours. However, this time is amortised given that the Human genome version changes infrequently, so the index only needs to be created once for a large number of alignment jobs to be performed. The FM-index is the most common indexing method used by currently available alignment software tools due to its small memory footprint and efficient substring searching. The design of the FM-index is based on the Burrows-Wheeler compression algorithm (BWT) [4] and the suffix-array (SA) data structure [18].

The SA of a text R is the permutation of the lexicographically sorted suffixes of R, where each suffix is represented by its starting position. To simplify the substring searching operation, the symbol '$', which is lexicographically smaller than all other symbols in the alphabet Σ, is appended to the text. The SA for a text R = ACGTTAAA\$ is shown in Table 1(a).

An interval in an SA represents a sequence of lexicographically consecutive suffixes of R. The interval (low, $high$) corresponds to the smallest and largest indexes in SA which have the same prefix. The result of searching for a pattern Q

in R can be represented as an SA interval. If $low \leq high$, the query can be located in the text. Conversely if $low > high$, the query cannot be located in the text. For a query Q = AA, the SA interval equals (2, 3). This solution interval can be converted from SA to text coordinates using the relationship R_1 = SA[low], R_2 = SA[$low + 1$], ... , R_n = SA[$high$]. For the interval (2, 3), the corresponding positions in R which have Q as prefix are 6 and 5.

The FM-index is built upon the BWT, a data compression technique which generates a permutation of the symbols in a text R. The BWT of a text (denoted by B) is a *self index*, which refers to the property that it does not require R and SA to perform a search operation. Each position in B is computed from R and SA using the relationship $B[i] = R[(\text{SA}[i]-1) \bmod |R|]$. For a text R = ACGTTAAA\$, the corresponding B = AAAT\$ACTG. The FM-index supports searching operations through two counting functions derived from B. $C(c)$ is a function that returns the number of symbols in B that are lexicographically smaller than c, and $Occ(c, i)$ is a function that returns the number of occurrences in B of character c, from positions 0 to i. These functions are typically precomputed and stored as arrays to improve performance. As a trade-off between memory space and time, only the $Occ()$ values for positions that are a multiple of some integer distance d are stored. This technique (referred to as bucketing) allows compression of Occ(), but requires counting some of the occurrence directly from B. Table 1(b) displays the functions $C()$ and $Occ()$ for a text R = ACGTTAAA\$.

(a)

R = ACGTTAAA\$

i	sorted suffixes	SA[i]
0	\$	8
1	A\$	7
2	AA\$	6
3	AAA\$	5
4	ACGTTAAA\$	0
5	CGTTAAA\$	1
6	GTTAAA\$	2
7	TAAA\$	4
8	TTAAA\$	3

(b)

$Occ()$

i	$B[i]$	A	C	G	T
0	A	1	0	0	0
1	A	2	0	0	0
2	A	3	0	0	0
3	T	3	0	0	1
4	\$	3	0	0	1
5	A	4	0	0	1
6	C	4	1	0	1
7	T	4	1	0	2
8	G	4	1	1	2

$C()$

A	C	G	T
1	5	6	7

Table 1: (a) SA for a text R =ACGTTAAA\$, and (b) the corresponding function tables $Occ()$ and $C()$.

Algorithm 1 shows the procedure for searching for a query Q in a text R. Concisely written, the SA interval is first initialised to (1, $|R|$). Then moving from the last symbol of Q to the first, the SA interval is iteratively updated using $C()$ and $Occ()$ (a backward search). After the final iteration, the SA interval gives all the consecutive indexes in the SA which have Q as prefix, which are subsequently converted into text coordinates. Currently available alignment software tools typically augment this algorithm with backtracking to support approximate matching between the query and text. In this approach edit operations (substitutions, insertions or deletions) are performed to the query. A stack is used to store the state at each edit position. If the modified query cannot be located in the text, the state is restored from the edit position and a new edit operation is performed. Heuristics for improving the approximate matching performance have been developed, such as the 2-way BWT algorithm [11].

Algorithm 1: Algorithm for substring searching using the FM-index.

Input : Query Q, $C()$, $Occ()$ and SA, corresponding to the text R
Output: Locations in R where Q is a prefix

```
begin
    (low, high) ← (1, |R|)
    for i ← |Q| − 1 to 0 do
        low ← C(Q[i]) + Occ(Q[i], low − 1)
        high ← C(Q[i]) + Occ(Q[i], high) − 1
    end
    for i ← 0 to high − low do
        Locations[i] ← SA[low + i]
    end
    return Locations
end
```

3. RELATED WORK

Currently, there are multiple software tools available for alignment. Some of the freely available programs include Soap2, BWA, and Bowtie. In the case of whole-genome alignment, extensive computational resources are required to run these tools with a reasonable runtime. For example a mid-size cluster with high-end multicore processors and a large amount of RAM in each node would be adequate for small-scale processing.

As a response to the rapidly increasing sequencing machine throughput, GPU-based tools have been developed to improve the alignment performance. Notable GPU-based tools include soap3-dp [17] and CUSHAW [15], which perform up to 10 times faster than CPU-based tools.

There are various efforts related to accelerating alignment with FPGAs, among which accelerating the Smith-Waterman algorithm is the most popular approach. These designs typically target a single hardware platform with a specific number of FPGA devices and memory architecture. Olson et al. [19] propose an FPGA-based alignment design based on the Smith-Waterman algorithm. In their work, both the seed location and score table computation are performed in hardware. The design is partitioned into 8 Pico M-503 boards, each with one Xilinx Virtex-6 FPGA. This 8-FPGA system can align 50 million reads in 34 seconds. Fernandez et al. [7, 8], propose FPGA-based alignment design based on the FM-index. In the first work, the index of a small reference genome is stored in on-chip BRAM. The design is implemented on a single Xilinx Virtex-6 FPGA and can exactly align 1000 reads in 60.2us. In the second work, their previous design is extended to allow for approximate alignment. For every n mismatches allowed, $n + 1$ exact string matchers statically populate an FPGA device in a pipeline. If a mismatch is detected, multiple copies of the read are generated in which the mismatched symbol is replaced with other symbols from the reference genome alphabet. The copies are sent to the next exact string matcher in the pipeline for further processing. The design is implemented on the Convey HC-1 platform and can align 18M reads in 138 seconds.

Arram et al. [1, 2, 3] propose FPGA-based alignment design based on the FM-index. In the first work, the FM-index is extended with depth-first backtracking to support approximate alignment. The design is implemented on a single Xilinx Virtex-6 FPGA and can align reads up to 8 times faster than soap2. In the second work, a two-stage architecture for accelerating alignment is proposed. The reads are first aligned using exact string matchers based on the FM-index and reads which are unable to be aligned are subsequently processed by approximate string matchers based on the Smith-Waterman algorithm. Performance estimates of the interesting design regions indicate that a dynamically reconfigurable design achieves the highest performance. In the third work, an overview of a reconfigurable architecture for accelerating suffix-trie alignment algorithms is presented. An application of this architecture based on the FM-index is implemented on a single Xilinx Virtex-6 FPGA and can align reads 3 times faster than soap2. In contrast to that work, Section 4 of this paper formalises a reconfigurable architecture which is not confined to suffix-trie alignment algorithms; Section 6 illustrates how this architecture is applied to bisulfite sequence alignment.

4. RECONFIGURABLE ARCHITECTURE

In this section we present a reconfigurable architecture for accelerating alignment. Our approach generalises the work in [3] to all alignment algorithms, and provides analytical methods for estimating design performance.

4.1 Rationale

In the various efforts related to accelerating alignment using FPGAs, the target device is statically configured with a circuit functionally equivalent to an alignment algorithm. These circuits consist of several interlinked modules corresponding to the different stages of the alignment algorithm. For example, in [19] the Smith-Waterman circuit consists of modules for seed extraction, seed location and score computation. Similarly in [8], the FM-index circuit consists of modules for exact match, one mismatch and two mismatches alignment. With a static configuration these modules are able to process data concurrently, however there exist a number of limitations for this approach which can reduce both the performance and usefulness of a design.

Data Hazards. Alignment algorithms feature numerous data hazards in which execution of the next stage depends on the result from the previous stage. For example, in an FM-index circuit, a read will only be processed by the one mismatch module if it cannot be aligned by the exact match module. For a statically configured design all modules in the circuit are mapped to the target device. Data hazards result in some modules being left idle from time to time which reduces the hardware efficiency.

Distinct Module Latencies. Each module in a circuit takes a particular number of cycles to process an item of data. To create a balanced pipeline, certain modules are replicated more than others in an attempt to match their latencies. For a statically configured design this approach can be challenging given the limited resources available on the target device fabric. For example, in a Smith-Waterman circuit the score computation module must be replicated a large number of times to match the throughput of the seed location module. It is often impossible to replicate this module a sufficient number of times given the large amount of resources consumed by the other modules.

Extensive Resource Usage. For large static circuits comprising many modules, the resources required to map the circuit may exceed that available on the target device fabric. In this case, a subset of the modules are implemented in software to reduce the resources required to map the circuit. This solution comes at the cost of potential speedup, which is reduced according to Amdahl's Law. For example, in a Smith-Waterman circuit the seed location module often requires more off-chip memory than most hardware platforms have available. As a result, this module is often implemented in software which reduces the overall speedup.

Inflexible Alignment Parameters. Alignment parameters, such as the maximum number of mismatches, gap size and hit reporting method, will change depending on the sequenced data quality and experiment being performed. For a statically configured design there is limited control over these parameters as the circuit is fixed. Any substantial changes will often require several modules to be redesigned, and the circuit to be re-placed and re-routed. This process can take days to complete, which reduces the usefulness of a design.

4.2 Architecture Description

This section proposes a general architecture for accelerating alignment which exploits the reconfigurability of FPGAs. In this architecture distinct FPGA configurations are created for each stage of an alignment algorithm. The configurations comprise a homogeneous array of modules which are functionally equivalent to the corresponding algorithm stage. Runtime reconfiguration is used to load each configuration onto the target device consecutively, where the data are processed concurrently by the modules. Figure 2 illustrates how the proposed architecture is applied to an alignment algorithm with 3 stages. In step 1 modules are designed which are functionally equivalent to a stage in the alignment algorithm. In step 2 the modules are replicated to form an FPGA configuration. The number of times a given module can be replicated is given by Equation 1, in which P_i is the module population, A is the total available resources on the target device, and r_i is the amount of resources required for the module.

$$P_i = \frac{A}{r_i} \tag{1}$$

In Step 3, the computational workflow shown in Algorithm 2 is performed. For each stage in the alignment algorithm the corresponding configuration is loaded onto the target device. Data from the previous stage (or initial data) are streamed to the target device, where they are processed concurrently by the modules. The output data from the modules are stored in off-chip memory attached to the target device, or in host memory.

The performance of this architecture can be modelled using Equation 2, in which T is the alignment time, N_i is the number of data items processed by a given alignment stage, t_i is the time for the corresponding module to process a single item of data, and P_i is the population of modules in the configuration. The overhead of this architecture is the reconfiguration time t_r, and the data communication overhead t_o. For typical alignment workloads these overheads

Algorithm 2: Multi-configuration alignment pipeline algorithm.

Data: Reads
Result: Alignment locations

for $i \leftarrow 1$ **to** *number of stages in algorithm* **do**
  load configuration i onto target device
  if $i = 1$ **then**
    stream initial data from host mem.
  else if *intermediate data in off-chip mem.* **then**
    stream data from off-chip mem.
  else
    stream data from host mem.
  process data on target device
  if $i =$ *last stage in algorithm* **then**
    stream alignment locations to host mem.
  else if *intermediate data fits in off-chip mem.* **then**
    stream output data to off-chip mem.
  else
    stream data to host mem.
end

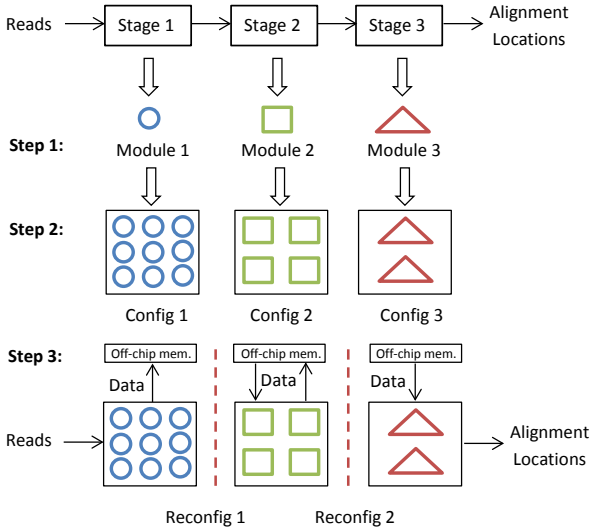

Figure 2: Multi-configuration alignment pipeline.

are negligible compared to the total runtime.

$$T = \sum_i \left(t_r + t_o + \frac{N_i t_i}{P_i} \right) \quad (2)$$

The reconfigurable property of this architecture addresses the limitations of a statically configured design. First, each configuration consists of a homogeneous array of independent modules, therefore there are no data hazards or unbalanced pipeline stages. This feature comes at the cost of concurrent processing of the algorithm stages. However, the number of modules in each algorithm stage is maximised according to the available resources on the target device fabric, which increases the intra-stage parallelism. Second, distinct configurations are created for each algorithm stage, therefore it is easier to fully map large alignment algorithms to hardware. Finally, runtime reconfiguration of the target device allows the user greater control over the alignment parameters. Configurations can be re-ordered, added, or removed at runtime, making this architecture completely modular. For example, the computational workflow can be dynamically modified based on whether certain runtime conditions are met. This feature allows highly flexible designs to be created which can address a large number of experiments.

5. FM-INDEX OPTIMISATION

In this work, the architecture in Section 4 is applied to bisulfite sequence alignment. Given the parameters for this type of alignment, our choice of alignment algorithm is the FM-index. In this section we present an algorithm optimisation to improve the performance of the FM-index.

5.1 n-step FM-index

In Algorithm 1, each step of the for-loop matches a single symbol in query Q. After $|Q|$ steps, the final interval gives all the consecutive indexes in the SA, which have Q as a prefix. Chacón et al. [5] propose the n-step FM-index, an algorithmic variation of the FM-index which reduces the number of steps required, in order to improve execution time. In essence this variation reduces the number of steps by a factor of n by allowing n symbols in Q to be matched per step. This reduction of search steps comes at the cost of increased computational complexity per step and a larger index size. We extend this work to allow for improved execution time, but with identical computational complexity to the standard FM-index search operation. The novel feature of our modification is a compression and merging step when generating the n-step FM-index, which reduces the complexity of search operations compared to previous efforts.

To generate the index, first the reference genome is compressed into a reduced bitmap. For a reference genome R = AGT\$ the corresponding reduced bitmap is R_r = [00, 01, 10, 11]. The number of bits required for each symbol is $log_2|\Sigma|$. Note that the appended \$ symbol is included in the alphabet size. To allow for n symbols to be matched per step, n BWTs are generated from R_r and the SA of R, as shown in Algorithm 3. These BWTs are merged together to form a single BWT, denoted by B, in which each element is $n \times log_2|\Sigma|$ bits.
Next, B is divided into buckets of d elements, such that the i^{th} bucket contains the sequence of elements from $B[i \cdot d]$ to $B[((i+1) \cdot d)-1]$. For each bucket a set of $|(\Sigma-1)^n|$ counters are computed using Equation 3, in which str is the reduced bitmap for each n symbol permutation of Σ (excluding the \$ symbol which does not appear in the query), i is the bucket index, and $Occ()$ and $C()$ are the two counting functions defined in Section 2. For the alphabet Σ = {A, G, T, \$} and $n = 2$, str = [0000 (AA), 0001 (AG), ..., 1010 (TT)].

$$Counters(str) = C(str) + Occ(str, (i \cdot d) - 1) \quad (3)$$

Finally, the index denoted by F is formed by interleaving the buckets of B with their corresponding counters. The index structure is illustrated in Figure 3.

Algorithm 4 shows the new procedure for searching for a query Q in a text R. It is worth noting that if $|Q|$ is not exactly divisible by n, the interval is first updated for

Algorithm 3: Algorithm to generate the merged BWT B. Note: The concatBits function concatenates the bits in the argument (left to right) from high to low order bits.

Input : SA corresponding to the text R, and reduced bitmap of text R_r
Output: B
begin
  /* generate n BWTs: $b_1, b_2, \ldots, b_n$ */
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 0$ **to** $|R_r|$ **do**
      $b_i[j] = R_r[(\text{SA}[j] - i) \text{mod} |R_r|]$
    end
  end
  /* merge BWTs to form B */
  for $i \leftarrow 0$ **to** $|R_r|$ **do**
    $B[i] = \text{concatBits}(b_n[i], b_{n-1}[i], \ldots, b_1[i])$
  end
  return B
end

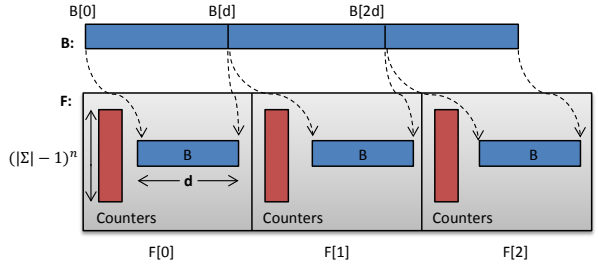

Figure 3: n-step FM-index structure.

the excess symbols using precomputed values, after which Algorithm 4 is applied.

Our modification reduces the number of steps in a search operation by a factor of n, but contains the same computational complexity as the standard FM-index (in which the index is bucketed). Consequently, the performance for search operations increases by a factor of n. This increase in performance comes at the cost of increased index size, however the cost can be alleviated by increasing the bucket size d. The total memory required for F can be calculated using Equation 4. For example, with $n = 3$ and $d = 128$, an index of the Human genome can fit in 10GB of memory.

$$M = \frac{4 \cdot |R| \cdot (|\Sigma| - 1)^n}{d} + \frac{|R| \cdot n \cdot \log_2 |\Sigma|}{8} \text{ Bytes} \quad (4)$$

6. RAMETHY: BISULFITE SEQUENCE ALIGNMENT DESIGN

In this section we present Ramethy, our bisulfite sequence alignment design. First, the alignment parameters specific to this type of alignment are described. Second, an overview of Ramethy, including the module designs and workflow is provided.

6.1 Bisulfite Sequencing Alignment

Whole-genome bisulfite sequencing involves treating the DNA with sodium bisulfite to convert all cytosines (Cs) into uracils (Us), whilst the methylated cytosines remain unchanged. In the sequencing process, all Us are identified as thymines (Ts), therefore the methylation state of the DNA can be inferred by counting the number of Cs and Ts in the treated DNA at each genomic cytosine site in the original DNA. Our work targets Methy-Pipe [10], a recently developed integrated bioinformatics pipeline for whole-genome bisulfite sequencing data analysis. Methy-Pipe uses two consecutive software modules to perform methylation data analysis. The first module is the aligner, which performs the bisulfite sequence alignment, and the second module is the data analysis module which provides various functions to facilitate downstream methylation data analysis.

Bisulfite sequencing alignment differs from the standard alignment procedure as all the Cs in the reads and reference genome are converted to Ts, reducing their alphabet to $\Sigma = \{A, G, T\}$. For seed-and-extend algorithms, this feature increases the computational workload as the average number of candidate locations which must be extended increases by a factor of $4^n/3^n$, in which n is the seed length. The alignment parameters used in Methy-Pipe are: 1) reads must be aligned to the reference genome with a maximum of two mismatches, and 2) only reads which are uniquely aligned are reported.

Algorithm 4: Algorithm for substring searching using the n-step FM-index.

Input : reduced bitmap of Q (Q_r), F, and SA corresponding to the text R
Output: Locations in R where Q is a prefix

begin
  $(low, high) \leftarrow (1, |R|)$
  for $i \leftarrow |Q| - 1$ **to** n **step** $-n$ **do**
    $str \leftarrow \text{concatBits}(Q_r[i-(n-1)], \ldots, Q_r[i-1], Q_r[i])$
    $low \leftarrow F[low - 1/d].Counters(str) +$
    $Count(str, F[low - 1/d].B, low - 1 \bmod d)$
    $high \leftarrow F[high/d].Counters(str) +$
    $Count(str, F[high/d].B, high \bmod d)$
  end
  for $i \leftarrow 0$ **to** $high - low$ **do**
    $Locations[i] \leftarrow \text{SA}[low + i]$
  end
  return *Locations*
end

function *Count(str, B, pos)*:
  $cnt \leftarrow 0$
  for $i \leftarrow 0$ **to** *pos* **do**
    if $str = B[i]$ **then**
      $cnt \leftarrow cnt + 1$
  end
  return *cnt*
end

6.2 Module Designs

To support the first alignment parameter, modules based on the n-step FM-index are designed for exact match, one mismatch and two mismatches alignment. For the proposed reconfigurable architecture there are two techniques for improving the design performance: 1) pipeline the module op-

erations to increase the throughput, and 2) replicate the modules on the target device to increase the parallelism.

For the exact match module, Algorithm 4 is mapped to hardware. The index is too large to fit in on-chip BRAM, and is therefore stored in off-chip DRAM directly attached to the FPGA device. Accessing the off-chip DRAM adds latency to the module, which coupled with the step interdependence results in a non-full pipeline. To address this issue the processing of multiple reads are interleaved such that in each pipeline stage a different read is being processed. Consequently, the pipeline can be completely filled, which allows a throughput of one SA interval update per clock cycle. This feature is implemented using a circular buffer that can store a batch of reads with a size equal to the total module latency. The cost of this throughput improvement is the additional resources required to store the batch of reads and their corresponding alignment states. In order to reduce the impact of this cost on the number of times the module can be replicated, techniques are developed to minimise the module latency. For example, the computation with the largest latency is the $Count()$ function. A binary adder tree is developed to count the occurrences in parallel so that the latency for this computation can be reduced by a factor of $(\log_2 d)/d$ in comparison to the sequential version.

Figure 4 illustrates the high level design for the exact match module. In each cycle the index elements $F[low-1/d]$ and $F[high/d]$ for a given read are accessed from off-chip DRAM via an intermediate data buffer. The index data, along with the corresponding read symbols and the current interval are used to compute the new interval. The new values of low and $high$ are stored in the circular buffer (overwriting the previous values), and $low - 1/d$ and $high/d$ are streamed as addresses to the off-chip memory command generator. If the required number of alignment steps have been performed, the final interval is transferred to host memory.

The one and two mismatches modules are based on the exact match module with additional logic incorporated to control the backtracking. To support the second alignment parameter, all possible mismatch positions in the read must be tested to ensure it is uniquely aligned. This specification is covered for by a breadth-first backtracking strategy. A challenge with using backtracking for approximate matching is that the number of steps required to align a read increases exponentially with the number of mismatches permitted. To address this challenge we incorporate features from the 2-way BWT algorithm into our design. The basis of this algorithm comes from the property that in each step of the for-loop of Algorithm 4, the interval can only decrease or remain the same size. By constraining the mismatch position in the read, long segments of the read can be initially exactly matched to the reference genome, reducing the search space size. As a result, the number of steps spent on aligning read permutations which do not occur in the reference genome is substantially reduced. Each mismatch module has a number of computational phases according to where the mismatch position is constrained. These positions are chosen in order to maximise the segment of the read which can be exactly matched. Figure 5 illustrates the different computational phases for the one and two mismatches modules. To permit forward searching, the index of the reversed reference genome is used.

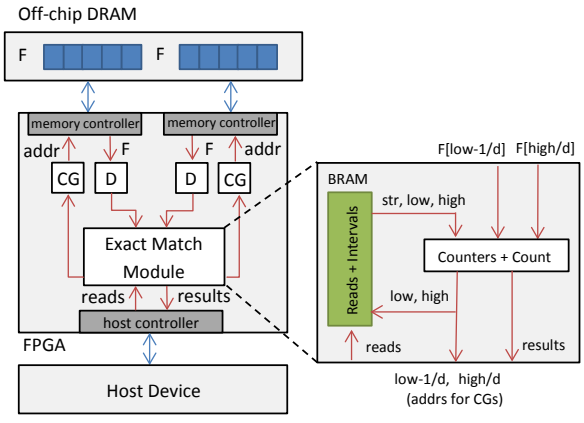

Figure 4: Exact match module design. In this Figure, data buffers are denoted by D, and memory command generators are denoted by CG.

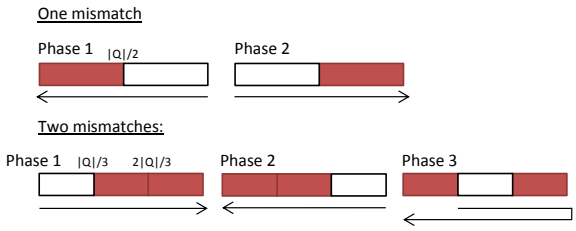

Figure 5: Computational phases for the one and two mismatches modules. In this diagram, a read Q is represented by a rectangle. The non-shaded segments are exact match regions, whilst the shaded segments are regions where mismatch/s are tested for. The arrows indicate the search direction.

6.3 Ramethy Workflow

In accordance with the reconfigurable architecture presented in Section 4, each module is replicated as many times as possible to create distinct FPGA configurations. Prior to alignment, the reads are loaded onto the host machine memory and are compressed into a reduced bitmap. For reads whose length is not exactly divisible by the index parameter n, the initial interval is updated for the excess symbols using precomputed values, otherwise it is set to $(1, |R|)$. Once this processing is complete, the FPGA device is configured with the exact match configuration. The compressed reads along with their initial interval are streamed to the FPGA device where they are processed concurrently by the modules. The final interval for each read is streamed back to the host, which determines if it is aligned ($low \leq high$) or unaligned ($low > high$). Next, the FPGA device is reconfigured with the one mismatch configuration. The unaligned reads are streamed to the FPGA device where they are processed concurrently by the modules according to the computational phases in Figure 5. If a read can be aligned, a single interval is streamed back to the host, as well as the total number of alignment hits detected (in order to determine if the read is uniquely aligned). This step is then repeated again for the two mismatches configuration. After alignment, the host converts the intervals for the aligned reads into reference genome coordinates using the SA. It is

worth noting that most of the software processing time can be hidden by the hardware processing.

7. EVALUATION

In this section we evaluate the performance of Ramethy on the Maxeler MPC-X1000 dataflow node. The runtime, energy consumption and alignment accuracy are compared to the fastest CPU, GPU and FPGA-based alignment programs currently available.

7.1 Maxeler Platform

The MPC-X1000 provides up to 8 dataflow engines (DFEs) in a 1U form factor with power consumption comparable to a single high-end server. Each DFE comprises a single Altera Stratix V FPGA connected to 48GB of DRAM. The DRAM consists of six 8GB memory modules, which are coupled together to give a word length of 384 Bytes. A single memory controller is used to manage the read and write operations. The DFEs are a shared resource on a network and are connected to a CPU host machine via Infiniband. To implement a design on the Maxeler Platform, first the software code is expressed as a dataflow design using the MaxJ language, based on the Java language. The MaxCompiler then maps the design into an FPGA configuration and enables its use from a host application.

The memory architecture of the MPC-X1000 coupled with the random access pattern of the FM-index are the limiting factors of performance. Our implementation of Ramethy is restricted to just a single module per configuration running with an off-chip memory bandwidth of approximately 4.2 GB/s (11% of the theoretical peak 38.5 GB/s). To improve the performance, the memory modules can be decoupled to allow up to six memory controllers per DFE. This modification permits multiple modules in each configuration by allowing parallel read operations to the index. Although this solution has not been implemented, we use it to provide an upper bound performance estimate for the MPC-X1000.

7.2 Experimental Parameters

Ramethy is implemented on a MPC-X1000 with 8 DFEs. The index is constructed with the parameter values $n = 3$, $d = 368$ and $|\Sigma| = 4$. Consequently, an index of the Human genome can fit in 3.3GB of memory. Our experimental results include two performance measurements for Ramethy. The first (denoted by v1) is based on actual measurements obtained from our system, in which Ramethy is mapped onto 8 DFEs with a population of one module per configuration. The second (denoted by v2) is an upper bound estimate for the MPC-X1000 in which the memory modules attached to each DFE are decoupled, allowing for 3 modules per configuration. Note that each module requires two memory controllers to access the index elements $F[low - 1/d]$ and $F[high/d]$ in parallel, and there are six memory controllers per DFE.

The runtime, energy consumption and accuracy of Ramethy is compared to those of soap2 running on a 1U rack server with dual Intel Xeon E5-2650 CPUs and soap3-dp running on an NVIDIA GTX-580 GPU. These programs are used in our comparison as: a) they are widely regarded as the fastest CPU and GPU-based alignment tools currently available, and b) the bisulfite sequence alignment program developed for Methy-Pipe is based on soap2, with future releases utilising soap3-dp. It is worth noting that in all runtime measurements reported, only the alignment time is considered.

```
$ ./soap -a reads.fastq -D index -v 2
  -p 16 -o output

$ ./soap3-dp single index reads -s 2 -h 3
  -o output
```

In the following experiments Chromosome 22 of the Human genome is used as the reference genome. Sherman, a bisulfite sequencing read simulator, is used to generate single-end reads. The following command is used to simulate reads with similar properties to experimental data:

```
$ ./Sherman -q 40 -I 75 -CG 20 -CH 98 -e 0
```

Given the relatively small workload used (3% of a full alignment workload), the reconfiguration time (3-4 seconds per configuration) for the MPC-X1000 is not included in the runtime measurements of Ramethy, as it would introduce a large negative bias to the results. With a full alignment workload of 300M reads the reconfiguration time would amount to approximately 3.5% of the alignment time.

7.3 Experimental Results

In the first experiment, the alignment performance of Ramethy is measured for exact match, one mismatch and two mismatches. Three data sets are generated in which 10M reads of 75 bases are directly sampled from the reference genome. Mismatches are inserted at random positions in the reads according to the number being tested. The graph in Figure 6 indicates that the exact match, one mismatch and two mismatches configurations are all faster at aligning reads than both soap2 and soap3-dp. The largest performance improvement is with the exact match configuration, which is 45.1 times and 10.1 times faster than soap2 and soap3-dp respectively. The mismatch configurations show less substantial improvements as additional time is spent processing the reads with each alignment phase and the previous configuration(s). The upper bound performance estimates indicate that a high population of modules in each configuration yield significant performance improvements over soap2 and soap3-dp. For example, Ramethy can be up to 270.6 times and 60.5 times faster at exactly aligning reads than soap2 and soap3-dp respectively.

In the next experiment the resource usage of each module is analysed. The final resource usage for look-up tables

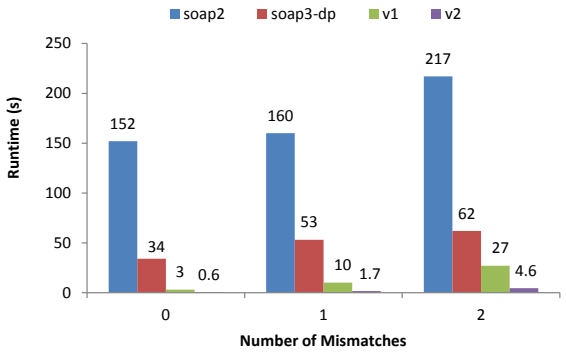

Figure 6: Configuration performance compared to soap2 and soap3-dp.

(LUTs), flip-flops (FFs), and BRAMs, as well as the achievable clock frequency are recorded from the build report of Ramethy. Table 2 indicates that all three modules utilise approximately 24–27% of the available resources on the device fabric. If each module is replicated on the device fabric as many times as possible given the available resources, there could be up to 3 modules per configuration, which is identical to the populations for the upper bound estimate. Aside from the number of memory controllers supported, the critical resource for each module is BRAMs. An explanation for this result is that the circular buffer used to store the batch of reads and the corresponding alignment states is implemented using BRAMs. Future implementations could benefit from FPGA devices with increased BRAM and logic resources at the cost of DSP blocks (which are entirely unused), which would allow a higher population of modules per configuration.

Next, the performance of Ramethy is measured for realistic bisulfite sequenced data. The Sherman read simulator is used to generate 10M reads of 75 bases from the reference genome. To make the data as realistic as possible, the bisulfite conversion rate of the reads is adjusted to values typically seen in real experiments. For completeness, the alignment time of Ramethy is also compared to notable hardware-based designs, including the Smith-Waterman design in [19] and the FM-index design in [8]. Since different data sets and alignment parameters are used in these efforts, we define a normalised performance merit, bases aligned per second (bps), to allow for a fairer comparison.

$$\text{bps} = \frac{\text{read length} \times \text{read count}}{\text{alignment time}} \tag{5}$$

Table 3 indicates that Ramethy is approximately 14.9 times faster than soap2 running on dual Intel Xeon E5-2650 CPUs and 3.8 times faster than soap3-dp running on a NVIDIA GTX-580 GPU. The upper bound performance estimate for the MPC-X1000 indicates that Ramethy can achieve a maximum speedup of 88.4 times and 22.6 times compared to soap2 and soap3-dp respectively. All of these systems are housed in a 1U rack unit, suggesting that Ramethy offers the highest performance per unit volume.

To evaluate the proposed reconfigurable architecture, the performance of Ramethy is compared to a statically configured design. Given the available resources on the MPC-X1000, and the process time for each configuration, the optimal static design would appear as illustrated in Figure 7. Assuming there is zero communication overhead between modules and the modules are never stalled, the maximum performance for a static design is given by Equation 6, in which N_i is the number of reads processed by the i mismatch module, t_i is the time for the corresponding module to process a read and P_i is the module population.

$$T = max\left(\frac{N_0 t_0}{P_0}, \frac{N_1 t_1}{P_1}, \frac{N_2 t_2}{P_2}\right) \tag{6}$$

Table 3 indicates that Ramethy exceeds the maximum performance for a static design. When accounting for the reconfiguration time in a full alignment workload, Ramethy is 11% faster than the static design.

Module	Clock (MHz)	LUT	FF	BRAM
EM	150	70676 (27%)	124224 (24%)	690 (27%)
OM	150	70267 (27%)	124001 (24%)	621 (24%)
TM	150	70858 (27%)	125624 (24%)	633 (25%)

Table 2: Module resource usage on a Altera Stratix V FPGA. Note: EM, OM, and TM denote exact match, one mismatch and two mismatches respectively.

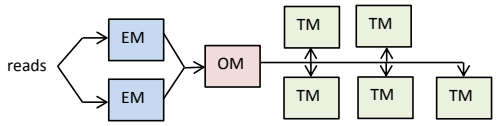

Figure 7: Optimal statically configured design. Note: EM, OM, and TM denote exact match, one mismatch and two mismatches respectively, and each module is mapped to a single DFE.

Given that the alignment time scales linearly with the read count, the runtime of Ramethy can be linearly extrapolated to that of a typical alignment workload. Experiments typically sequence DNA using a single flow cell lane, which produces approximately 300M reads with 75 bases. Ramethy can align this volume of reads in 6 minutes, whilst the upper bound estimate gives a minimum time of just over a minute. Considering that an alignment workload this size can take hours for a high-end server to complete, Ramethy can significantly shorten the diagnosis time, which would allow faster responses and increase the number of patient samples that can be analysed per day.

The bps values in Table 3 indicate that Ramethy is approximately 5.0 times faster than the design in [8], yet 1.8 times slower than the design in [19]. The upper bound performance estimate for the MPC-X1000 indicates that if the platform-specific limitations are addressed, Ramethy can achieve 3.3 times speedup compared to the design in [19]. Whilst it is difficult to directly compare these hardware-based designs fairly, it can be noted that Ramethy has three disitinct advantages over the FM-index based designs in previous efforts. First, the n-step FM-index modification reduces the required number of steps in each alignment with no increase in the computational complexity of each step. Second, the incorporation of the 2-way BWT into the one and two mismatches modules greatly increases the approximate matching performance. Third, the breadth-first backtracking strategy improves the alignment accuracy, which is identical to that of soap2 and soap3-dp.

In the final experiment the energy consumption of Ramethy is measured. Power values are taken from the vendors product information for the CPU and GPU devices, and directly from the operating system in the case of the MPC-X1000 dataflow node. Table 4 indicates that Ramethy consumes approximately an order of magnitude less energy than

Program	Total Power (W)	Energy Consumption (kJ)
soap2	190	31.9
soap3-dp	244	10.5
Ramethy v1	72	1.5

Table 4: Energy consumption.

Table 3: Performance comparison.

program	read length	read count (M)	platform	clock freq. (MHz)	devices	runtime (s)	bps (million)	speedup
soap2	75	10	Intel E5-2650	2000	2	168	4.5	1.0x
soap3-dp	75	10	NVIDIA GTX-580	772	1	43	17.4	3.9x
[8]	101	18	Convey HC-1	150	4	138	13.2	3.0x
[19]	76	54	Pico M-503	250	8	34	120	26.8x
static design	75	10	MPC-X1000	150	8	12.8	58.5	13.1x
Ramethy v1	75	10	MPC-X1000	150	8	11.3	66.4	14.9x
Ramethy v2	75	10	MPC-X1000	150	8	1.9	395	88.4x

soap2 and soap3-dp. This result can be explained by the low operational clock frequencies which the FPGAs are run at, coupled with the shorter runtime. This result suggests that the relatively high initial cost of the MPC-X1000 can be amortised given the much lower operational energy costs.

8. CONCLUSION

This paper presents a novel reconfigurable architecture for accelerating sequence alignment. This architecture is applied to bisulfite sequence alignment, a bottleneck in recently developed bioinformatics pipelines for cancer and non-invasive prenatal diagnostics. Our design, referred to as Ramethy, is based on the FM-index which we optimise to reduce the amount of search steps and improve approximate matching performance. Ramethy is implemented on the Maxeler MPCX-1000 dataflow node and measured results show a 14.9 times speedup compared to soap2, and 3.8 times speedup compared to soap3-dp. In addition to runtime, Ramethy consumes over an order of magnitude lower energy, and has an accuracy identical to soap2 and soap3-dp, making it a strong candidate for integration into bioinformatics pipelines. Future work involves accelerating the other stages of the Methy-Pipe application, exploring how the proposed reconfigurable architecture can be applied to other bioinformatics pipelines, and automating the implementation of such pipelines from a high-level description.

Acknowledgement

We thank Dennis Lo, Rossa Chiu and Alex Ross for their advice and encouragement. This work was supported in part by Maxeler University Programme, Xilinx, UK EPSRC, the European Union Seventh Framework Programme under grant agreement number 257906, 287804 and 318521, and the HiPEAC NoE.

9. REFERENCES

[1] J. Arram et al. Hardware acceleration of genetic sequence alignment. In *ARC*, pages 13–24, 2013.

[2] J. Arram et al. Reconfigurable acceleration of short read mapping. In *FCCM*, pages 210–217, 2013.

[3] J. Arram et al. Reconfigurable filtered acceleration of short read alignment. In *FPT*, pages 438–441, 2013.

[4] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

[5] A. Chacón et al. n-step fm-index for faster pattern matching. *Procedia Computer Science*, 18(0):70 – 79, 2013.

[6] K. C. Chan et al. Noninvasive detection of cancer-associated genome-wide hypomethylation and copy number aberrations by plasma DNA bisulfite sequencing. *Proc. Natl. Acad. Sci. U.S.A.*, 110(47):18761–18768, Nov 2013.

[7] E. Fernandez et al. String matching in hardware using the FM-index. In *FCCM*, pages 218–225, May 2011.

[8] E. Fernandez et al. Multithreaded FPGA acceleration of DNA sequence mapping. In *HPEC*, pages 1–6, Sept 2012.

[9] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. SODA*, pages 269–278, 2001.

[10] P. Jiang et al. Methy-pipe: An integrated bioinformatics pipeline for whole genome bisulfite sequencing data analysis. *PLoS ONE*, 9(6):e100360, 06 2014.

[11] T. W. Lam et al. High throughput short read alignment via bi-directional BWT. In *BIBM*, pages 31–36, November 2009.

[12] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nat. Methods*, 9(4):357–359, Apr 2012.

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

[14] R. Li et al. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[15] Y. Liu et al. CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform. *Bioinformatics*, 28(14):1830–1837, Jul 2012.

[16] F. M. Lun et al. Noninvasive prenatal methylomic analysis by genomewide bisulfite sequencing of maternal plasma DNA. *Clin. Chem.*, 59(11):1583–1594, Nov 2013.

[17] R. Luo et al. SOAP3-dp: fast, accurate and sensitive GPU-based short read aligner. *PLoS ONE*, 8(5):e65632, 2013.

[18] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 319–327, 1990.

[19] C. Olson et al. Hardware acceleration of short read mapping. In *FCCM*, pages 161–168, April 2012.

[20] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147(1):195–197, Mar 1981.