Leveraging FPGAs for Accelerating Short Read Alignment

James Arram, Thomas Kaplan, Wayne Luk, and Peiyong Jiang

Abstract—One of the key challenges facing genomics today is how to efficiently analyze the massive amounts of data produced by next-generation sequencing platforms. With general-purpose computing systems struggling to address this challenge, specialized processors such as the Field-Programmable Gate Array (FPGA) are receiving growing interest. The means by which to leverage this technology for accelerating genomic data analysis is however largely unexplored. In this paper, we present a runtime reconfigurable architecture for accelerating short read alignment using FPGAs. This architecture exploits the reconfigurability of FPGAs to allow the development of fast yet flexible alignment designs. We apply this architecture to develop an alignment design which supports exact and approximate alignment with up to two mismatches. Our design is based on the FM-index, with optimizations to improve the alignment performance. In particular, the n-step FM-index, index oversampling, a seed-and-compare stage, and bi-directional backtracking are included. Our design is implemented and evaluated on a 1U Maxeler MPC-X2000 dataflow node with eight Altera Stratix-V FPGAs. Measurements show that our design is 28 times faster than Bowtie2 running with 16 threads on dual Intel Xeon E5-2640 CPUs, and nine times faster than Soap3-dp running on an NVIDIA Tesla C2070 GPU.

Index Terms—FM-index, FPGA, runtime reconfiguration, short read alignment

1 Introduction

In recent years the advances in throughput of next-generation sequencing (NGS) platforms has far exceeded Moore's Law [11]. The latest NGS platforms available are able to generate Terabytes of data in a single run, and their throughput is expected to increase 3-5× each year. This rapidly growing amount of sequencing data has put considerable strain on the computing systems used for subsequent analysis, with many sequence analysis pipelines requiring hours or even days to transform the raw data into appropriate information for diagnosis or research.

The bottleneck of most sequence analysis pipelines is short read alignment. In this stage the short fragments of DNA produced by NGS platforms, called reads, are mapped to locations in a known reference genome, as shown in Fig. 1. Due to incorrect base calls and genetic diversity between the sample DNA and reference genome, approximate alignment must also be considered. This is achieved by permitting a number of mismatches, insertions and deletions in the reads.

There currently exist many software tools for short read alignment, including Soap2 [15], BWA [14], and Bowtie2 [13]. Despite featuring highly optimised algorithms, these tools can take many hours to align the short read data. For example, Soap2 takes over 5 hours to align 300M reads when run on a system with dual 12-core Intel Xeon processors and 100 GB of RAM. A common solution for accelerating alignment is to use extensive computational resources. Examples of this are the 1,000 genome project [1] which uses a 1192-processor cluster, and the BGI Bio-cloud [4] computing platform which has a current total of 14,774 processors delivering 157T flops of performance. Given the projected advances in throughput of NGS platforms, the cost of building and running such systems is becoming increasingly impractical.

Reconfigurable hardware, such as the Field-Programmable Gate Array (FPGA), is a promising candidate for accelerating short read alignment. The multiple levels of exploitable parallelism can provide substantial application speed-up, whilst the low operational clock frequencies allow reduced energy consumption, and high rack unit densities. There are various efforts related to accelerating short read alignment using FPGAs [9], [20]. Although these designs perform alignment faster than most of the software tools currently available, their speed often comes at the cost of accuracy and functionality. As a result, few FPGA-based tools have been fully integrated into sequence analysis pipelines.

In this work we show how FPGAs can be leveraged to accelerate short read alignment. The major contributions of this work include:

- A runtime reconfigurable architecture for accelerating short read alignment using FPGAs. This architecture exploits the reconfigurability of FPGAs to allow the development of fast yet flexible alignment designs.
- An application of this architecture to develop an alignment design which supports exact and approximate alignment with up to two mismatches. Our design is based on the FM-index, with optimisations to improve the alignment performance. In particular, the n-step FM-index, index oversampling, a seed-

- *J. Arram, T. Kaplan, and W. Luk are with the Department of Computing, Imperial College London, London SW7 2BZ, United Kingdom. E-mail: {jma11, wl}@imperial.ac.uk, tk2112@doc.ic.ac.uk.*
- *P. Jiang is with the Department of Chemical Pathology, The Chinese University of Hong Kong, Hong Kong. E-mail: jiangpeiyong@cuhk.edu.hk.*

*Manuscript received 10 May 2015; revised 22 Jan. 2016; accepted 3 Feb. 2016. Date of publication 29 Feb. 2016; date of current version 1 June 2017.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCBB.2016.2535385*

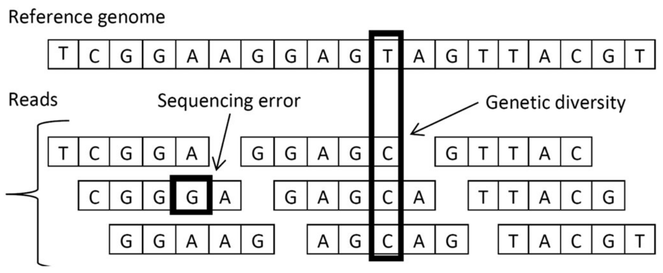


Fig. 1. Alignment of reads to a reference genome.

and-compare stage, and bi-directional backtracking are included.

- An implementation of our design on a Maxeler MPC-X2000 system. The performance is evaluated and compared to the fastest software alignment tools currently available.

2 Background

In this section we present background information on the FM-index and its search operation. In addition, we provide a brief overview of FPGAs.

2.1 FM-Index

The FM-index [10] is a full-text compressed index which supports substring searching in linear time with respect to the substring length. The FM-index is built upon the Burrows-Wheeler transform (BWT) [5], a permutation of a text generated from its Suffix Array (SA) [18].

The SA of a text R is a lexicographically sorted array of the suffixes of R, where each suffix is represented by its position in R. The SA interval (low, $high$) covers a range of indices in the SA where the suffixes have the same prefix. The pointer low gives the index in the SA where the pattern is first found as a prefix, and the pointer $high$ gives the index after the one where the pattern is last found. Fig. 2a illustrates the construction of the SA for a text. In this example the SA interval for the substring A is (1, 4). The result of searching for a substring can be represented as a SA interval. If $low < high$, the substring occurs in the text. Conversely, if $low \geq high$, the substring does not occur.

The FM-index is built upon the BWT, a transformation which generates a permutation of the symbols in a text. Each position in the BWT is computed using the relationship: $\mathrm{BWT}_i = R[(\mathrm{SA}_i - 1) \bmod |R|]$. Fig. 2a illustrates the construction of the BWT from the SA of a text. The FM-index supports substring searching through two functions performed on the BWT. $\mathrm{Count}(s)$ returns the number of symbols in the BWT which are lexicographically smaller than the symbol s. $\mathrm{Occ}(s, i)$ returns the number of occurrences of the symbol s in the BWT from positions 0 to $i-1$. The values of these functions are precomputed and stored as arrays, as shown in Fig. 2b. To compress the size of the FM-index, the Occ array is sampled into buckets of width d. In this procedure the Occ values are stored every d positions as markers, reducing the array size by a factor of d. The Occ values omitted are reconstructed by summing the previous marker and the result of counting the occurrence of the remaining positions directly from the BWT. To simplify the search operation the Count value for each symbol

(a) R = BANANA$

i	SA	suffix
0	6	$
1	5	A$
2	3	ANA$
3	1	ANANA$
4	0	BANANA$
5	4	NA$
6	2	NANA$

BWT = ANNB$AA

(b) $\mathrm{Occ}(s, i)$

i	A	B	N
0	0	0	0
1	1	0	0
2	1	0	1
3	1	0	2
4	1	1	2
5	1	1	2
6	2	1	2
7	3	1	2

$\mathrm{Count}(s)$

A	B	N
1	4	5

(c) FM-index ($d = 4$)

0

Markers

A	B	N
1	4	5

BWT

A	N	N	B

1

Markers

A	B	N
2	5	7

BWT

$	A	A	-

Fig. 2. Generating the FM-index. Note $ is the terminal symbol, the smallest symbol lexicographically.

can be added to the corresponding markers. The markers and the section of the BWT covered are then interleaved to form the FM-index structure, as shown in Fig. 2c.

Algorithm 1. FM-index Search Operation

Input: substring Q, FM-index F with bucket width d, and the suffix array of the text R
Output: positions in R where Q occurs
Procedure: $\Phi(F, s, i)$ – returns $\mathrm{Occ}(s, i)$ from FM-index bucket F

```
 1: $low \leftarrow 0$                              ▷ initialise suffix array interval
 2: $high \leftarrow \max(\mathrm{Occ}_i)$
 3: for $i \leftarrow |Q| - 1$ to 0 do              ▷ update suffix array interval
 4:     $low \leftarrow \Phi(F[low/d], Q_i, low)$
 5:     $high \leftarrow \Phi(F[high/d], Q_i, high)$
 6:     if $low \geq high$ then                     ▷ terminate if symbol not matched
 7:         end
 8:     end if
 9: end for
10: for $i \leftarrow low$ to $high - 1$ do           ▷ get reference positions
11:     positions $\leftarrow \mathrm{SA}_i$
12: end for
13: procedure $\Phi$ F, s, i                        ▷ get $\mathrm{Occ}(s, i)$ from $F$ bucket
14:     $marker \leftarrow (F \rightarrow \mathrm{Markers}[s])$    ▷ get marker value
15:     $count \leftarrow 0$
16:     for $j \leftarrow 0$ to $j < i \bmod d$ do       ▷ count from BWT
17:         if $s = (F \rightarrow \mathrm{BWT}_j)$ then
18:             $cnt \leftarrow cnt + 1$
19:         end if
20:     end for
21:     return $marker + count$
22: end procedure
```

The FM-index search operation is described in Algorithm 1. Concisely written, low and $high$ are first initialised

Substring: `ANA`
Text: `BANANA`

Iteration 1	Iteration 2
symbol = `A`	symbol = `N`
$(0, 7) \rightarrow (1, 4)$	$(1, 4) \rightarrow (5, 7)$
Iteration 3	Convert
symbol = `A`	SA Interval $\rightarrow$ Pos
$(5, 7) \rightarrow (2, 4)$	$(2, 4) = 3, 1$

Fig. 3. FM-index search operation example.

to the minimum and maximum indices of the Occ array respectively. Then moving from the last symbol in the read to the first (a backward search), the SA interval is updated using the equations in lines 4 and 5. After the final update, the SA interval gives the range of indices in the SA where the suffixes have the substring as a prefix. These indices are then converted to reference genome positions using the SA. Fig. 3 illustrates an example of the FM-index search operation. Backtracking can be used to extend the search operation to support approximate alignment. In this approach the short read is permuted using edit operations (substitutions, insertions or deletions). A stack is used to store the position and symbol of each edit. If the permuted read is unable to be aligned, the state is restored from the stack and a new edit operation is performed.

2.2 Field-Programmable Gate Arrays

FPGAs are integrated circuits composed of a matrix of configurable logic blocks and interconnects. These logic blocks and interconnects can be programmed to create a digital circuit functionally equivalent to an application or procedure. The circuit design is typically coded in a hardware description language, which is subsequently mapped into an FPGA configuration. FPGAs support runtime reconfiguration, which is the ability to dynamically switch between configurations. This feature allows developers to change the functionality of the FPGA on-the-fly. The advantages of using FPGAs for application acceleration are: 1) substantial speed-up can be achieved through highly parallel custom computations, and 2) energy and power consumption is much lower than that for a CPU.

3 Related Work

There currently exist many software tools for short read alignment, including Soap2 [15], BWA [14], and Bowtie2 [13]. These tools use either index based algorithms, such as the FM-index, or dynamic programming algorithms, such as the Smith-Waterman algorithm [21], to perform alignment.

As a response to the rapidly increasing NGS platform throughput, GPU-based tools have been developed to improve the alignment performance. Notable GPU-based tools include Soap3-dp [17] and CUSHAW [16], which are reported to perform up to 10 times faster than the CPU-based tools.

There are various efforts related to accelerating sequence alignment using FPGAs, among which accelerating the Smith-Waterman alignment algorithm is the most common approach. Here we summarise the novel aspects of some notable efforts.

TABLE 1
Architecture Analysis Symbol Definitions

Symbol	Definition
T	Alignment time
A	Total resources available on the FPGA fabric
P_i	Population of a given module on the FPGA fabric
r_i	Number of resources required by a given module
N_i	Number of data items processed by a given module
t_i	Time for a given module to process an item of data
t_r	FPGA reconfiguration time
t_t	Data transfer time to the FPGA device

Olson et al. [20] accelerate the Smith-Waterman algorithm using FPGAs. In this work, both the seed location and score table computation are performed in hardware. The design is partitioned into 8 Pico M-503 boards, each comprising a single Xilinx Virtex-6 FPGA. This eight-FPGA system can align 50 million reads in 34 seconds.

Fernandez et al. [8], [9] accelerate the FM-index using FPGAs. 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 1,000 reads in 60.2us. In the second work, their previous design is extended to allow for approximate alignment. Here multiple exact alignment modules align the permuted reads. The design is implemented on the Convey HC-1 platform and can align 18M reads in 138 seconds.

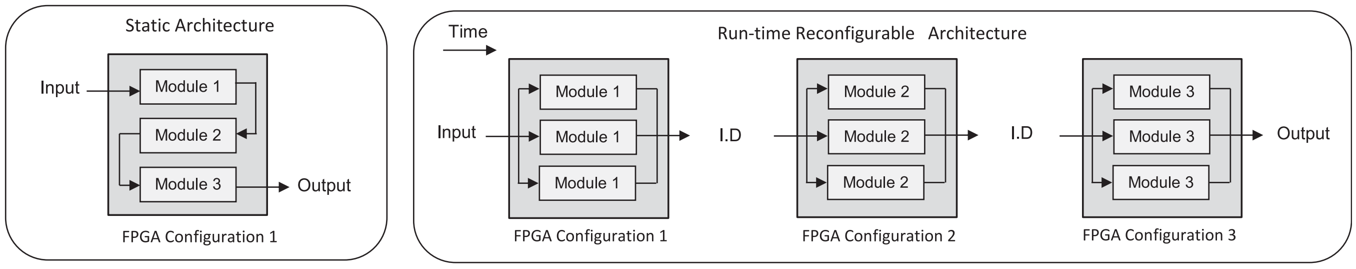
This work expands on our previous efforts in accelerating sequence alignment using FPGAs [2], [3]. In these efforts we present: 1) a hardware acceleration of the FM-index for exact and approximate alignment, and 2) design space exploration for FPGA-based alignment architectures. This work consolidates our previous efforts, and presents a new alignment design which supports exact and approximate alignment with up to 2 mismatches.

4 Runtime Reconfigurable Architecture

In this section we present a runtime reconfigurable architecture for accelerating short read alignment using FPGAs. This architecture exploits the reconfigurability of FPGAs to allow the development of fast yet flexible alignment designs. Table 1 defines the symbols used in our architecture analysis.

4.1 Motivation

In previous related efforts, the FPGA fabric is configured with a design functionally equivalent to an alignment algorithm. These designs are typically composed of several interlinked modules which correspond to the stages that make up the alignment algorithm. For example, a dynamic programming design could consist of modules for seed location, score table calculation, and trace-back. Since the FPGA fabric is configured with a single design, we define this a static architecture. Fig. 4a illustrates a static architecture for an alignment design comprising three modules. The population of a given module and best case alignment time for a static architecture are given by Equations (1) and (2) respectively. Note that in a static architecture the modules process the data concurrently,



(a) Static architecture.

(b) Run-time reconfigurable architecture. Note: I.D denotes intermediate data.

Fig. 4. FPGA alignment architectures.

thus the best case alignment time is the longest module process time.

$$P_i = \frac{A - \sum_{j \neq i} P_j \cdot r_j}{r_i} \tag{1}$$

$$T = \max\left(\frac{N_1 t_1}{P_1}, \frac{N_2 t_2}{P_2}, \dots\right). \tag{2}$$

We observe a number of limitations which can reduce the alignment performance for designs with a static architecture:

1) There may be insufficient resources available on the FPGA fabric to fit the entire alignment design. In this case a subset of the modules must be performed in software, which reduces the overall performance according to Amdahl's Law.
2) In order to improve the throughput, slower modules are replicated on the FPGA fabric. With limited available resources it may be impossible to replicate these modules the required amount of times, resulting in an unbalanced pipeline of modules.
3) Alignment algorithms feature many data hazard points where the processing of an item by a given module is dependant on the result from a previous computation; consequently, some modules may exist in an idle state for the majority of the runtime. These idle modules consume FPGA resources which could be better utilised in increasing the module population of the more computationally intensive stages. For example, in an FM-index-based design a short read is only approximately aligned if it cannot be exact aligned. Given the typical error rates observed, approximate alignment modules consume the majority of the FPGA resources but only process a small fraction of the data.
4) Changes to the alignment parameters, such as the number or type of edits permitted require modules to be modified; consequently, the entire design must be rebuilt which can take many hours or even days to perform. An alignment design with a static architecture can therefore only target a small portion of the parameter space.

4.2 Runtime Reconfigurable Architecture Overview

In this work we propose a runtime reconfigurable architecture for accelerating short read alignment using FPGAs. In this architecture each module in the alignment design is mapped to its own FPGA configuration, where it is replicated as many times as possible according to the resources available on the FPGA fabric. Runtime reconfiguration is used to dynamically switch between the configurations. Fig. 4b illustrates a runtime reconfigurable architecture for an alignment design comprising three modules. Since each configuration contains only a single module type with no interconnections, the performance is not reduced by the limitations observed for a static architecture.

The runtime reconfigurable architecture has the following operation cycle: for each stage in the alignment algorithm the corresponding configuration is loaded onto the FPGA fabric. Input data, or intermediate data from the previous stage are streamed to the FPGA where they are processed in parallel by the modules. Output data are stored in off-chip memory directly attached to the FPGA device, or host memory. The configuration corresponding to the next stage in the alignment algorithm is then loaded onto the FPGA fabric and the process is repeated until the final stage is complete.

The population of a given module and the alignment time for a runtime reconfigurable architecture are given by Equations (3) and (4) respectively. The population of each module in the alignment design is maximised at the cost of concurrent execution; consequently, high performance is achieved through the parallel processing of data by the modules. The overheads of the runtime reconfigurable architecture are the reconfiguration time, and the data transfer time. Given that reconfiguration takes only a few seconds, and the streaming interface bandwidths are in the order of GB/s, these overheads do not substantially impact the alignment time.

$$P_i = \frac{A}{r_i} \tag{3}$$

$$T = \sum_i \left(\frac{N_i t_i}{P_i} + t_r + t_t\right). \tag{4}$$

A runtime reconfigurable architecture allows increased alignment flexibility compared to a static architecture. Configurations can be dynamically re-ordered, inserted and deleted without having to rebuild the design. For example, if the alignment percentage is below an acceptable value, additional configurations can be dynamically inserted to the alignment design to process the remaining short reads. With a comprehensive library of configurations, a runtime reconfigurable architecture can efficiently target a large portion of the parameter space.

5 Alignment Design

In this section we present a short read alignment design which supports exact and approximate alignment with up

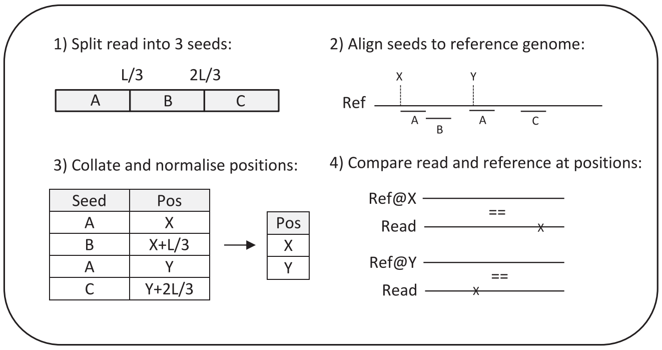

1) Split read into 3 seeds:
L/3
2L/3
A
B
C
2) Align seeds to reference genome:
X
Y
Ref
A
B
A
C
3) Collate and normalise positions:
Seed
Pos
A X
B X+L/3
A Y
C Y+2L/3
Pos
X
Y
4) Compare read and reference at positions:
Ref@X
==
Read
x
Ref@Y
==
Read
x

Fig. 5. Seed-and-compare stage.

to two mismatches. Our design is based on the FM-index search operation, with optimisations to improve the alignment performance.

5.1 Design Overview

The proposed alignment design comprises four stages: exact alignment, seed-and-compare, one mismatch alignment, and two mismatches alignment. The reads are first processed by the exact alignment module. For most sequencing data, approximately 70 percent of the reads should exactly align to the reference genome. Any unaligned reads are processed by the seed alignment module, which splits the reads into three non-overlapping seeds and independently aligns them to the reference genome. The computation then branches according to the total number of positions the seeds align to, denoted by n_{pos}: 1) If n_{pos} is zero, the short read is unable to be aligned to the reference genome with up to two mismatches, therefore it is omitted from any further processing, 2) If n_{pos} is less than a specified threshold value, the short read is directly compared to the reference genome at each position using the compare module, 3) If n_{pos} exceeds the threshold value, the short reads are processed by the one and two mismatch stages. The threshold value is used to balance the amount of processing done by the compare module and the one and two mismatch alignment modules. For a position threshold of 5, we observe that the majority of reads can be aligned using the compare module. This reduces the of the amount of processing done by the comparatively slower one and two mismatch alignment modules. Fig. 5 illustrates the seed-and-compare stage. Aside from the compare module, all the modules are based on the FM-index search operation.

5.2 Algorithmic Optimizations

In this section we present several algorithmic optimisations that are included to improve the performance of the proposed alignment design. These optimisations target two bottlenecks of the FM-index search operation: 1) random memory access to the index, and 2) the large search space when using a backtracking approach for approximate alignment. Table 2 defines the symbols used in the optimisation analysis.

n-step FM-index. In the FM-index search operation, two memory accesses ($F[low/d]$ and $F[high/d]$) are required to update the SA interval for each symbol in the read; consequently, $2L$ memory accesses are required to exact align a

TABLE 2
Optimization Analysis Symbol Definitions

Symbol	Definition
L	Number of symbols in read
R	Number of symbols in reference genome
Σ	Alphabet size of reference genome
n	Step size
X	Number of symbols before *low* and *high* point to the same FM-index bucket
d	FM-index bucket width
f	Bucket sampling factor
k	Number of mismatches permitted

read. Due to its large size, the FM-index is stored in off-chip DRAM. Given that the access latency to off-chip DRAM is in the order of hundreds of cycles, and the access pattern is random, the performance of the search operation is memory-bound.

To reduce the number of memory accesses, we utilise the n-step FM-index [6]. This is an algorithmic modification to the FM-index structure which allows the SA interval to be updated for n symbols in each step; consequently, the number of memory accesses is reduced from $2L$ to $2L/n$. The n-step FM-index is constructed by first computing n BWTs using the relationship: $\mathrm{B}_{j,i} = R[(\mathrm{SA}_i - j) \mathrm{mod}\ |R|]$, where $j = 1, \ldots, n$. The BWTs are then merged to create a single BWT with an increased alphabet size. The n-step FM-index is then generated using the same procedure described in Section 2.

The n-step FM-index search operation is shown in Algorithm 2. Note that if L is not divisible by n, the SA is first updated for the remainder symbols using precomputed values. The trade-off for the n-step FM-index is the increase in index size. The index size is calculated using Equation (5). For the Human genome, with R = 3G, Σ = 4, n = 3, and d = 128, the index size is 8.7 GB.

$$M = \frac{4 \cdot R \cdot \Sigma^n}{d} + \frac{R \cdot \mathrm{ceil}(\log_2(\Sigma^n + 1))}{8} \quad \text{Bytes.} \tag{5}$$

Algorithm 2. n-step FM-index Search Operation

Input: substring Q, n-step FM-index F with bucket width d and step size n
Procedure: merge$(s_1, \ldots, s_n)$ – merge symbols from left to right

```
1: for $i \leftarrow |Q| - 1$ to 0 step $-n$ do   ▷ update suffix array interval
2:     $s \leftarrow \mathrm{merge}(Q_{i-n+1}, \ldots, Q_i)$   ▷ merge $n$ substring symbols
3:     $low \leftarrow \Phi(F[low/d], s, low)$
4:     $high \leftarrow \Phi(F[high/d], s, high)$
5:     if $low \geq high$ then   ▷ terminate if symbol not matched
6:         end
7:     end if
8: end for
```

Index oversampling. With each update of the SA interval, the values of *low* and *high* converge. After several iterations it is often the case that *low* and *high* are sufficiently close that $F[low/d]$ and $F[high/d]$ point to the same index bucket. In this case only one memory access is required to update both *low* and *high*. If $F[low/d]$ and $F[high/d]$ always point to the same index bucket after X symbols have been aligned, then the total number of memory accesses is reduced from $2L$ to $2X + (L - X)$. The value of X is dependent on the size of the reference sequence. Tests using the Human-genome as a reference sequence indicate that the average value for X is 13, therefore for a sequence of 100 symbols, the number of memory accesses is reduced by 1.8 times.

To eliminate cases where the values of low and $high$ are sufficiently close, but $F[low/d]$ and $F[high/d]$ point to adjacent buckets, the index is oversampled by a factor of f. In this procedure, the Occ values are stored every d/f symbols, however the BWT size remains d symbols. The trade-off is that the index size increases by a factor of f, however this can be mitigated by increasing the bucket width. If $high - low < d/f$, then $F[low/d]$ and $F[high/d]$ will point to the same index bucket; consequently, only one memory access is required to update the suffix array interval for each of the remaining symbols. The SA interval update is modified according to Algorithm 3.

Algorithm 3. Oversampled n-step FM-index Search Operation

Input: substring Q, oversampled n-step FM-index F with bucket width d, step size n, and oversampling factor f

```
1: for $i \leftarrow |Q| - 1$ to 0 step -n do    ▷ update suffix array interval
2:     $s \leftarrow \mathrm{merge}(Q_{i-n+1}, \ldots, Q_i)$
3:     $low \leftarrow \Phi(F[low/d], s, low)$
4:     if $high - low \geq d/f$ then    ▷ point to different buckets
5:         $high \leftarrow \Phi(F[high/d], s, high)$
6:     else    ▷ point to same bucket
7:         $high \leftarrow \Phi(F[low/d], s, high)$
8:     end if
9: end for
```

Bi-directional backtracking. The FM-index search operation is extended with backtracking to support approximate alignment. An exhaustive search is used to detect all possible alignment hits. In this approach all possible read permutations must be tested, therefore the worst case time complexity is $O(\Sigma^k L^{k+1})$. To improve the approximate alignment performance, we prune the search space by constraining the edit positions and using a bi-directional search [12].

To support search operations in both directions (forward and backward), the FM-index is generated for both the reference genome and its reverse. In a backward search, the FM-index generated from the reference genome is used, and in a forward search the FM-index generated from its reverse is used. To prune the search space the edit positions are constrained. For example, if one edit is permitted, the edit position can either exist in: 1) the first half of the read, or 2) the second half. For case 1) the second half of the read must exact align. A backward search is used to update the SA interval for the second half of the read. The backward search is then extended to the first half of the read, however an edit is considered at each position. For case 2) the first half of the read must exact align. A forward search is used to update the SA interval for the first half of the read. Then the forward search is extended to the second half of the read, however an edit is considered at each position. The advantage of this approach is that long sections of the read can be exact aligned first, reducing the SA interval size and the search space. In general, for k permitted edit operations, the short read is split into $k + 1$ sections. The short read is then tested for k edit operations using a series of phases which aim to maximise the length of the section exact aligned first.

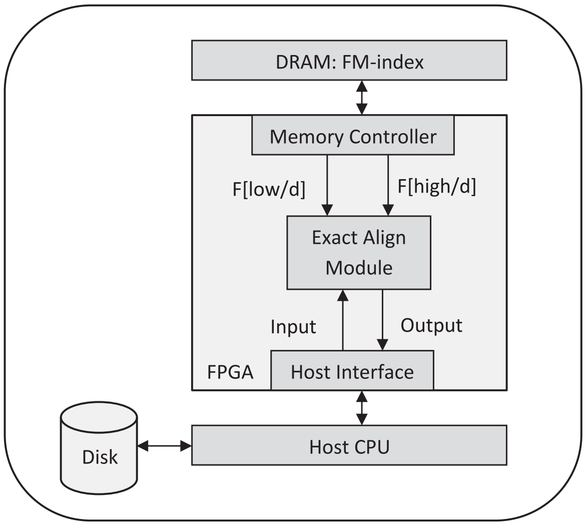


Fig. 6. Exact alignment configuration. Arrows indicate data streams between the design components.

6 Hardware Implementation

In this section we present the implementation details of our alignment design. Hardware optimisations are presented which improve the alignment performance.

6.1 Module Designs

Modules are developed for each stage in the alignment design. Each module is mapped to its own FPGA configuration, where it is replicated as many times as possible according to the resources available on the FPGA fabric. Our hardware design targets computing systems with FPGA coprocessor boards. The host CPU reads in the short read data from disk and offloads the reads to the FPGA for alignment. The results are transferred back to the host and written to disk.

FM-Index Modules. The host packages the short reads into packets composed of; a read identifier, read length, and the read symbols (encoded using 2-bits). The host sends these packets to the FPGA, which performs the FM-index search operation. After all the read symbols have been aligned, the final SA interval is transferred back to the host, where it is converted to reference genome positions. Fig. 6 illustrates a high-level overview of the exact alignment configuration.

The FM-index is stored in off-chip DRAM attached to the FPGA device. Accessing DRAM takes hundreds of cycles, which coupled with the step interdependence of the FM-index search operation results in a non-filled pipeline of operations. To improve the module performance, the processing of multiple reads is interleaved such that in each pipeline stage a different read is processed; consequently, the pipeline is completely filled, increasing the throughput. Furthermore, the DRAM memory controller is constantly processing commands, maximising the memory bandwidth utilisation. Interleaving is implemented using a circular buffer, where the buffer size is made equal to the total

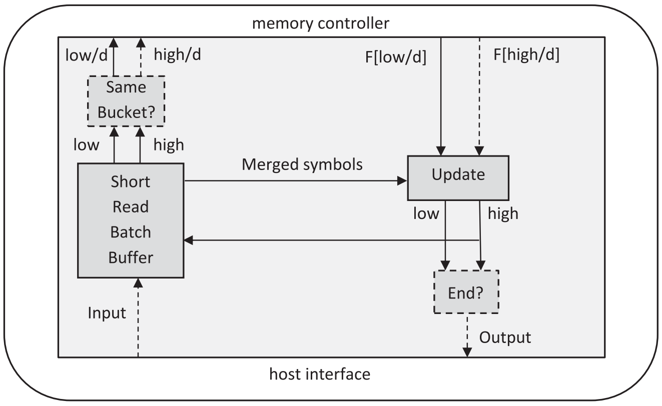


Fig. 7. Exact match module block diagram. Note dashed lines indicate logic controlling the state of the stream.

module latency. The trade-off is that additional logic and BRAM resources are required to store the batch of reads and their corresponding alignment state. The additional resources can be reduced by minimising the module latency. For example, counting the occurrences of a symbol directly from the BWT is transformed into a Hamming weight computation in order to reduce resource usage.

In each cycle, two memory commands are sent to the memory controller requesting $F[low/d]$ and $F[high/d]$. A custom memory command generator is developed so that the index oversampling optimisation can be realised in hardware. For the $F[high/d]$ memory stream, a control bit is used to disable the command when low and $high$ point to the same bucket. In this case the bucket corresponding to $F[low/d]$ is used to update both low and $high$, reducing the number of memory commands processed. Multiplexers are used to select the appropriate input and computation result based on the values of low and $high$. Fig. 7 shows a block diagram of the exact alignment module.

For the one and two mismatch alignment modules, additional control logic is used to enable bi-directional backtracking. Circular buffers are used to store the state (symbol, position, low and $high$) for each mismatch permitted. These states, along with the updated values of low and $high$ are used to control the backtracking. Both modules are designed to find all possible alignment hits for the given number of mismatches. Since the number of alignment hits for each short read is non-deterministic, the host is unable to synchronise with the FPGA based on the number of output bytes received. The alignment results are therefore stored first in off-chip DRAM, then streamed to the host after processing has finished. Both modules therefore require an additional stream to DRAM to write the alignment results.

Compare module. The compare module performs an equality test of the short read and reference genome at each position in parallel. The host transfers packets to the FPGA composed of, a read identifier, read length, the read symbols, and the reference genome segment symbols. The FPGA directly compares the read and reference symbols in each position, storing the number of mismatches, and the corresponding position and symbol for up to two mismatches. The result is transferred back to the host CPU, where the validity of the alignment position is determined based on the number of mismatches found.

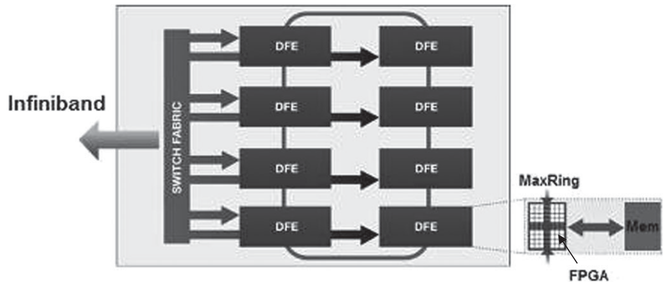


Fig. 8. Maxeler MPC-X2000 architecture [19].

7 Evaluation

In this section we evaluate the performance of our FPGA-based alignment design, and provide comparisons to the fastest software alignment tools currently available.

7.1 Experiment Setup

Our alignment design is implemented on a Maxeler MPC-X2000 system [19] with eight dataflow engines (DFEs). Each DFE comprises a single Altera Stratix V FPGA (28 nm feature size) connected to 48 GB of DRAM. The DFEs are connected to a CPU host machine via Infiniband. The Maxeler development process involves expressing the to-be-accelerated computation as a dataflow graph using the MaxJ language. The MaxCompiler then transforms the dataflow graph into low-level hardware. Fig. 8 illustrates the MPC-X2000 architecture.

The FM-index is constructed with a bucket width $d = 128$, step size $n = 3$, and a sampling factor $f = 2$. These values are chosen to maximise the step size whilst ensuring that: a) the index fits in DRAM, and b) the bucket size is optimal with respect to the DRAM burst size. Figs. 9 and 10 show how both the FM-index and bucket size vary with the bucket width and step size respectively. With the parameter values chosen, the FM-index size is 34 GB, and the bucket size is 384 Bytes, which matches the burst size of the Maxeler DFEs. The FM-index is static throughout alignment, therefore it only needs to be transferred to DRAM once for many alignment jobs to be performed. For hardware platforms with soft memory controllers, partial reconfiguration can be used to retain the data in memory when a reconfiguration is performed.

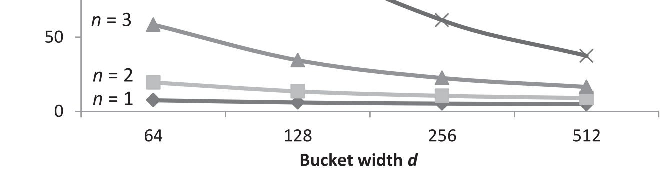


Fig. 9. FM-index size as a function of the bucket width d and step size n. Note that the sampling factor $f = 2$, and the size accounts for both the FM-index of the reference genome and its reverse.

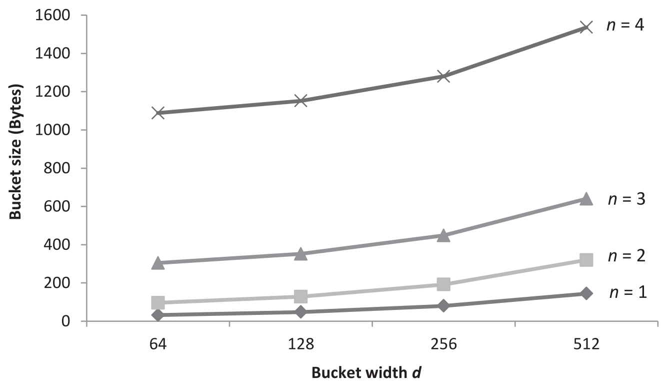


Fig. 10. FM-index bucket size as a function of the bucket width d and step size n.

The resource usage and achievable clock frequency for each hardware configuration is shown in Table 3. The FM-index-based configurations show relatively low resource usage as only a single module is instantiated on the FPGA fabric. This limitation is specific to the Maxeler platform: each DFE supports a single channel to off-chip memory; consequently, no additional performance is observed when instantiating multiple modules as the memory bandwidth is saturated. To improve the performance, the memory modules can be decoupled, allowing up to six independent memory channels per DFE. Although this modification has not been implemented yet, we use it to provide an upper bound performance estimate for the MPC-X2000. Measurements indicate that the achievable memory bandwidth per DFE is 4.2 GB/s, which is 11 percent of the theoretical peak. For hardware platforms with random access speeds close to that of sequential access, we expect the performance of our design to substantially increase.

The performance of our hardware-accelerated alignment design is compared to the software tools listed in Table 4. The CPU-based tools are run on a 1 U system with dual Intel Xeon E5-2640 s (22 nm feature size) and 64 GB of DDR3-1333. Soap3-dp is run on an NVIDIA Tesla C2070 GPU. For this evaluation both simulated and real experimental sequencing data are aligned. The real experimental data is taken from the ERP001652 [7] study which comprises 10M reads of 100 bases. The runtime options for each tool are set to report the best alignment hits with up to two mismatches. Given the size of the data sets evaluated, we do not include the FPGA reconfiguration time (~4s) and the FM-index transfer time (~50s) in our runtime measurements, as it would introduce a large negative bias. Furthermore, in all runtime measurements, including the software tools, disk IO time is omitted.

TABLE 3
Module Resource Usage on a Altera Stratix V FPGA

Alignment Configuration	Clock (MHz)	LUT (k)	FF (k)	BRAM (k)
Exact	200	68 (26%)	129 (25%)	0.6 (22%)
Seed	200	74 (28%)	142 (26%)	0.9 (33%)
Compare (×8)	200	86 (33%)	219 (40%)	1.2 (46%)
One mismatch	150	74 (28%)	136 (25%)	0.9 (33%)
Two mismatches	150	75 (28%)	136 (25%)	0.9 (33%)

Note that the memory controller is run at 800 MHz.

TABLE 4
Alignment Tools Used in Evaluation

Tool	Version	Options
BWA aln+samse	0.7.12	-n 2 -p 16
Bowtie2	2.2.6	–very-fast -p 16
Soap2	2.21	-v 2 -p 16
Soap3-dp	2.3.177	-s 2

Note that 16 threads were used for all the CPU-based tools tested, and up to two edits were permitted where available.

7.2 Results

The performance of our alignment design is evaluated for exact, one mismatch, and two mismatches alignment. In each test 10M short reads of 100 bases are directly sampled from Hg19. Mismatches are then inserted at random positions in the reads according to the number being tested. Fig. 11 shows that for exact match, one mismatch, and two mismatches alignment, our design is faster than all the software tools tested. The largest performance improvement is for exact alignment which is 112 times faster than Soap2, and 42 times faster than Soap3-dp. For one and two mismatches alignment our design is 31 and 23 times faster than Bowtie2 respectively, and 18 and 15 times faster than Soap3-dp respectively.

The n-step FM-index is the largest contributing factor to the hardware performance, providing a three times improvement over the base algorithm. The performance improvement from index oversampling is difficult to quantify as it depends on the reference sequence length. When the Human genome is used as a reference sequence the performance improves by 1.8 times. For the seed-and-compare optimisation a position threshold of 5 is used to ensure that software tasks, such as generating the hardware input, do not become a bottleneck. Over 70 percent of the short reads are aligned in the seed-and-compare stage, reducing the amount of work done by the comparatively slower one and two mismatches alignment modules. The largest performance improvement is for two mismatches alignment, which is 1.6 times faster when the seed-and-compare stage is included.

The performance of our alignment design is evaluated on real experimental data from the ERP001652 study. The sequenced data comprises 10M reads of 100 bases, where 73 percent of the short reads exactly align, 13 percent align with one mismatch, and 3 percent align with two mismatches. Table 5 shows that our design is faster than all the software tools tested. In particular, our design is 28 times faster than Bowtie2, and nine times faster than Soap3-dp.

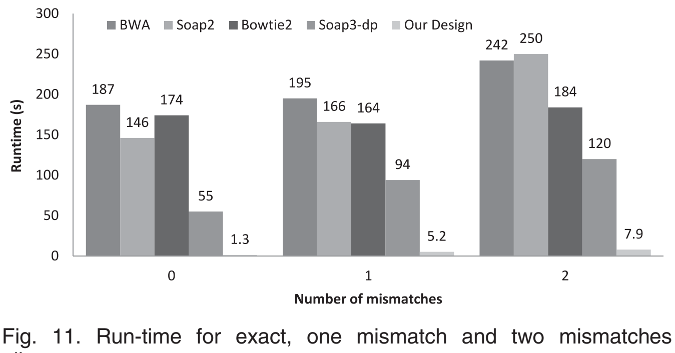


Fig. 11. Run-time for exact, one mismatch and two mismatches alignment.

TABLE 5
ERP001652 Data Set Alignment Performance

Tool	Accuracy	Runtime (s)	Speed-up
BWA	90.1	273.4	1.0
Soap2	89.4	188.9	1.4
Bowtie2	96.3	175.0	1.6
Soap3-dp	89.4	56.3	4.9
Our design	89.4	6.3	43.4
Upper bound estimate	89.4	1.1	248.5

TABLE 6
Device Energy Consumption

Tool	Device Power (W)	Energy Consumption (kJ)
BWA	180	24.6
Soap2	180	17.0
Bowtie2	180	15.7
Soap3-dp	238	13.4
Our design	86 (average)	0.54

The bottleneck of our design is the seed-and-compare stage which accounts for 40 percent of the runtime. This bottleneck can be reduced by making the position threshold smaller, however this increases the runtime as more work is done by the one and two mismatches alignment modules. For the upper bound performance estimate it is assumed that there are enough resources on the FPGA fabric to support six independent memory channels. If each memory channel can sustain the same bandwidth observed in our current design, a six times performance improvement would be expected. Note that issues such as meeting timing and resource limitations could reduce the achievable improvement.

The accuracy of our design for up to two mismatches is measured to be identical to that of Soap2 and Soap3-dp. These tools have the same alignment strategy as our design: the short reads are first tested for exact alignment, followed by one mismatch alignment, and then two mismatches alignment. The greater alignment accuracy observed for BWA and Bowtie2 can be attributed to gaps being natively supported, and the tool not supporting precise control over the number of mismatches permitted respectively. Since the reconfigurable architecture is completely modular, a dynamic programming module can be added to support gapped alignment.

Given that the runtime scales linearly with the read count, the alignment time can be linearly extrapolated to that for a typically sized workload of 300 M reads. Our design could align this sized workload in just over 3 minutes, which would significantly reduce the time taken to analyse sequencing data. One concern is that disk IO time would become the bottleneck of the alignment stage, however our design can be easily adapted to read compressed sequencing data formats such as gzip and reference-based compression formats.

Table 6 shows the energy consumption for our design and the software tools. The CPU and GPU device power values are taken from the vendors' product information, whilst the FPGA device power is measured from the MaxOS operating system. The values in Table 6 indicate that our design consumes over an order of magnitude less energy than the software tools tested. This can be attributed to the low operational clock frequency, coupled with the comparatively short alignment time. With relatively small energy consumption, form factor and cooling requirements, our design is a promising candidate for integration in data centres and clinical settings.

8 Conclusion

In this work we show that a runtime reconfigurable architecture can be used to accelerate short read alignment. This architecture exploits the reconfigurability of FPGAs to enable the development of fast yet flexible alignment designs. We apply this architecture to develop a short read alignment design which supports exact and approximate alignment with up to two mismatches. Our design is based on the FM-index search operation, with optimisations to improve the alignment performance. In particular, the n-step FM-index, index oversampling, bi-directional backtracking and a seed-and-compare stage are included. Measurements show that when aligning 10M reads from the ERP001652 study, our design is 28 times faster than the CPU-based Bowtie2 and nine times faster than the GPU-based Soap3-dp. Future work involves applying our work to real sequence analysis pipelines, and automating the implementation of such pipelines from a high-level description. For enquiries regarding the availability of our design, please contact the first author.

Acknowledgments

The authors would like to thank Dennis Lo, Rossa Chiu, and Alex Ross for their advice and encouragement. This work was supported in part by Maxeler University Programme, Altera, UK EPSRC Project EP/I012036/1, the European Union Horizon 2020 Research and Innovation Programme under grant agreement number 671653, and the HiPEAC NoE.

References

[1] G. R. Abecasis, D. Altshuler, and A. Auton, "A map of human genome variation from population-scale sequencing," *Nature*, vol. 467, no. 7319, pp. 1061–1073, Oct. 2010.
[2] J. Arram, et al., "Hardware acceleration of genetic sequence alignment," in *Proc. Appl. Reconfigurable Comput.*, 2013, pp. 13–24.
[3] J. Arram, et al., "Ramethy: Reconfigurable acceleration of bisulfite sequence alignment," in *Proc. Field-Programmable Gate Arrays*, 2015, pp. 250–259.
[4] (2016). BGI-Biocloud. [Online]. Available: http://biocloud.cngb.org/.
[5] M. Burrows and D. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Palo Alto, CA, USA, Tech. Rep. 124, 1994.
[6] A. Chacón, et al., "n-step FM-index for faster pattern matching," *Procedia Comput. Sci.*, vol. 18, pp. 70–79, 2013.
[7] (2016). ENA short reads. [Online]. Available: http://www.ebi.ac.uk/ena/data/view/ERP001652.
[8] E. Fernandez, et al., "String matching in hardware using the FM-index," in *Proc. Field-Programmable Custom Comput. Machines*, May 2011, pp. 218–225.
[9] E. Fernandez, et al. "Multithreaded FPGA acceleration of DNA sequence mapping," in *Proc. High Perform. Extreme Comput.*, Sep. 2012, pp. 1–6.
[10] P. Ferragina and G. Manzini, "An experimental study of an opportunistic index," in *Proc. Annu. ACM-SIAM Symp. Discrete Algorithm*, 2001, pp. 269–278.

[11] S. D. Kahn, "On the future of genomic data," *Sci. (Washington)*, vol. 331, no. 6018, pp. 728–729, 2011.
[12] T. W. Lam, et al., "High throughput short read alignment via bi-directional BWT," in *Proc. IEEE Int. Conf. Bioinformatics Biomed.*, Nov. 2009, pp. 31–36.
[13] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nat. Methods*, vol. 9, no. 4, pp. 357–359, Apr. 2012.
[14] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows-wheeler transform," *Bioinf.*, vol. 25, no. 14, pp. 1754–1760, Jul. 2009.
[15] R. Li, et al., "Soap2: An improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 25, no. 15, pp. 1966–1967, 2009.
[16] Y. Liu, et al., "CUSHAW: A CUDA compatible short read aligner to large genomes based on the burrows-wheeler transform," *Bioinf.*, vol. 28, no. 14, pp. 1830–1837, Jul. 2012.
[17] R. Luo, et al., "SOAP3-dp: Fast, accurate and sensitive GPU-based short read aligner," *PLoS ONE*, vol. 8, no. 5, p. e65632, 2013.
[18] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," in *Proc. 1st Annu. ACM-SIAM Symp. Discr. Algorithms*, 1990, pp. 319–327.
[19] (2016). Maxeler Technologies. [Online]. Available: http://www.maxeler.com/products/mpc-xseries/.
[20] C. Olson, et al., "Hardware acceleration of short read mapping," in *Proc. Field-Programmable Custom Comput. Mach.*, Apr. 2012, pp. 161–168.
[21] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Mol. Biol.*, vol. 147, no. 1, pp. 195–197, Mar. 1981.

James Arram received the BSc degree in physics from Bristol University, Bristol, United Kingdom, and the MSc degree in computing science from Imperial College London, London, United Kingdom. He is currently working toward the PhD degree in the Department of Computing, Imperial College London. His research interests include FPGA-based acceleration of genomic data analysis, and compression.

Thomas Kaplan received the BEng degree in computing from Imperial College London, London, United kingdom. He is currently working at Fidelity International, Pembroke, Bermuda.

Wayne Luk is a professor of computer engineering at Imperial College London. His current research interests include theory and practice of customizing hardware and software for specific application domains such as genomics and climate modelling, and high-level compilation techniques and tools for high-performance computers and embedded systems. He is a fellow of the Royal Academy of Engineering, the IEEE, and the BCS. He received the Research Excellence Award from Imperial College London, and 11 awards for his publications from various international conferences.

Peiyong Jiang received the PhD degree from the Department of Chemical Pathology, The Chinese University of Hong Kong, Hong Kong. He is currently an assistant professor and a bioinformatician in the Department of Chemical Pathology, The Chinese University of Hong Kong.