Imperial College London
Department of Computing

# FPGA Acceleration of DNA Sequence Alignment: Design Analysis and Optimization

Ho-Cheung Ng

July 2021

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

**Abstract**

Existing FPGA accelerators for short read mapping often fail to utilize the complete biological information in sequencing data for simple hardware design, leading to missed or incorrect alignment. In this work, we propose a runtime reconfigurable alignment pipeline that considers all information in sequencing data for the biologically accurate acceleration of short read mapping. We focus our efforts on accelerating two string matching techniques: FM-index and the Smith-Waterman algorithm with the affine-gap model which are commonly used in short read mapping. We further optimize the FPGA hardware using a design analyzer and merger to improve alignment performance. The contributions of this work are as follows.

1. We accelerate the exact-match and mismatch alignment by leveraging the FM-index technique. We optimize memory access by compressing the data structure and interleaving the access with multiple short reads. The FM-index hardware also considers complete information in the read data to maximize accuracy.

2. We propose a seed-and-extend model to accelerate alignment with indels. The FM-index hardware is extended to support the seeding stage while a Smith-Waterman implementation with the affine-gap model is developed on FPGA for the extension stage. This model can improve the efficiency of indel alignment with comparable accuracy versus state-of-the-art software.

3. We present an approach for merging multiple FPGA designs into a single hardware design, so that multiple place-and-route tasks can be replaced by a single task to speed up functional evaluation of designs. We first experiment with this approach to demonstrate its feasibility for different designs. Then we apply this approach to optimize one of the proposed FPGA aligners for better alignment performance.

# Declaration of Originality

I hereby declare that I am the sole author of this thesis and that the material within has not been submitted for a degree in any other institution. The materials and information used and derived from other published sources have been properly cited and acknowledged.

# Copyright Declaration

# Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor Professor Wayne Luk. It was his wisdom and knowledge that guided me to the completion of this thesis. I am grateful to have his continuous support and encouragement during my Ph.D. studies. I would like to thank my M.Phil. supervisor Dr. Hayden So who introduced me to reconfigurable computing research. Special thanks to Izaak Coleman and Shuanglong Liu for all the invaluable discussions and their technical support on FPGA design analysis and short read alignment research. Finally, and most importantly, I must thank my family for their unconditional love and endless care throughout the years of my studies.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

**B** BWT within the bucket of F

**bp** Base Pair(s)

**bps$^{-1}$** Base Pairs Aligned per Second

**BRAM** Block Memory on FPGA

**BWT** Burrows-Wheeler Transform

**BWT(R)** Burrows-Wheeler Transform of the Reference Genome R

**d** Sampling Distance

**DAM** Automated Design Analysis and Merging

**EM** Exact-match Strategy

**F** Restructured FM-index

**GCUPs** Billion Entry/cell Updates per Second

**ICFPT** International Conference on Field Programmable Technology

**ISEs** Instruction Set Extensions

**LUT** Look-Up Table on FPGA

**MCS** Maximum Common Subgraph

**MSA** Multiple Sequence Alignment

**NSG** Next-generation Sequencing

**OM** One-mismatch Strategy

**P** Search Pattern

**PE** Processing Element

**PSA** Pairwise Sequence Alignment

**qs** Quality Sum

**R** Reference Genome

$\mathbf{R}_{human}$ Human Reference Genome

**SA** Suffix Array

**SE** Seed-and-extend strategy

**SNPs** Single Nucleotide Polymorphisms

**V** Scoring Matrix of the Smith-Waterman Algorithm

# Chapter 1

# Introduction

In recent years, there has been a significant explosion in the quantity of genomic data. This is mainly due to technological advancement and the declining cost of next-generation sequencing (NGS) technology. The NGS machines are now capable of generating millions or even billions of short DNA fragments from the sampled cells within hours [3]. These fragments, *i.e.* reads, are produced by segmenting DNA strands in the sampled cells randomly. As a consequence of this action, the orientation and position information of the reads with respect to the original, long DNA is lost. Therefore, a critical prior step of lots of downstream analysis of genomics data is short read alignment (mapping), where millions or billions of short DNA reads generated by an NGS machine are mapped to a reference genome [4]. Figure 1.1 displays an example of aligning reads to a short reference.

Figure 1.2 shows a snippet of an example read file, which is normally stored in FASTQ format using ASCII encoding. It uses four lines per sequence. The first line is the sequence identifier while the second line is the DNA short read. The third line begins with an '+' character and this line only stores optional information. The fourth line carries important information that encodes the quality metric for the short read. Each quality value is associated with a nucleotide in the same position of line 2, and each value has a range between

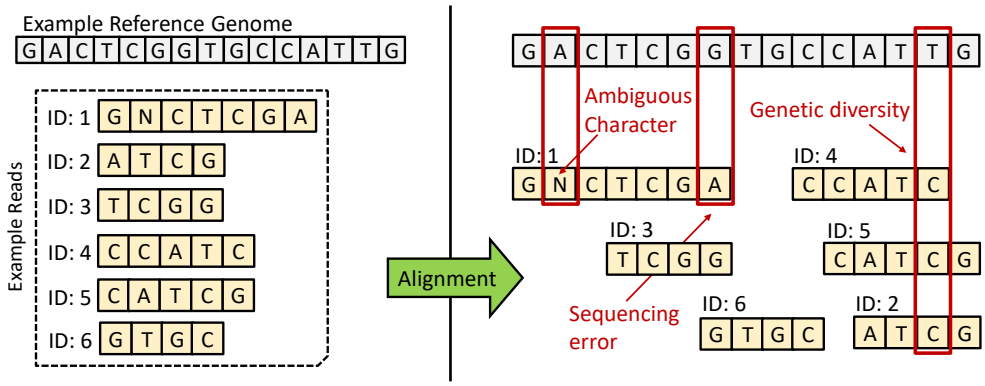



Figure 1.1: Example of aligning reads to a reference.


Figure 1.2: Snippet of an example FASTQ file.

33 ('!' in ASCII) denoting the lowest quality and 75 ('K') denoting the highest quality.

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

Figure 1.3 shows the increase in sequenced data produced by the Illumina NGS platforms from 2006 to 2017. Given that the throughput of NGS machines is far exceeding the growth of transistor counts based on Moore's Law, the time required for existing software aligners to map NGS data is becoming

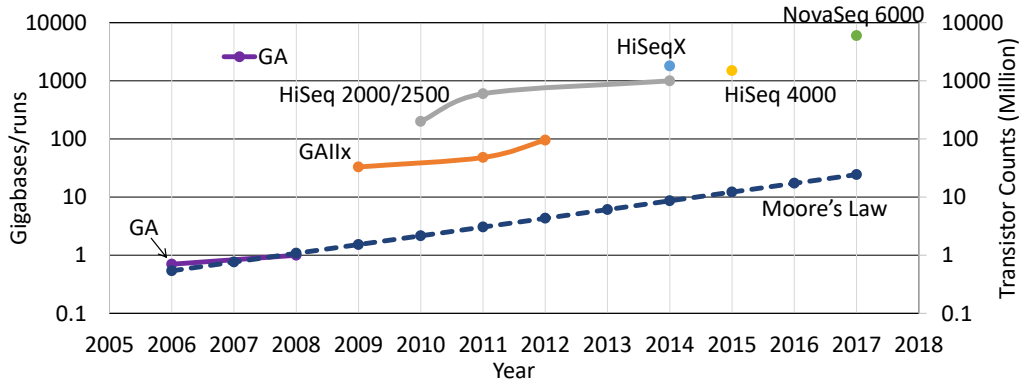



Figure 1.3: The sequencing throughput of Illumina NGS platforms. The data in this graph is acquired from [1, 2].

prohibitive [3]. This hinders the medical applications of NGS, such as prenatal diagnostics and monitoring, where individuals' DNA and RNA should be analyzed quickly at a low cost [5, 6]. Therefore, their acceleration would bridge the gap between alignment research and practice, allowing these diagnosis techniques to become part of routine clinical procedures [7].

FPGA technology is a promising candidate to accelerate short read mapping [8]. Its highly-parallel bit-oriented architecture has been leveraged to accelerate different mapping algorithms. As the sequenced alphabet produced by the NGS machines is abstracted into {A, C, G, T, N} which can be represented in 3 bits, mapping the DNA nucleotides to the reference genome is inherently a bitwise operation. Different alignment algorithms, for example, the FM-index technique [9] or Smith-Waterman algorithm [10], have been implemented and accelerated on FPGA, as they are commonly used in state-of-the-art software such as Bowtie [4], Bowtie2 [11] and BWA-MEM [12]. Finally, based on our initial experiment with exact match alignment between the reads (101 base pairs in length) and the human reference genome, we discover that a Virtex Ultrascale+ VU9P FPGA starts outperforming Bowtie on Intel Silver 4110 CPU running with 16 threads, should there be 7 million reads or more to process.

## 1.1 Challenges and Contributions

Despite the success, FPGA-accelerated short read mapping is rarely adopted in genomics research and medical applications due to the following reasons:

1. Most accelerators fail to utilize the complete information available in NGS data. To simplify the hardware design, they utilize only the Watson-Crick alphabets {A, C, G, T}. The quality metric information and ambiguous characters (N characters) that are commonly present in NGS data are usually discarded or neglected. This can result in incorrect alignment and generate biologically invalid results.

2. Many FPGA researchers select and accelerate alignment algorithms that are in favor of hardware implementations. As a result, the alignment workflow of these accelerators can be inconsistent with state-of-the-art software. Without a comprehensive analysis of the corresponding alignment accuracy, this diminishes the confidence of realistic applications of FPGA aligners. For example, the accelerator in [13] performs exact string matching based on the FM-index, followed by approximate string matching based on the seed-and-extend strategy with the linear Smith-Waterman algorithm. However, this workflow is inconsistent with the alignment model in software such as Bowtie or Bowtie2, which in turn limits the biological validity and reproducibility of the alignments it outputs.

3. Most FPGA aligners are platform-dependent where the design optimizations often target a single FPGA device. As a result, the design cannot be optimized easily across platforms. Because of the cost and privacy concerns, sometimes local clusters are preferable for alignment [14]. The lack of design portability reduces the attraction of FPGA technology to genomics scientists, allowing its expenses and requirement of re-optimization to outweigh its potential benefits. For example, Arram *et*

*al.* [6] propose a suffix-trie-based accelerator that is 18.1 × faster than software. However, the design targets Maxeler MaxWorkstation and the corresponding optimizations are unlikely to generalize across platforms.

To increase the utility of FPGA aligners in genomics research and medical applications, we present novel designs and architecture that consider complete biological information, including quality metic and ambiguous characters to accelerate short read alignment. With the runtime reconfigurability of FPGA, we study different alignment workflows by proposing and implementing different arrangements of alignment strategies. For each arrangement, we study the corresponding alignment speed and accuracy when compared to Bowtie and Bowite2. We research the alignment accelerator according to Bowtie and Bowtie2 because they are extensively used in alignment research and practice with more than 19,470 and 29,592 citations respectively. To aid design and performance portability, we propose an automated design analyzer and merger to facilitate the optimization of an aligner implementation based on the target FPGA platform. Specifically, we use the analyzer and merger to determine the optimal FM-index implementation. Multiple similar designs that can concurrently process a different number of reads are developed, so as to conclude the one that can mask the memory access latency optimally.

The aim of this work is to address the above challenges and ultimately promote the use of FPGAs to alleviate the data processing bottlenecks in DNA sequencing alignment without compromising accuracy. The contributions of this work are as follows:

1. We propose and develop accelerator designs that implement exact-match, one-mismatch, two-mismatch and three-mismatch alignment strategies. We leverage FM-index and backtracking FM-index to achieve biologically accurate alignment, by considering complete biological information including quality metric and ambiguous characters. We also maximize the performance by compressing the nucleotide representations and data

structure, interleaving the memory access with multiple short reads, and applying bi-directional FM-index (Chapter 3).

2. We implement the seed-and-extend strategy on FPGA to perform indel alignment that includes matches, mismatches, deletions and insertions. The exact-match accelerator is extended to perform seeding while a Smith-Waterman implementation with the affine gap model is developed for the extension. We also arrange one or multiple aforementioned alignment strategies into a consecutive, sequential pipeline. Reads aligned by a strategy are reported immediately while unaligned reads are directed to subsequent strategies. For each alignment pipeline, we investigate the relationship between speed-up and accuracy to provide guidance for genomics scientists (Chapter 4).

3. We propose a multi-configuration alignment pipeline by exploiting the runtime reconfigurability of FPGA. Distinct hardware implementations are loaded in turn, so as to construct alignment strategies in the required order. This architecture can promote portability as users can have better control over alignment parameters. Implementations can be re-arranged, removed, or added straightforwardly to accommodate different sequenced data quality and experiments being performed (Chapter 3 and Chapter 4).

4. We introduce an approach for merging multiple FPGA designs into a single hardware design, so that multiple place-and-route tasks can be replaced by a single task to speed up functional evaluation of designs, especially during the optimization stage of the applications, such as FPGA aligners. This approach has three key elements. First, a novel approximate maximum common subgraph detection algorithm for dataflow graphs with linear time complexity to maximize sharing of resources in the merged design. Second, a prototype tool implementing this common subgraph detection algorithm for dataflow graphs derived from Verilog

designs. This tool would also generate the appropriate control circuits to enable the selection of the original designs at runtime. Third, a comprehensive analysis of compilation time versus degree of similarity to identify the optimized user parameters for the proposed approach (Chapter 5).

## 1.2 Thesis Organization

This work is organized into six chapters. In this chapter, the motivations, research challenges and contributions are presented.

In Chapter 2, we present background information on the most commonly used alignment algorithms: the FM-index technique and the Smith-Waterman algorithm with the affine gap model. We also provide a comprehensive literature review that studies and analyzes previous FPGA accelerators for DNA sequence alignment.

In Chapter 3, we describe the FM-index accelerators that implement exact-match and mismatch alignment strategies. In addition, we present a runtime reconfigurable architecture that aligns reads with different edit-distance-based strategies. A few novel and general methods which enable customization of the FPGA aligners for the platform beyond this work are also given. We summarize this chapter by demonstrating the speed-up and accuracy, compared to previous work and Bowtie.

In Chapter 4, we elaborate on the runtime reconfigurable accelerator that implements the seed-and-extend strategy, which is based on an FM-index and a Smith-Waterman implementation with the affine gap model. We then extend the architecture by incorporating one or more of the implemented strategies in Chapter 3, forming different arrangements of alignment strategies. For each arrangement, a detailed investigation of the alignment performance and accuracy is presented.

In Chapter 5, we propose an automatic merger that combines multiple versions of a design project into a single hardware implementation. The proposed

merger can identify common computational kernels between versions, perform the necessary merging and generate a final hardware design in linear time. We also benchmark the merger by using several designs from the VTR Benchmarks [15, 16], and provide a case study on Binomial Filters and one of the proposed FPGA aligners. We conclude this chapter with some previous work on FPGA design analysis and merging.

Finally, in Chapter 6 we conclude this thesis with the current state of our work and thoughts about future work.

## 1.3 Selected Publications

The work in this thesis has been published in five conferences and journals. We provide a list of selected publications below that are mostly based on the material in this work:

1. H.-C. Ng, I. Coleman, S. Liu and W. Luk, "Reconfigurable Acceleration of Short Read Mapping with Biological Consideration," in *2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2021, pp. 229-239, doi: https://doi.org/10.1145/3431920.3439280.

2. H.-C. Ng, S. Liu, I. Coleman, R. S.W. Chu, M.-C. Yue and W. Luk, "Acceleration of Short Read Alignment with Runtime Reconfiguration," in *2020 IEEE International Conference on Field-Programmable Technology (FPT)*, 2020, pp. 256-262, doi: 10.1109/ICFPT51103.2020.00044, © 2020 IEEE.

3. I. Coleman, G. Corleone, J. Arram, H.-C. Ng, L. Magnani and W. Luk, "GeDi: applying suffix arrays to increase the repertoire of detectable SNVs in tumour genomes," in *BMC Bioinformatics*, vol. 21, no. 45, pp. 1-12, 2020, doi: https://doi.org/10.1186/s12859-020-3367-3.

4. H.-C. Ng, S. Liu and W. Luk, "ADAM: Automated Design Analysis and Merging for Speeding up FPGA Development," in *2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018, pp. 189-198, doi: https://doi.org/10.1145/3174243.3174247.

5. H.-C. Ng, S. Liu and W. Luk, "Reconfigurable Acceleration of Genetic Sequence Alignment: A Survey of Two Decades of Efforts," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1-8, doi: 10.23919/FPL.2017.8056838, © 2017 IEEE.

# Chapter 2

# Background and Related Work

This chapter presents the background information that forms the foundation of this work. Section 2.1.1 elaborates on the most commonly used indexed string alignment techniques: Burrows-Wheeler transform (BWT) [17] and FM-index. Then Section 2.1.2 explains the Smith-Waterman algorithm and the affine gap model. The seed-and-extend alignment strategy, which is broadly used in practical indel alignment, is explained in the sub-section followed. Finally, Section 2.2 provides a comprehensive discussion on previous FPGA accelerators for DNA sequence alignment.

## 2.1 Background

### 2.1.1 Indexed String Alignment

In software aligners such as Bowtie, Bowtie2 and BWA-MEM which runs on commodity CPU, indexing is a commonly applied method to quickly locate the occurrence of a read in a long reference sequence. Indexing techniques, such as suffix array [18], BWT and FM-index, require preprocessing so as to substantially reduce the search space and provide fast string alignment. Since the reference genome changes infrequently, the cost of building the index can be amortized with an abundant number of searches.

### Suffix Array and BWT

The formation of the FM-index for a reference genome $R$ combines the suffix array SA and BWT of $R$. To generate the suffix array, $R$ is first terminated with a unique character: '$', which is lexicographically the smallest value. Then, all the rotations of the text are obtained and sorted correspondingly, forming a sorted rotation list. The suffix array can be obtained by considering the characters before '$' in each entry of the rotation list.

The BWT of R: $BWT(R)$, can be formed by extracting and concatenating the last characters of all the entries on the sorted rotation list. Table 2.1 demonstrates the derivation of BWT with an example reference genome $R =$ ACTAG. The string preceding the '$' sign in the sorted rotation list forms the suffix array, which indicates the position of each possible suffix in the original string.

Table 2.1: Example of deriving the suffix array and BWT of $R$.

| $R =$ ACTAG$ | | |
|---|---|---|
| Index $\iota$ | SA | Sorted Rotation List |
| 0 | 5 | $ACTAG |
| 1 | 0 | **ACTAG**$ |
| 2 | 3 | **AG**$ACT |
| 3 | 1 | **CTAG**$A |
| 4 | 4 | **G**$ACTA |
| 5 | 2 | **TAG**$AC |
| $BWT(R) =$ G$TAAC | | |

### FM-index

To generate the FM-index of $R$, the suffix array, $BWT(R)$ and $R$ are used to generate two functions: $i(x)$ and $c(\iota, x)$ functions. For each element $x$ of the alphabet of $R$, $i(x)$ is the number of characters in $R$ that is lexicographically smaller than $x$. For each index $\iota$ in $BWT(R)$ and for each character $x$ in

the alphabet, $c(\iota, x)$ stores the number of occurrences of $x$ in $BWT(R)$ in the range $[0, \iota - 1]$. Table 2.2 illustrates the $i(x)$ and $c(\iota, x)$ tables for the reference genome $R$.

Table 2.2: $c(\iota, x)$ and $i(x)$ functions for $R$.

$c(\iota, x)$

| Index $\iota$ | A | C | G | T |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 1 | 1 |
| 4 | 1 | 0 | 1 | 1 |
| 5 | 2 | 0 | 1 | 1 |
| 6 | 2 | 1 | 1 | 1 |

$i(x)$ $\{1, 3, 4, 5\}$

Essentially, the FM-index is a pattern searching technique that operates on the $i(x)$ and $c(\iota, x)$ functions recursively. Two pointers *top* and *bottom* are defined to perform the search. *top* refers to the index of an SA element where a specific pattern is first located, and *bottom* is the location in SA where the pattern is lastly found. If *bottom* points to an index that is less than or equal to the index pointed by the *top*, the pattern does not appear in the reference genome.

To search for a specific pattern $P$ in $R$ using its FM-index, we need to process one character at a time, starting from the last character of $P$. The *top* and *bottom* are first initialized with the first and last indices of the $c(\iota, x)$ function respectively. Then both pointers are updated according to the following equations:

$$\begin{aligned} top_{new} &= c(top_{current}, x) + i(x) \\ bottom_{new} &= c(bottom_{current}, x) + i(x) \end{aligned} \tag{2.1}$$

The final results of *top* and *bottom* are the range of indices in SA that contains $P$ as the prefix, which can be subsequently converted into the coordinates in $R$. In Algorithm 1, we provide pseudocode that details the procedure of searching $P$ in the reference $R$. Note that the time complexity of locating a pattern in the reference genome is linear with respect to $|P|$ instead of $|R|$.

```
Data: Pattern P, c(), i(), SA of Reference R
Result: The starting locations Loc of P in R
l = |P| − 1
(top, bottom) = (0, c(|R|, P[l]))
for j = l to 0 do
    top = c(top, P[j]) + i(P[j])
    bottom = c(bottom, P[j]) + i(P[j])
end
if top < bottom then
    for j = 0 to top − bottom do
        Loc[j] = SA[top + j]
    end
end
```

**Algorithm 1:** Exact substring matching using FM-index.

Finally, FM-index can be extended to support mismatch alignment with backtracking [4]. To start, a stack is maintained to store the current search state when a mismatch happens. A different character is then attempted to align with the reference genome. The state is restored when the number of mismatches exceeds the permitted value. Another untested character is used and attempted to perform alignment again.

### 2.1.2 The Smith-Waterman Algorithm

The Smith-Waterman algorithm is a dynamic programming technique to perform alignment with mismatches, insertions and deletions. It uses a scoring matrix $V$ to reveal the optimal local alignment between the pattern $P$ and the reference genome $R$, where $|P| = m$ and $|R| = n$. Every entry in $V$ is

calculated recursively according to the following equation:

$$V(i,j) = \max \begin{cases} 0 & \\ V(i-1,j-1) + \sigma(P[i],R[j]) & Match/Mismatch \\ V(i-1,j) + \sigma(P[i],\_) & Deletion \\ V(i,j-1) + \sigma(\_,R[j]) & Insertion \end{cases} \tag{2.2}$$

$$\text{for} \quad 1 \leq i \leq m, 1 \leq j \leq n$$

$$\text{Base case} \begin{cases} V(i,0) = 0 & 0 \leq i \leq n \\ V(0,j) = 0 & 0 \leq j \leq m \end{cases} \tag{2.3}$$

The function $\sigma(x,y)$ determines the relative weighting of match, mismatch, deletion and insertion between characters $x$ and $y$. The weighting can be adjusted according to different alignment requirements. For example, the insertion and deletion penalties can be set to a higher value than the substitution penalty if the presence of redundant characters is less acceptable than the character difference.

Table 2.3: Example of calculating the matrix $V$ for the pattern $P$ = AT and the reference $R$ = CTCATGG.

| | - | C | T | C | A | T | G | G |
|---|---|---|---|---|---|---|---|---|
| - | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 |
| T | 0 | 0 | 2 | 1 | 1 | 4 | 3 | 2 |

Match: $\sigma(x,x) = +2$, Mismatch: $\sigma(x,y) = -1$
Deletion: $\sigma(x,\_) = -1$, Insertion: $\sigma(\_,x) = -1$

Table 2.3 illustrates the calculation of the matrix $V$ for the alignment of read $P$ = AT to a reference $R$ = CTCATGG. The optimal alignment can be obtained by completing the matrix $V$. The highest score indicates how a

pattern can be best aligned to another sequence within the allowed diversity. In Table 2.3, the highest score is 4 which indicates that pattern $P$ can be exactly mapped to sequence $R$. By backtracking from the highest score to the entry in which the score becomes zero, the optimal alignment can be reconstructed as a string representation.

### The Affine Gap Model

The Smith-Waterman algorithm with the affine gap model is mostly used in state-of-the-art software such as Bowtie2. This model provides a more realistic computation where a genetic mutation generally causes the insertion/deletion of a large block. Compared to the above linear model, this model can account for gaps or inserts as a single event instead of individual ones. The calculation of $V$ is now changed to the following:

$$V(i,j) = \max \begin{cases} 0 & \\ V(i-1,j-1) + \sigma(P[i],R[j]) & Match/Mismatch \\ s_{i,j,\rightarrow} & Deletion \\ s_{i,j,\downarrow} & Insertion \end{cases} \tag{2.4}$$

$$\text{for} \quad 1 \leq i \leq m, 1 \leq j \leq n$$

The function $\sigma(x,y)$ determines the relative weighting of match or mismatch between characters $x$ and $y$. Equation (2.5) explains the affine gap functions $s_{i,j,\rightarrow}$ and $s_{i,j,\downarrow}$. The $\rightarrow$ and $\downarrow$ denote a character deletion and a gap insertion in $P$ respectively.

$$s_{i,j,\rightarrow} = \max \begin{cases} V(i-1,j) - \alpha \\ s_{i-1,j,\rightarrow} - \beta \end{cases} \quad s_{i,j,\downarrow} = \max \begin{cases} V(i,j-1) - \alpha \\ s_{i,j-1,\rightarrow} - \beta \end{cases} \tag{2.5}$$

$\alpha$ is the cost for an opening-gap penalty while $\beta$ is the cost for a continuous-gap penalty, with $\beta < \alpha$. Basically, the penalty for initiating the gap is more expensive than the gap extension. Compared to the linear Smith-Waterman algorithm with a standardized $\sigma$ function for all matches, mismatches, insertions and deletions, the affine gap model requires two additional scoring matrics to buffer $s_{i,j,\rightarrow}$ and $s_{i,j,\downarrow}$ which increases the memory requirements.

### 2.1.3 The Seed-and-Extend Strategy

Despite the Smith-Waterman algorithm providing an optimal mapping for alignment with matches, mismatches, insertions and deletions, its time complexity is $O(mn)$ which is prohibitively expensive given the reference genome is three billion base pairs ($bp$) in length. Therefore, a general scheme to achieve indel alignment in Bowtie2 or BWA-MEM is based on the seed-and-extend strategy.

To start, the short read is first partitioned into fixed-length subsequence, *i.e.* seeds. An indexing algorithm such as the FM-index is used to quickly identify all possible matching locations, by exactly aligning seeds to the reference. Finally, the short read is then mapped to the reference using the Smith-Waterman algorithm at each matching location. This model substantially improves the alignment efficiency with a negligible accuracy loss.

## 2.2 Related Work

Depending on applications, different alignment strategies can be applied to perform different genomic analyses. In the rest of this chapter, we present a comprehensive literature review that studies the existing FPGA-accelerated aligners, including BLAST and multiple sequence aligners. As they share some very similar properties with short read mapping, their implementations and experiments can provide insight and guidance to our work. The interplay

between the hardware characteristics of FPGA and the algorithmic techniques is also briefly mentioned and described.

### 2.2.1 Pairwise Sequence Alignment

A fundamental problem in the field of computational biology is the comparison and alignment of two sequences of DNA strands. Such analysis aims to identify the relationships between two sequences to reveal mutations, insertions or deletions of nucleotides from one biological sequence to another. Depending on the applications, the alignment results can provide useful biological or medical information such as evolutionary development of a species, or identification of causal cancer genes and genetic diseases [19].

The Smith-Waterman algorithm is the most commonly used method to perform pairwise sequence alignment (PSA). However, because of the enormous size of DNA sequences, essentially three billion *bp* in the human genome, software implementations of this algorithm suffer from prolonged execution latency. To accelerate PSA, reconfigurable devices have been extensively used to reduce the time complexity from $O(mn)$ in software to $O(m+n)$ on FPGA hardware.

#### Simple Aligner

In [20], the authors present one of the first FPGA accelerators for the Smith-Waterman algorithm. Reconfigurable systolic array is adapted to provide a large amount of parallelism for the computation of the scoring matrix $V$. The proposed design was tested on Virtex-II XC2V6000 and 3,225 billion entry/cell updates are generated per second (GCUPs). Similarly, Puttegowda *et al.* [21] propose a systolic architecture named HokieGene to accelerate PSA. This design targets Virtex-II XC2V6000 and it can generate 1,260 GCUPs.

The above implementations and the performance figures are, however, achieved using runtime reconfiguration, which directly writes one string into

the FPGA's bitstream, and the reconfiguration process can be a bottleneck for applications that require both sequences to be modified very often.

Therefore, Yu *et al.* [22] propose a reconfiguration-free implementation of the Smith-Waterman algorithm. In particular, the design of the accelerator is less FPGA device-specific and thus can be deployed on cross-vendor FPGA. Systolic Array is also used as the architecture and the proposed design can produce 814 GCUPs when implemented on Virtex-E XCV1000E-6.

### Affine Gap Cost Model

Very often, the alignment of two sequences favors gap extension rather than single insertion and deletion. Therefore, instead of giving a fixed negative score to every gap, biologists usually apply the affine gap penalty when computing the scoring matrix.

In [23] and [24], the authors propose the first FPGA-based accelerator that supports the affine gap model. Systolic matching cells are implemented to support different cost functions for the Smith-Waterman Algorithm. Compared to software implementation on Xeon 3 GHz processor, a speed-up of 370× can be achieved when implemented on Virtex-II Pro XC2VP70 FPGA.

Similarly, Jiang *et al.* [25] implement a reconfigurable accelerator that can adopt the affine gap penalty. In this design, a modified equation is proposed to improve the mapping efficiency of a processing element (PE). A special floor plan is applied to the fine-grain parallel PE array to cut down their routing delay. With these two techniques, the proposed implementation on Stratix EP1S30 can generate 6.6 GCUPs and improve the performance by 345× versus a similar software on Xeon 2.8 GHz processor.

Most of the research efforts such as [26–33] utilize systolic array or fine-grain PE architecture to accelerate PSA with the affine gap penalty. Experiments show that, when compared to state-of-the-art software implementations, the reconfigurable accelerators can achieve a speed-up from around 40× to 246×.

### Accelerator with Traceback

To further improve the accelerator performance, some FPGA designs realize the traceback procedure instead of relying on the host CPU to perform backtracking. For example, Benkrid *et al.* [34] implement a Smith-Waterman accelerator on Virtex-II XC2VP100. A pipeline of PEs can be used to calculate the scoring matrix and traceback. An improved accelerator is later proposed in [35] in which a space-efficient algorithm is used to overcome the memory size and bandwidth limitations. Compared to software on Core2 Duo 2.4 GHz, a performance gain over 300× can be obtained with 256 PEs on Virtex-4 FX100 FPGA.

Furthermore, a few researchers accelerate variants of the Smith-Waterman such as DIALIGN [36] to accomplish better alignment sensitivity. For example, Boukerche *et al.* [37] propose a reconfigurable accelerator for DIALIGN by implementing wavefront array processors on Stratix-II EP2S180. The traceback procedure can also be executed on FPGA to retrieve the alignment and the overall speed-up is around 141× compared to a similar software implementation.

Finally, Sebastião *et al.* [38] and Fei *et al.* [39] propose Smith-Waterman accelerators that analyze the computing features of this particular dynamic problem. By eliminating the storage requirements and data dependency, especially for the backtracking stage, these accelerators can achieve at least 3× speed-up versus software.

### Hardware Abstraction in FPGA-PSA

Some of the efforts are devoted to improving the portability and usability of the accelerated system. In [40], the authors design a systolic architecture that can be applied to solve general dynamic programming-based alignment problems. Others such as [24, 34, 41] provide generic, parameterizable FPGA cores for PSA which are portable across various FPGA platforms. Liu *et*

*al.* [42] introduce the concept of "accelerator on the cloud" where a web server is used to serve alignment requests. All these implementations, compared to state-of-the-art CPU designs, can deliver a speed-up of more than 62×. Finally, Greaves *et al.* [43] tackle the problem from another approach where the software engineer can model the Smith-Waterman algorithm in C# and automatically compile the program into parallel architecture on FPGA.

### 2.2.2 Database Search

Computational search through large databases of DNA is one of the important tools to reveal homologous sequences in modern molecular biology. Database sequences that display high similarity with the query are often expected to derive from the same ancestral sequence.

Heuristics such as BLAST [44] are extensively used by biologists to perform database search. Essentially, BLAST algorithm works in three consecutive stages:

1. Word Matching
2. Ungapped Extension
3. Gapped Extension

However, as the size of the most commonly used database such as NCBI databank [45] grows at the same pace as the increase in transistor counts based on Moore's law, running BLAST on a processor has been the bottleneck in homology analysis.

In this sub-section, previous work on the reconfigurable acceleration of BLAST is included as it shares some similar properties compared to short read alignment. Although some variations of BLAST such as BLASTp or BLASTx do not target the alignment of genetic nucleotides, they are also included in the discussion because of their similarities in heuristic and methodology. A summary of the previous work is shown in Table 2.4.

Table 2.4: Summary of the previous work on FPGA acceleration of database search.

| Category | Paper | Algorithm & Method | Co-Processor | Device | Max. Query Length Tested | Database Length | Speedup |
|---|---|---|---|---|---|---|---|
| Basic Accelerators | [46] | Smith-Waterman | Pentium-III 1 GHz | Virtex-E XCV2000E | 2,048 | 64 M bases | **330×** |
| | [47][48] | All Stages | - | Virtex-4 4VFX140 | 5,000 | 44 M bases | **215×** |
| Hybrid Systems | [49] | | PowerPC 405 300 MHz | Virtex-II PRO V2P30 | 200 k | - | **32×** |
| | [50] | Stage 1, 2 | Pentium-4 2.60 GHz | Stratix-II EP2S130C5 | 3,000 | 101 M bases | **48×** |
| | [51] | | Core i7 2.80 GHz | Virtex-5 ML509 | 3,000 | 123 M bases | **46×** |
| | [52] | Stage 1 | DRC coprocessor | Virtex-5 LX330 | 1 M | 1.4 G bases | **10×*** |
| Mercury BLAST | [53] | Stage 1 | Pentium-D 3 GHz | | 1 M | 1.16 G bases | **7×** |
| | [54] | Stage 1, 2 | Opteron 2 GHz | Virtex-II 6000 | 64,000 | 1.5 G bases | **11×** |
| | [55] | Stage 1, 2 (partially) | - | | 25,000 | 1.5 G bases | **50×** |
| Result Compatible | [56] | Function `Blast_Nt_Scan` | PowerPC 405 300 MHz | Virtex-4 ML410 | 975 M bases | 400 MB | **3×** |
| Database Pre-filter | [57] | Pre-filtering - Threshold | - | Virtex-5 XC5VLX330T | - | - | **5×** |
| | [58] | Pre-filtering - TreeBLAST | - | Stratix-III EP3SL340 | 1,000 | 7.17 G bases | **12×** |
| Hardware Abstraction in FPGA-BLAST | [59] | Two-Hit Method | - | Virtex-4 XC4VLX60 | - | - | **52×** |
| | [60] | Mitrion Virtual Processor | Itanium | Virtex-4 LX200 | 100 M | - | **20×** |
| | [61] | Smith-Waterman | Xeon E5620 | Stratix-IV E530 | 1.6 M | 975 M bases | **59×** |

### Basic Accelerators

One of the earliest efforts in accelerating BLAST on reconfigurable devices is the TUC BLAST. In [47, 48], Sotiriades *et al.* develop the first version of TUC BLAST in which the entire BLAST algorithm is mapped onto Virtex-4 4VFX140 FPGA. This architecture can support small queries of up to 1,000/5,000 letters regardless of the database size. Hash table is used to build hit finders and extension is done with basic comparators. Experiments indicate that the proposed accelerator can achieve a speed-up of 215× versus BLASTn on Xeon 2 GHz.

### Hybrid Systems

The above TUC BLAST is later on revised and incorporated with the PowerPC processor onboard to perform extension [49]. Implemented on Virtex-II PRO V2P30, the modified accelerator achieves 32× speed-up compared to the execution on Pentium-4 3.0 GHz. The authors also explore the design space

on ASIC to work around the limitations of FPGA in [62].

Xia *et al.* [50, 63, 64] also propose a hybrid architecture where the first two stages of BLAST are accelerated on Stratix-II EP2S130C5 FPGA and the final stage is executed on CPU. To decrease the on-chip memory requirement and support longer queries, a systolic array of 3072 PEs is used to perform multi-seeds detection and multi-channel hardware modules are implemented to complete ungapped extension. Experimental results demonstrate a speed-up of 48× versus Pentium-4 2.6 GHz CPU.

Furthermore, Guo *et al.* [51] implement a similar systolic array architecture which also supports long query ($\geq$ 3K). Word Matching and Ungapped Extension of BLAST are accelerated on Virtex-5 ML509 FPGA. However, this design is different from Xia *et al.* as look-up tables (LUTs) are used to implement the hit detection module instead of block memories (BRAMs) to increase the throughput of Word Matching. Experiments show that this particular implementation can deliver 46× speed-up when compared to the Core i7 processor on Dell Precision T1500 PC.

Chen *et al.* [52] also present an FPGA-based reconfigurable architecture to accelerate the word-matching stage of BLAST while maintaining the computations of other stages on CPU. This design consists of three sub-stages, a parallel Bloom filter, an off-chip hash table, and a match redundancy eliminator. The performance of this architecture, when implemented on Virtex-5 LX330, demonstrates 10× speed-up against Core2 Duo 3.2 GHz (1-thread) in Word Matching*.

### Mercury BLAST

The Mercury system is a platform that consists of reconfigurable logic associated with the disk controller, so as to provide computation in close proximity to the data in the backing store [65]. This platform is frequently employed to accelerate BLAST because of its high-throughput data channels.

Krishnamurthy *et al.* [53] develop the first Mercury BLAST implementa-

tion on Virtex-II 6000. The Word Matching stage is accelerated with Bloom filters and a hash table. Since the first stage is found to be the bottleneck in BLAST execution, this accelerator can demonstrate 7× overall speed-up against Pentium-4 2.8 GHz.

Mercury BLAST is then improved by Buhler *et al.* [54] where Word Matching and Ungapped Extension are both accelerated on FPGA. The FPGA-accelerated ungapped extension employs a similar heuristic as BLAST. This approach can achieve a speed-up of 11× while retaining 98.5-99% of all alignments found by NCBI BLASTN.

Finally, Lancaster *et al.* [55] further improve the design by implementing a pre-filter on FPGA for the third stage and at the same time offloading the computation of ungapped extension on CPU. By highly paralleling and pipelining the hardware modules, the accelerator accepts query of 25k bases and achieves 50× improvement. The same sensitivity can still be maintained when compared to the BLAST software.

Another type of BLAST, BLASTp is also accelerated using the Mercury framework. Word Matching is accelerated in [66] and Gapped Extension is accelerated with Smith-Waterman in [67]. Finally, [68] presents a full acceleration of BLASTp where all the previous efforts are combined to provide a full implementation.

#### Single-Pass BLAST

Since BLAST involves multiple passes during database queries, some researchers introduce a new algorithm that operates a single-pass, streaming rate mechanism to improve performance. For example, Herbordt *et al.* [69, 70] propose the use of a dynamic programming approach on FPGA to emulate the seeding and extension phases of BLAST. This algorithm, TreeBLAST, can improve the performance of the database search by 400× on Virtex-4 LX160 FPGA compared to multiple-pass NCBI BLASTp on Xeon 2.8 GHz.

### Results Compatible Accelerator

Although the mentioned implementations demonstrate significant speed-up compared to software, the search outcomes are not always consistent with the NCBI results. Since typical biologists would have no idea whether the differences are statistically significant, some FPGA researchers argue that the hardware-accelerated design should be NCBI BLAST compatible.

Datta *et al.* [56] propose a memory-efficient FPGA design that implements `Blast_Nt_Scan` function of BLAST. The primary function of the scan function is to stream the subject sequence data and to locate hits. Without compromising fidelity, the proposed implementation on Virtex-4 ML410 can improve performance by a factor of 3 (compared to Pentium-4 3.2 GHz) while in complete agreement with the standard NCBI BLAST.

### Database Pre-filtering

In addition to accelerating different phases of BLAST using FPGA, another useful approach to improve the overall performance is to profile the codebase and reduce the database size. One idea is to quickly reduce the size of the database to a small fraction, and then use the original NCBI BLAST code to process the query.

Afratis *et al.* [57] propose the first pre-filtering approach to BLAST by finding and reporting matches in the areas of high similarity between database and query. They found that pre-filtering can provide at least a factor of 5 and up to 3 orders of magnitude reduction in the database space.

Park *et al.* [58, 71] also apply pre-filtering with the TreeBLAST algorithm and quickly reduce the size of the database to a small fraction. The sensitivity of the pre-filtering approach is tuned to exceed that of the NCBI BLAST implementation so as to ensure identical results. Experimental results show that, compared to NCBI BLASTn, the speed-up is greater than 12× when pre-filtering and accelerator in [69] are used in the execution.

#### Hardware Abstraction in FPGA-BLAST

Despite the promising results described, the FPGA-based solutions should also be portable and straightforward to promote the use of reconfigurable accelerators among biologists.

In [72], Muriki *et al.* present the first portable, cost-effective and open-source solution of FPGA-accelerated BLAST to guarantee usability. However, this BLAST implementation on Xilinx 4085XLA is limited by the IO bandwidth and as a result, this approach is slower than software. Kasap *et al.* [59] also present a portable FPGA accelerator for BLAST by capturing the design with an FPGA-platform-independent language Handel-C. The architecture of the accelerator can be parametrized in terms of the sequence lengths, match scores, gap penalties, and cut-off and threshold values. It is reported that the hardware implementation is 52× faster than equivalent software implementations on Centrino Duo 2.2 GHz.

Moreover, Abelsson *et al.* [60] propose the use of Mitrion Virtual Processor to accelerate BLAST. Since Mitrion enables software developers to target FPGA-based computers without any requirements of the hardware design skills, users can continue using the familiar BLAST interface, while at the same time getting searches completed 10× to 20× faster.

Finally, Lam *et al.* [61] introduce an FPGA-accelerated BLAST on the cloud. To provide database search, the Smith-Waterman is accelerated on 64 Stratix-IV E530 on multiple PS4 compute nodes of Novo-G. A robust software interface is also provided to seamlessly integrate the FPGA design into the existing processing pipelines of NCBI BLAST.

### 2.2.3 Multiple Sequence Alignment

Multiple sequence alignment (MSA) is an extension to PSA and is generally used to construct family representations of sequences or to reveal evolutionary histories of species. However, it is an NP-Hard problem and therefore

the optimal solution can only be obtained using an $s$ dimensional dynamic programming table where $s$ is a number of sequences [73].

A heuristic algorithm, such as ClustalW [74], has been widely used among biologists because of its efficiency. ClustalW uses a progressive algorithm that consists of three major steps:

1. PSA between all sequences to generate a distance matrix
2. Guiding the tree generation based on the distance matrix
3. Successively building MSA by performing PSA based on the branching order of the guide tree

However, ClustalW faces the same problem as BLAST does due to the rapid growth of the sequence database, and aligning a few hundred sequences could require several hours on CPU.

Research has been done to overcome this problem by accelerating MSA with reconfigurable devices. In [75] and [76], the authors present an accelerated ClustalW by offloading the computation of the first stage onto FPGA. As more than 90% of the runtime is spent in the first stage , [75] provides a speed-up of 50× on Virtex-II XC2V6000 compared to Pentium-4 3 GHz for the first stage, and [76] achieves 10× performance improvement when Stratix PEIS30 is used instead of Xeon 2.8 GHz.

Finally, the third stage of ClustalW is accelerated in [77]. Compared to Core2 2.4 GHz, an overall speed-up of 150× can be achieved by reducing subgroups of aligned sequences into discrete profiles before PSA is performed on Virtex-4 FX100.

### 2.2.4 Mapping

As mentioned, mapping is one of the dominant applications of next-generation sequencing where millions of short reads are generated by NGS machine and mapped to the reference genome. Software such as Bowtie, Bowtie2 and BWA-MEM is widely used among biologists as de facto sequence alignment programs

Table 2.5: Summary of the previous work on FPGA acceleration of short read alignment.

| **Category** | **Paper** | **Algorithm & Method** | **Platform** | **Device** | **Speedup** | **M$bps^{-1}$** |
|---|---|---|---|---|---|---|
| Basic Mapper | [78] | Brute-Force | - | Virtex-5 LX330 | **4×** | 0.00567 |
| Approximate String Matching | [79] | Hash + Needleman-Wunsch | - | Virtex-5 LX330 | **5.2×** | 3.57 |
| | [80] | BFAST SW | Pico Computing M-503 | Virtex-6 LX240T×8 | **31×** | 112 |
| | [81] | FM-index | Convey HC-1 | Virtex-5 LX330×4 | **2.43×** | 13.2 |
| Accurate Mapper | [82] | Compact Linear Systolic | - | Virtex-6 LX550T | **1×** | 0.13 |
| | [83] | Bowtie2 + Smith-Waterman | Maxeler MAX5C | Virtex Ultrascale+ VU9P | **35%** | 1.03 |
| Runtime Reconfigurable Mappers | [13] | FM-index + Smith-Waterman | Maxeler MAX3 | Virtex-6 SX475T | **72.2×** | 151 |
| | [6] | FM-index | Maxeler MAX3 | Virtex-6 SX475T | **18.1×** | 97.8 |
| | [84] | FM-index | Maxeler MPC-X1000 | Stratix-V×8 | **14.9×** | 66.4 |
| Hybrid Systems | [85] | PerM Algorithm | Blade Server (AMD 2.8GHz CPU×2) | Virtex-5 LX330 | **42.9×** | 8.00 |
| | [86] | BWT | Pico Computing M-505 | FPGAs×12 | **48×** | - |
| | [87] | Dynamic Programming | IBM Power8 S824L | Virtex-7 VX690T | **1.86×** | 2.50 |
| BWA-MEM Accelerator | [88][89] | Smith-Waterman | Alpha Data ADM-PCIE-7V3 | Virtex-7 VX690T-2 | **2×** | 4.1 |
| | [90] | Smith-Waterman | - | Virtex-7 VC707 | **26.4×** | - |
| | [91] | BWT | Inter-Altera HARP | Stratix-V | **26%**$^{\dagger}$ | - |

of choice. Yet, the sequencing machine is improving at a rate faster than the transistor counts according to Moore's law. Mapping the generated sequence such as the complete human genome is taking an order of day's worth of computing time [73]. Therefore, FPGA technology has been extensively studied by researchers to speed up the mapping process. A summary of the previous work on the reconfigurable acceleration of short read alignment is displayed in Table 2.5. To allow a fair comparison, we define a normalized metric, base pairs aligned per second ($bps^{-1}$), which is given by:

$$bps^{-1} = \frac{read\ length \times read\ count}{alignment\ time} \tag{2.6}$$

### Basic Mappers

Fernandez *et al.* [78] implement the first hardware short read mapper in 2010 where the design is based on a naive solution. The reconfigurable implementation on Virtex-5 LX330 delivers a speed-up of 1.6× to 4× when compared to the fastest software tool RAMP [92] and ELAND [93] on Xeon Harpertown 2.5 GHz (1-thread) processor. However, the performance of this design

decreases with the increase of read length, therefore a followed-up work [94] is proposed in which the authors develop the first implementation of FM-index on FPGA. As the FM-index does not need to perform all character matching for the reference compared to the naive solution, this approach, when implemented on Virtex-6 LX760, outperforms the previous work by around 2× and more importantly, provides a 133× speed-up compared to Bowtie on Xeon 2.5 GHz (1-thread) processor.

### Approximate String Matching

Since [94] is only limited to exact string matching, the authors extend their work as a multi-threaded FPGA design called FHAST which supports up to 2 mismatches [81]. In this implementation of FM-index, each read represents a thread in the search and maximally 512 concurrent threads can be executed using a single Virtex-5 LX330 FPGA on Convey HC-1. Experimental results show that FHAST achieves a speed-up of up to 2.43× over Bowtie running on Xeon L540B and E5520 (16-thread) processors, and a second version that runs on Convey Computers HC-2ex provides higher sensitivity for a larger number of mismatches [95]. Using four Virtex-6 LX760 FPGAs, FHAST version-II can provide a speed-up up to 12× compared to Bowtie on two Xeon E5-2634 (8-thread) processors.

Besides FM-index, other researchers propose different FPGA-solution for approximate string matching. In [80], Olson *et al.* propose an accelerator that is based on indexing of reference with Smith-Waterman alignment both performed on FPGA. The authors optimize the size of the candidate alignment location (CAL) lookup table and partition the design into eight Pico M-503 boards each with one Virtex-6 LX240T FPGA. This 8-FPGA system can achieve 31× speed-up versus Bowtie running on two Xeon E-5520 (8-thread) processors.

Chen *et al.* [79, 96] also implement an accelerated short read aligner based on the seed-and-extension strategy. The basic idea of such strategy rests on

the heuristic that only a limited amount of errors (substitution, insertion and deletion) exists for a significant amount of alignments and therefore long exact match regions would exist. Thus aligning the exact matches, i.e. *seed* first, and then extending to both directions of the sequence for approximate string matching can reduce the search space enormously.

In these implementations, Chen *et al.* use a hash table as the seed engine and apply Needleman-Wunsch search [97], a dynamic programming algorithm, for the extension. Using a Virtex-5 LX330 device, the hardware aligners can achieve a speed-up between 2.5× to 5.9× compared to BWA [98] at a higher sensitivity.

### Highly Accurate Mappers

On the other hand, Knodel *et al.* [99] design a short read mapper on FPGA that allows a freely adjustable character mismatch threshold. This mapper is based on a brute-force approach that relies on a massive amount of shift registers using BRAMs and comparators to perform matching, and it guarantees a 100% mapping rate within the mismatch threshold. Compared to Bowtie on Core2 Duo 2.66 GHz (2-thread) processor, the hardware mapper can run 2× faster and can align 20% more genome when implemented on Virtex-6 LX240T FPGA.

The authors continue their work and design another short read mapper based on a linear systolic computation scheme to achieve better performance [82]. Implemented on Virtex-6 LX550T FPGA, the hardware mapper reports 2× more locations than Bowtie while maintaining the execution latency competitive to software executed on i7-2600K (4-thread) processor. This solution is also migrated onto Virtex-7 VX485T and is released as an open-source package called PoC-Align [100].

Recently, Koliogeorgi *et al.* [83] design a Bowtie2 accelerator where the extension stage is accelerated on FPGA. The extension and backtracking of the Smith-Waterman algorithm account for 60% of the entire Bowtie2 execution.

By leveraging techniques such as data interleaving and double buffering, the accelerated design is guaranteed to deliver exactly the same results as Bowtie2 while providing 35% overall performance gain.

### Runtime Reconfigurable Mappers

Some researchers manage to take advantage of the reconfigurable property of the FPGA device to further improve the performance of hardware short read aligners. In [101] and [13], Arram *et al.* introduce a hardware design that incorporates specialized matchers for exact and approximate sequence alignment, while at the same time runtime reconfiguration is used to fully populate the FPGA with each type of matchers. This decoupling enables the flexibility of optimizing each matcher according to the intended workload, hence resulting in higher parallelism and performance. With this scheme, results reported on Virtex-6 SX475T within Maxeler MAX3 are 72.2× faster than BWA, and 56× faster than Bowtie on Xeon X5650 (20-thread) processor.

Using the same approach, the authors further extend their work and design specialized filters that can align short reads to a reference genome with different edit distances [6]. These filters are arranged in a pipeline according to an increasing edit distance, in which reads unable to be mapped by a given filter are forwarded to the next filter in the pipeline for further processing. Specifically, each time the FPGA is fully populated with each filter in the pipeline in turn with runtime reconfiguration. With specialized filters based on a novel bi-directional backtracking version of the FM-index, it is found that the alignment time on Maxeler MAX3 can be up to 18× faster than BWA running on two X5650 (12-thread) processors.

### Hybrid Systems

Hybrid aligner refers to the concept of hardware-software co-design for accelerating short read alignment. In [86], Draghicescu *et al.* design BWT aligners

on twelve Virtex-5 505 FPGAs within the Pico Computing's machine. The accelerator ties into existing BWA software and allows the CPU to perform tasks that it is optimized for, such as file handling and memory management. The proposed system can achieve 48× speed-up compared to the software version of BWA running on two Xeon CPUs.

Tang *et al.* [85] also develop a hybrid accelerator where a host program running on PC is dedicated to controlling the loading and storing of reads and references data to and from the hardware. The hardware mapper is based on PerM [102], a software with periodic spaced seeds to significantly improve mapping efficiency for large reference genomes. Meshes of processing elements are implemented on Virtex-5 LX330 to take the advantage of the spatial parallelism on FPGA. Experiments show that this accelerator can deliver 22.2× to 42.9× speed-up versus PerM on a 6-core Xeon (Westmere) processor.

Recently, Banerjee *et al.* [87] propose an ASAP aligner that accelerates Levenshtein distance calculation based on dynamic programming. The design is implemented on Virtex-7 VX690T FPGA in an IBM POWER8 system that uses the CAPI interface for cache coherence across the CPU and FPGA. Experiments show that the accelerator is 2× faster for an end-to-end alignment tool running on POWER8 CPU. Other mentioned efforts, such as [96] and [100], are also tightly coupled with the software environment and presented as a hybrid system to accelerate short read alignment.

#### Acceleration of BWA-MEM

Some research efforts are devoted to accelerating certain alignment software. In particular, BWA-MEM has been widely studied and accelerated by FPGA researchers because of the accuracy and improved efficiency of the software [12]. Basically, the BWA-MEM algorithm consists of three main steps which are executed in succession for each read in the input:

1. SMEM (i.e. seeds) Generation

2. Seed Extension

3. Output Generation

In [90], Chen *et al.* propose an acceleration engine for BWA-MEM by offloading the seed extension, which is the computation bottleneck, onto Virtex VC707 FPGA. The authors develop an efficient Smith-Waterman implementation that supports massive task-level parallelism, sharply varied input sizes, and software-pruning strategies. Compared to BWA-MEM software on a 6-core processor with 24 threads, the proposed design can demonstrate 26.4× improvement in execution latency. The authors continue their work by offloading SMEM generation onto the FPGA in the latest Intel-Altera HARP system [91]. With a 16-PE accelerator engine, the generation of the seed is accelerated by 4×, and the overall SMEM seeding stage by 26% when compared with 16-thread CPU execution†. This work is later on improved by Cong *et al.* [103] where SMEM generation stage is further improved and a non-blocking pipeline methodology is applied to hide the latency off-chip memory access. This improves the accelerator by 6.3× and reduces 43% resource consumption.

Houtgast *et al.* [88, 89] also implement a hardware aligner based on BWA-MEM. The design is composed of a systolic array architecture to accelerate seed extension kernel with the Smith-Waterman. By offloading the computational bottleneck onto Virtex-7 VX690T-2 FPGA, the entire system can deliver a total acceleration of about 45%. This work is later extended by Ahmed *et al.* [104] where a hardware suffix array is used to partially accelerate SMEM generation, which enables a total application acceleration of 2.6× compared to the original software version.

## 2.3 Summary

In this chapter, we have explained the background information that forms the foundation of this work which includes the FM-index technique and the Smith-

Waterman algorithm. We have discussed the related work on various types of FPGA-accelerated aligners for DNA sequences. We have also presented the comprehensive literature review in this chapter in [105]. The following chapters form the main contributions of this work: 1) in Chapter 3 we accelerate exact-match and mismatch alignment strategies, and present novel methods that maximize the alignment performance and accuracy; 2) in Chapter 4 we accelerate the seed-and-extend strategy that performs indel alignment, and investigate the relationship between speed-up and accuracy for different alignment pipelines; 3) in Chapter 5 we present an approach for merging multiple FPGA designs into a one, and provide further optimization on one of the implemented aligners.

# Chapter 3

# Alignment with Exact-Match and Mismatch

The exact-match and mismatch alignment strategies are broadly used in state-of-the-art software aligners such as Bowtie. In particular, the exact-match strategy plays an important role in the entire alignment workflow, as it is statically found that around 70-80% of reads can be aligned exactly to the reference. Therefore, the exact-match strategy contributes to a major part of the computations.

The remaining 20-30% of reads that fail to align exactly to the reference are then processed by the approximate string matcher, such as different mismatch strategies. Depending on the alignment requirements, one-mismatch, two-mismatch and even three-mismatch strategies are used to identify genetic variations. In state-of-the-art software such as Bowtie, it supports alignment up to 3 mismatches to accommodate several mapping requirements:

1. Many of the reads have at least one good, valid alignment.

2. Many of the reads are relatively high-quality.

3. the number of alignments reported per read is close to 1.

In this chapter, we present a novel approach that accelerates exact-match

and mismatch alignment strategies based on the FM-index technique. This accelerator employs complete biological information including quality metric and ambiguous characters for biologically accurate short read alignment. We exploit a **multi-configuration pipeline** that aligns reads with different edit-distance-based strategies using **runtime reconfiguration**. These strategies, which include exact-match, one-mismatch, two-mismatch, and three-mismatch strategies, are arranged in ascending edit-distance order, resembling a similar workflow compared to Bowtie. Unlike previous designs targeting an individual platform, we accelerate the memory access in computations to aid performance portability, by partitioning the FM-index into buckets with the size equal to multiple of the memory burst size.

Ultimately, we aim to achieve the contributions 1 and 3 stated in Chapter 1 through the following methods:

1. We propose a novel alignment architecture that is composed of a four-stage configuration alignment pipeline. It exploits the reconfigurability of FPGA to achieve highly efficient implementation for each configuration and a fully optimized alignment pipeline.

2. We optimize the alignment architecture using different optimization techniques, which include index compression, data interleaving, and bi-directional FM-index for mismatch alignment, while maximizing the alignment accuracy.

3. We experiment with the optimal architecture based on the target platform Xilinx VU9P, together with comparisons against state-of-the-art software Bowtie on multi-core processors and some of the existing FPGA solutions.

The rest of the chapter is organized as follows: in Section 3.1 we introduce the proposed multi-configuration alignment pipeline; in Section 3.2 we present our FM-index implementations and the details of the optimization techniques;

in Section 3.3 we present the performance and accuracy comparison of our proposed alignment pipeline versus other designs.

## 3.1 Multi-configuration Alignment Pipeline

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

### 3.1.1 Motivation for Runtime Reconfiguration

Previous efforts that accelerate alignment with FPGA usually rely on a **static architecture**. Essentially, the target device is configured with an implementation that is functionally equivalent to multiple alignment strategies. For example, the FPGA implementation in [81] is composed of different alignment modules to perform filtered search with FM-index, where reads unaligned by one strategy (**filter**) are directed to the following one. Respective modules are grouped to form exact-match, one-mismatch, and two-mismatch strategies and are interconnected to assemble one implementation. Although a static configuration of these strategies as a single implementation eliminates the time for reconfiguration, this approach has its limitations which reduce the overall performance and subsequently nullify the benefit of discarding reconfiguration. The main motivations for the runtime reconfigurable architecture can be summarized as follows:

**Significant Amount of Data Hazards -** A typical alignment workflow is

composed of multiple steps where data from the current step rely on the results from the previous one. In software such as Bowtie, a read, by default, is only handled by the mismatch strategies if an exact-match is failed, *i.e.* a **filtered search** with the exact-match then mismatches. As the occurrences of mismatch reads are unpredictable, this results in non-stationary workload and incurs numerous data hazards for a statically configured circuit. Since all the modules for different steps are mapped onto FPGA, data hazards reduce the FPGA efficiency because some modules are left idle occasionally.

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

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

### 3.1.2 Proposed Alignment Pipeline

To guarantee the biological validity and reproducibility of the alignment results, the proposed pipeline follows a similar workflow compared to state-of-

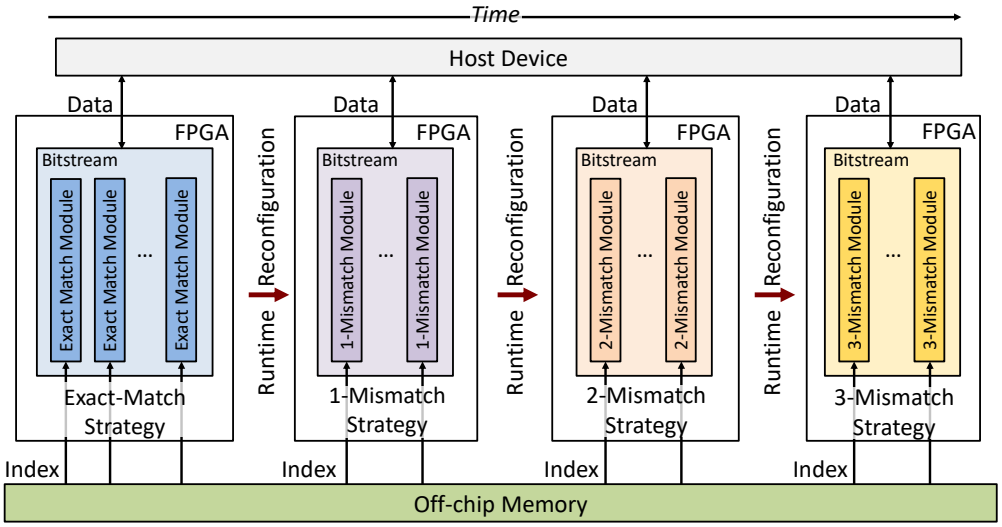



Figure 3.1: Alignment pipeline of the proposed architecture with runtime reconfiguration. Each implementation of a strategy is composed of a homogeneous array of modules. The module counts within an implementation are given by Equation (3.1).

the-art software Bowtie. We use FM-index to implement filtered search on FPGA with runtime reconfiguration. As illustrated in Figure 3.1 and Algorithm 2, strategies are arranged in a pipeline with the order: exact-match, one-mismatch, two-mismatch, and three-mismatch where each configuration is composed of a homogeneous array of modules. The FM-index, which is of a few GB, is stored on the external DDR memory attached to the FPGA. Here we analyze the design space for each alignment module and provide back of envelope estimations for the corresponding performance. The key parameters used in this analysis are defined in Table 3.1.

According to Algorithm 1 in Chapter 2, alignment with FM-index involves random access to an index location. Therefore, the index needs to be in proximity to FPGA to shorten the access time. In our proposed FM-index circuit, the index is stored on the external DDR memory attached to the FPGA, with a copy of the index stored in each DIMM. Each module is connected to a bank of external DDR memory using one memory channel, since the memory access from each module does not interfere with others. For this reason, the number

```
1  Function Mapping(Data):
2      do in parallel
3          stream Data from the host
4          process the transferred Data
5          stream aligned and unaligned Data back to the host
6      end
7  load configuration exact-match                          //FM-index
8  Mapping(all the reads)
9  load configuration one-mismatch            //backtracking FM-index
10 Mapping(unaligned reads R_na0 from line 8);
11 load configuration two-mismatch            //backtracking FM-index
12 Mapping(unaligned reads R_na1 from line 10)
13 load configuration three-mismatch                       //FM-index
14 partition unaligned reads R_na2 from line 12 into four seeds [S_0 : S_3]
15 Mapping( [S_0 : S_3])
16 compare on CPU ∀ aligned [S_0 : S_3]
```

**Algorithm 2:** The workflow of the proposed alignment pipeline with runtime reconfiguration. Details about the design and implementation of each configuration are provided in Section 3.2.

of DIMMs available confines the number of modules that can be replicated on FPGA. The number of times a module can be replicated is given by:

$$M_j = \min\left(\frac{R_T - R_{PCI} - R_{DDR}}{r_j}, N_{DIMM}\right) \tag{3.1}$$

The performance of the FM-index circuit can be modeled by (3.2). The overhead of this architecture is the reconfiguration time $t_{re}$ and the data communication overhead between the host $t_{cj}$. For a typical alignment process with more than a million reads to align, these two numbers are negligible as the alignment time is the dominant factor. Note that the alignment starts off as soon as the first read arrives from the host, $t_{cj}$ can be hidden by the alignment latency.

$$T = \sum_j \left(t_{re} + \max\left(t_{cj}, \frac{N_j t_j}{M_j}\right)\right) \tag{3.2}$$

Table 3.1: Key system parameters. Note that $j$-mismatch with $j = 0$ represents exact-match alignment strategy.

| | |
|---|---|
| $R_T$ | Total available resource on the target FPGA |
| $R_{PCI}$, $R_{DDR}$ | Resources for PCI-e, memory controllers |
| $N_{DIMM}$ | Number of memory channels for onboard DRAM |
| $r_j$ | Resources required by an FM-index module with $j$-mismatch (where $j = 0, 1, 2$) |
| $M_j$ | Number of modules for FM-index with $j$-mismatch |
| $t_{re}$, $t_{cj}$ | Time for runtime reconfiguration, and communication time for the $j$-mismatch strategy |
| $t_j$ | Time to align one read by the $j$-mismatch strategy |
| $N_j$ | Number of reads processed by the $j$-mismatch strategy |

## 3.2 Design of the Alignment Accelerator

According to Algorithm 1, the bottleneck of FM-index based alignment is the memory access where a character search involves access to a random memory location. Therefore, in this section, we detail the techniques that utilize the complete memory bandwidth and improve the performance of FM-index. This section also presents the FPGA accelerated aligner where details and collaboration between modules are described. In particular, the modules for handling the quality metric and ambiguous characters are elaborated. The techniques used to further improve the alignment performance are discussed as well.

### 3.2.1 Bucketing

#### Index Compression

Although FM-index is a space-efficient data structure that permits fast substring matching, when indexed, the human reference genome is around 51 GB. This is often far larger than the capacity of off-chip memory. To reduce the memory footprint, we store a subset of $c(\iota, x)$ of FM-index, while substitut-

ing the remaining entries with a portion of the original BWT. Specifically, we sample every $d$ entry of $c()$ and pack the BWT in the range of every $d$ and $d-1$ alongside, forming a ***bucket***. During a character search, the missing entries in $c()$ can be recalculated on the fly using the BWT $B$. Figure 3.2 shows the final compressed FM-index $F$ and Algorithm 3 demonstrates the corresponding procedure for substring matching.

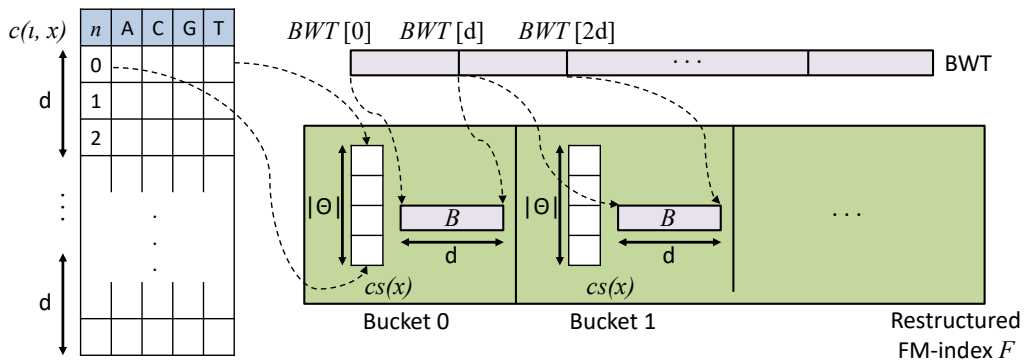



Figure 3.2: The proposed restructured FM-index.

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

$$\begin{aligned}
&\text{Sampled } c(\iota, x) \text{ Size} + \text{Sampled } BWT \text{ Size} \\
=&\frac{|R_{human}| \times (32bit)}{d} \times |\Theta| + |R_{human}| \times (\log_2(|\Theta|)bit) \\
=&\frac{3.2GB \times 4}{d} \times 4 + 3.2GB \times \frac{2}{8} \\
=&(\frac{51.2}{d} + 0.8)GB
\end{aligned} \tag{3.3}$$

**Sampling Distance** $d$

The value of $d$ plays an important role in memory access efficiency. First, it affects the final size of the compressed index. More importantly, it changes the bucket size and influences alignment performance. According to Algorithm 3,

```
Data: Pattern P, Compressed FM-index F, SA of Reference R
Result: The starting locations Loc of P in R
l = |P| − 1
(top, bottom) = (0, c(|R|, P[l])) //Assume we always store c(|R|, x)
for j = l to 0 do
    top = F[top/d].cs(P[j])+
            Count(P[j], F[top/d].B, top%d) + i(P[j])
    bottom = F[bottom/d].cs(P[j])+
            Count(P[j], F[bottom/d].B , bottom%d) + i(P[j])
end
if top < bottom then
    for j = 0 to top − bottom do
        Loc[j] = SA[top + j]
    end
end

Function Count(char, B, position):
    cnt = 0
    for j = 0 to position do
        if char == B[j] then
            cnt + +
        end
    end
    return cnt
```

**Algorithm 3:** Algorithm for exact substring matching using compressed FM-index. Note some notations are specified in Figure 3.2.

every character search of $P$ requires streaming of a bucket from the external memory. To enable better utilization of the memory bandwidth, $d$ should be set based on Equation (3.3) such that the final bucket size is a multiple of the memory burst size. For typical DDR4 memory with a burst size 64 B, $d = 192$. The final index size becomes 1.06 GB.

### 3.2.2 Module Designs and Optimization

As mentioned, our work is inspired by Bowtie where a read is handled by the mismatch subroutine if exact matching is failed, *i.e.* a filtered search with the exact match then mismatches. Therefore, the hardware implementations of the aforementioned strategies are arranged in an alignment pipeline with runtime reconfiguration. Given by Equation 3.1, modules are replicated on FPGA to increase parallelism. Each module within the strategy implementation is also fully pipelined to maximize the matching throughput.

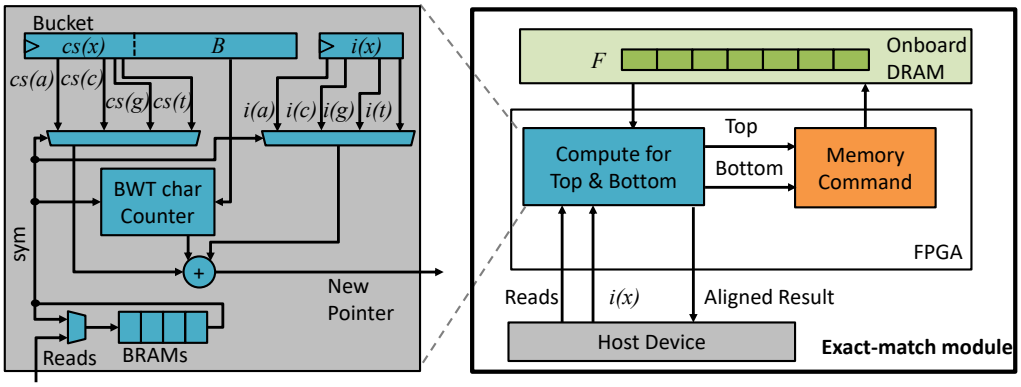



Figure 3.3: Simplified top-level diagram for an exact-match module. The diagram on the left shows the detail of the compute block for *Top* & *Bottom*. Note that for readability it only displays the logic for one pointer calculation.

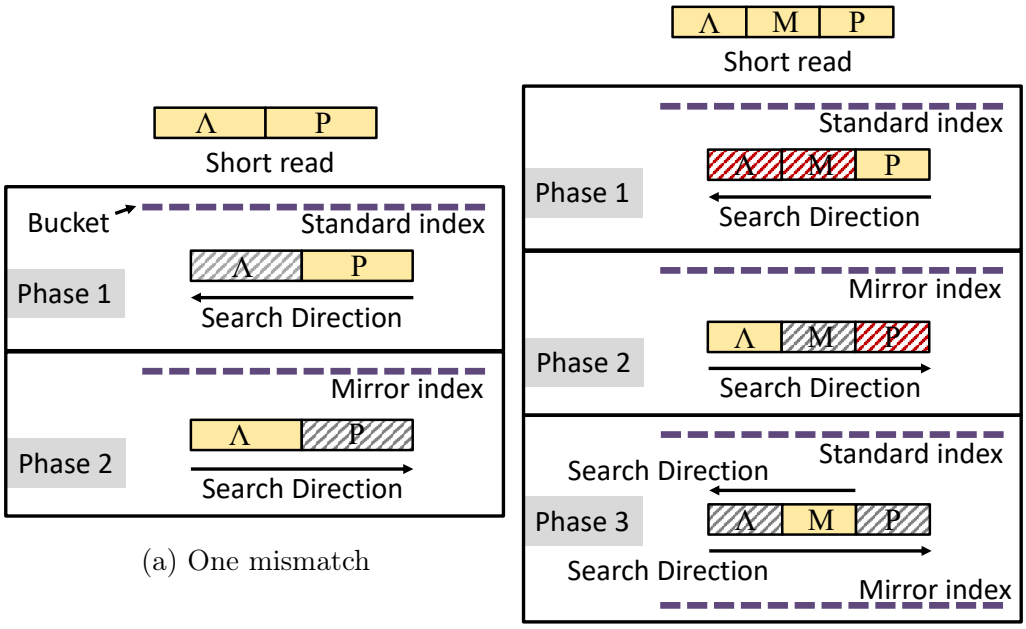
All strategies utilize our restructured FM-index to perform alignment. Figure 3.3 shows a simplified top-level diagram that implements Algorithm 3 for exact-match strategy. The process of alignment within all types of strategies begins with streaming the reads from the host to FPGA while the restructured FM-index $F$ is preloaded onto the onboard memory. The new pointers for *top*

and *bottom* are then calculated based on the symbol and the correlated buckets from the onboard memory. A command block is responsible for sending memory requests according to the new pointer values. Results are streamed back to the host once the reads are matched, or when the reads are unaligned and needed to redirect to the subsequent strategy.

To support mismatch alignment, additional logic is included in the exact-match strategy for one and two mismatch strategies to support backtracking. Since FM-index suffers from exponential scaling with the number of permitted mismatches, bi-directional FM-index is used in the one-mismatch and two-mismatch strategies to reduce the search space. Details about the bi-directional FM-index can be found in [4]. In addition, our three-mismatch strategy employs a seed-and-compare strategy. Seeds are exactly matched with the FM-index and the mismatch locations are identified in a subsequent direct comparison stage.

### Bi-directional FM-index

In the implementation of the one-mismatch strategy, we use two different indices of the reference. The first one contains the standard restructured FM-index of $R$, while the second one contains the index of the reversed of $R$, which is called the mirror index. This enables a character search from both the start or the end of a read. Figure 3.4a illustrates the mechanism of alignment using bi-directional FM-index. During the search, the mismatch character can either fall onto the left or right half of the read. Therefore, our one-mismatch strategy proceeds in two phases to handle these two cases: Phase 1 uses our restructured FM-index $F$ to begin the search from the end of the read, with the constraint that one mismatch occurs only in the left half of the read. Phase 2 uses the mirror index to begin the search from the start of the read, with the constraint that one mismatch occurs only in the right half. By constraining the mismatch position, a long segment of the read can be exactly matched to the reference initially. This can largely reduce the search space size without




(a) One mismatch

(b) Two mismatches

Figure 3.4: One and two-mismatch alignment using bi-directional FM-index. The segments shaded in grey are regions for testing one mismatch, while segments shaded in red are regions for testing two mismatches. The non-shaded areas are exact-match regions.

sacrificing any sensitivity.

The implementation of the two-mismatch strategy also utilizes the standard and mirror index to perform the search. To reduce the search space, the short read is partitioned into three parts (Left $\Lambda$, Middle $M$, Right $P$) with the same length. A reportable alignment falls into one of the three cases as illustrated in Figure 3.4b: (case I) 2 mismatches in $\Lambda M$; (case II) $\leq 1$ mismatch in $M$ and $\geq 1$ mismatch in $P$; (case III) 1 mismatch in $\Lambda$ and 1 mismatch in $P$. Therefore, the two-mismatch strategy proceeds in three phases corresponding to these three cases: Phase 1 begins with exact matching from the end of $P$, and attempts two-mismatch alignment once it reaches $M$, using the standard index. Phase 2 uses the mirror index to perform a similar search on the short reads, with the constraint of having at most one mismatch character in $M$. Phase 3 corporates the first 2 phases and finds partial alignment with one mismatch in $\Lambda$ using the standard index. Then it extends the partial alignment

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

### Seed-and-compare

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

### Data Interleaving

Although bucketing improves the utilization of memory throughput, the alignment performance still suffers from the latency of memory access. According to Algorithm 3, the new pointers for *top* and *bottom* are calculated based on the values from $F$, which in turn depends on *top* and *bottom* from existing iteration. In other words, when a symbol is processed, the requests for new memory pointers can only be known and made when the data from the previous request arrives. This incurs a vast amount of stall.

We tackle this problem by proposing an interleaving scheme to process multiple reads concurrently. Displayed on the left of Figure 3.3, a block memory (BRAM), which is initially empty after reconfiguration, is used to store a few short reads during the mapping process. In each clock cycle, a read is selected from BRAM and the next symbol is processed. This enables full utilization of the FPGA as almost one nucleotide can be processed and a memory request can be sent every clock cycle. From our experiment, the latency is 40.89 µs on

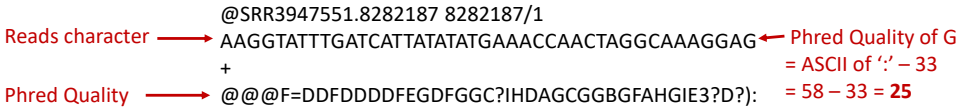



Figure 3.5: Explanation on Phred quality using an example.

average with DDR4-2400, which requires concurrent processing of 368 reads to negate the memory access latency. To enable fast evaluation of the interleaving scheme and the acquirement of this number, we use the proposed design analyzer and merger which will be explained in Chapter 5. Note that we apply the data interleaving technique to the implementations of all strategies.

### 3.2.3 Consideration for Complete Biological Information

Another major component of this work is the capability of recognizing the complete biological information in the alignment process. One important information is the quality value, also known as Phred quality [106], a metric that is usually neglected in previous work. Its values are ranged from 0 to 42 and are presented in ASCII characters (with an addition of 33). Figure 3.5 shows an example snippet of a read file where the second line is the read and the fourth line is Phred quality. Each character in the read is associated with a quality score in the same position and a larger value represents a better quality.

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

#### Challenges

Previous accelerators usually neglect Phred quality because of design challenges and performance issues. To start, Phred quality requires 6 bits per

value and there is no consensus on a practicable reduction of the resolution of the quality score [107]. This not only increases the communication overhead between the host and FPGA but also the resource usage. It also results in prolonged placement and routing time, more immense fan-out, and lower clock frequency. Furthermore, a read can have various combinations of mismatch locations that contribute to several aligned results per read. For example, given a reference $R$ = ACACGT and read pattern $P$ = ACC in Figure 3.6, the last character of $P$ can be replaced with A or G, forming the results ACA or ACG, for successful alignment with one mismatch. The consideration of Phred quality complicates the design logic and lengthens the alignment time, as we need to sort and rank these results based on the quality sum of each alignment.

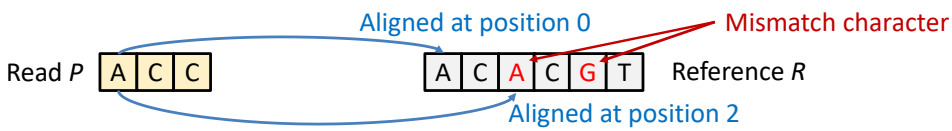



Figure 3.6: Example of one-mismatch alignment with two aligned results.

### Separated Computation and Minimal Ports

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

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

Note that the number of registers can be decreased by changing a parameter so as to accommodate smaller FPGA.

### On-the-fly Sorting

As shown in Figure 3.7, a set of comparators is used so that results with the highest quality values are always preserved. When a new alignment is found for particular mismatch location(s), these comparators match the quality sum of this aligned result with the existing ones from the shift registers. Accordingly, an insertion point can be obtained and the quality sums smaller than the current one are shifted in the shift registers. A simplified algorithm that illustrates the corresponding process is shown in Algorithm 4. Note the underlying logic is fully pipelined so that it can receive the aligned result from the compute block for *top* & *bottom* every cycle, despite the comparison and the shifting operations can take a few cycles.

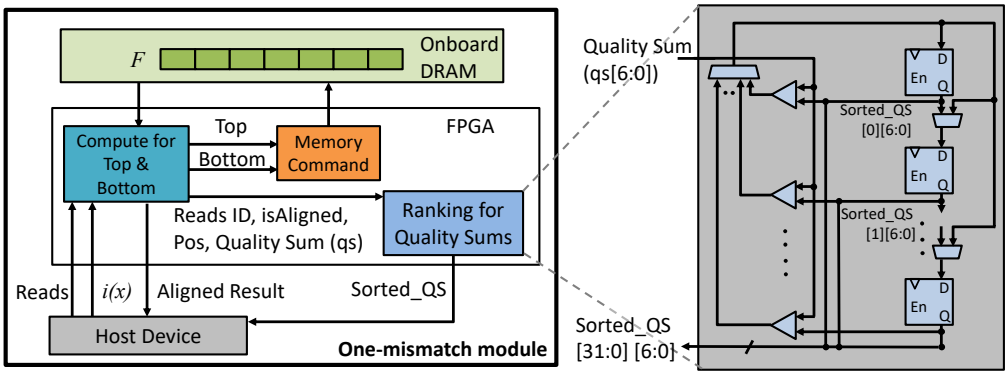



Figure 3.7: Simplified top-level diagram for one-mismatch module which contains the compute block for sorting quality sums of a read. For readability, some data and control paths are omitted.

### Three-mismatch Strategy and Reads containing Ambiguous Characters

Finally, Phred quality values are also considered in alignment with the three-mismatch strategy. The quality sum for each alignment is calculated during

```
Data:   Quality sum array QS ∀ mismatch locations of a read
Result: Sorted quality sum array Sorted_QS[31 : 0][6 : 0] for 32
        mismatch locations of a read

//Initialize Sorted_QS
for j = 0 to 31 do
    Sorted_QS[j] = 0          //Assume a qs value is always > 0
end

//Sorted_QS is rearranged in each iteration based on each
 input QS value
for i = 0 to len(QS)-1 do
    //Control and temporary variables
    gt_qs = False
    prev_qs = 0

    for j = 0 to 31 do
        if QS[i] > Sorted_QS[j] and gt_qs == False then
            gt_qs = True
            prev_qs = Sorted_QS[j]
            Sorted_QS[j] = QS[i]
            continue
        end
        //Start shifting along Sorted_QS
        if gt_qs == True then
            tmp_qs = Sorted_QS[j]
            Sorted_QS[j] = prev_qs
            prev_qs = tmp_qs
        end
    end
end
```

**Algorithm 4:** Simplified algorithm to rank the quality sum for the one-mismatch module.

the compare step on the processor, and only the ones with the highest values are reported. Additionally, reads containing the ambiguous characters (N characters) are handled by the processor as soon as the exact-match hardware starts processing. According to Bowtie, only alignments involving ambiguous characters in the read are legal, and ambiguous characters in the read mismatch all other characters. When an ambiguous character is seen in the input, the reads are processed directly on the processor to obtain the possible alignments with no more than three mismatches. Given that only a small portion of reads contains N characters, this step can be completely overlapped by the alignment on FPGA.

## 3.3 Evaluations and Discussion

To evaluate the performance and limitations of our proposed alignment architecture, we first investigate the alignment speed of each strategy, by adjusting module counts within each of the corresponding implementations. We then provide an accuracy and performance comparison between the proposed alignment pipeline and existing accelerators. To optimize the design from the memory access perspective and enable better utilization of the memory bandwidth, we set the parameter $d$ equal to the memory burst size of the target platform.

### 3.3.1 Evaluation of Individual Implementation

In this experiment, we use Maxeler's MAX5C DFE which is equipped with Xilinx Virtex UltraScale+ VU9P connected to three DIMMs of 16 GB on-board memory. As the memory burst size is 64 B, we set $d = 192$. Based on Equation 3.1, a module can be replicated at most by three times within each strategy on the target platform. The FPGA runs at 200 MHz while the host runs with Intel Xeon E5-2643 processor at 3.4 GHz and 64 GB DDR4-2400 memory. Centos 7.0 is installed on the host. MaxCompiler 2018.2 and Vivado

2017.4 are used for synthesis and implementation. PCI-e 2.0 is used to transfer the data between the host and FPGA. As the computation is bottleneck by the onboard memory on the FPGA, PCI-e 2.0 is already sufficient for the data transfer.

GRCh38 [108] is used as the reference and single-end reads from `human_100_300M` in [109] are used as the input. This dataset contains $300 \times 10^6$ reads with $101bp$, and was originally used to evaluate Bowtie and Bowtie2 on a multi-core system to investigate the alignment performance. It is composed of reads generated by the Illumina HiSeq2000 instrument from various genome projects.

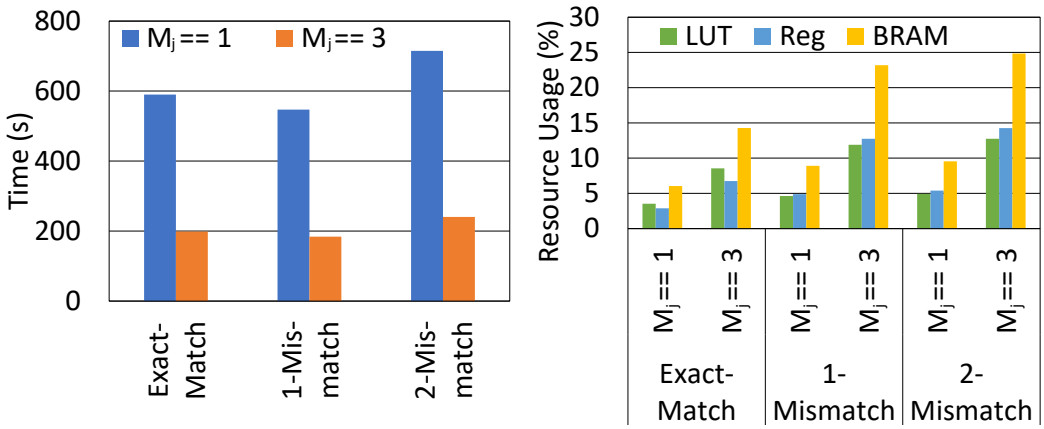



Figure 3.8: The alignment speed for different module counts of the exact-match, one-mismatch, and two-mismatch implementations (Left). The corresponding resource consumption with respect to the available resources is represented in percentage (Right).

Figure 3.8 shows the alignment speed for the implementations of the exact-match, one-mismatch, and two-mismatch strategies. For each strategy, the corresponding computational module is populated on the FPGA by one and three times to investigate the performance difference. The graph on the right elaborates on the corresponding resource consumption in percentage required by each strategy. The resource usage for DSP is not shown in the figures as less than 1 % is used in all the implementations.

As displayed in the figure, the alignment performance is dependent on the module counts. When the number of modules is tripled, the alignment per-

formance is improved by $3\times$. This showcases the scalability where the performance scales linearly with the module counts. It also indicates that measured results are closely compatible with Equation 3.2. The critical resource for each module is BRAM ($\sim 7-28\%$ usage) because of the circular buffer for concurrent processing. There remains an adequate amount of space for module replications should more memory DIMMs are given.

### 3.3.2 Overall Alignment Runtime and Accuracy

As our proposed hardware design is modeled upon the algorithms implemented in Bowtie, this subsection evaluates the alignment accuracy, with results from Bowtie acting as the ground truth. We use Bowtie 1.3.0 in this evaluation. Furthermore, we compare the performance difference between our aligner and software. We also select the accelerator proposed by Arram *et al.* [6] for comparison as it is based on a similar methodology to perform runtime alignment. We manage to download [110] and re-implement their design on our experimental device with a frequency of 200 MHz. Based on the evaluation from Section 3.3.1, we initialize the hardware with the maximum number of modules, $M_j = 3$, with all other settings identical to the previous sub-section.

Given the relatively small workload ($\sim$7 % of a full alignment workload), the reconfiguration time (9-12 s per configuration) for VU9P does not have a large impact on the overall alignment time. Moreover, different datasets and alignment parameters are used in different previous work, and some accelerators use more than one FPGA to perform the alignment. To allow a fair comparison, we use the normalized metric, base pairs aligned per second ($bps^{-1}$), given by Equation 2.6 in Section 2.2.4. Finally, we assume that the index $F$ is already buffered and preserved in the DRAM prior to each configuration.

Table 3.2 displays the comparison between the set-up and alignment results of the hardware and software. Accuracy is defined as the fraction of correct alignment and un-alignment from the proposed alignment pipeline among all

Table 3.2: A comparison between set-up and alignment results of Arram *et al.*, the proposed design and Bowtie.

| | **Arram *et al.*** | **Proposed Aligner** | **Bowtie** |
|---|---|---|---|
| **Device** | Xilinx Virtex UltraScale+ VU9P | | Intel Silver 4110 (16 threads) |
| **Frequency** | 200 MHz | | 2.1 GHz |
| **Lithography** | TSMC 16 nm | | Intel 14 nm |
| **Align time** | 638 s | 779 s | 3330 s |
| $bps^{-1}$ (million) | 47.5 | 38.5 | 9.10 |
| **Accuracy** | 96.3 % | 97.7 % | – |

of the reads. Correct alignment refers to reads matched to the same locations as Bowtie. Correct un-alignment means that if the reads are not aligned by Bowtie, they should not be aligned by the proposed aligner also.

The 2.3 % discrepancy in accuracy, between our accelerator and Bowtie (100 % accuracy for the ground truth), is due to two reasons. First, some of the reads are successfully aligned to the reference using FPGA but fail to map using Bowtie. Bowtie imposes a backtracking restriction to limit effort spent finding valid alignments. This causes Bowtie to miss some legal mismatch alignments. This clearly demonstrates the biological validity of the proposed aligner, since our design can exhaust all the possible mismatch locations without putting a backtracking limit. This is important for alignment accuracy because statically around 20 % of reads from the NGS machine can be aligned with one and two-mismatch strategies.

Second, Bowtie employs a heuristic to determine the mapping locations, especially when the reads have multiple aligned results. Some of the reads that can be successfully aligned by Bowtie are mapped to a different location in our proposed aligner. The accelerator proposed by Arram *et al.*, on the other hand, has lower accuracy. Despite the performance, the accuracy drops to 96.3 % because of the overlooking of the quality metric, as well as the missing stage of the three-mismatch strategy. Compared to Arram *et al.*,

our proposed aligner can report **4.48 M** more correct alignments. The better performance of Arram *et al.* is also due to the missing three-mismatch strategy, where short reads unaligned by the two-mismatch strategy are left unprocessed and redirected to the host straight away. If the three-mismatch strategy is omitted in our proposed aligner, our implementation needs around 646 s to finish processing all the reads which is similar to Arram's work.

Finally, the proposed aligner can achieve a reasonable speed-up of 4.27 × where the alignment time is decreased from **56 minutes** in software to around **13 minutes**. With the assurance of biological validity and reproducibility of the alignment results, the proposed aligner provides an opportunity to bridge the gap between alignment research and practice, allowing applications such as cancer diagnosis to become part of routine clinical procedures.

### 3.3.3 Performance Comparison with Existing Accelerators

Table 3.3 demonstrates the performance comparison of the proposed aligner with different accelerators. Since different designs conduct their evaluations using different datasets, we do not compare the alignment accuracy in this evaluation. Note that we only select the hardware aligners that are based on Bowtie, Bowtie2, or FM-index, as they are relatively similar to ours. Accelerators built upon other algorithms such as Minimap2 are not considered as they use different techniques or target long reads alignment.

The million $bps^{-1}$ values in Table 3.3 indicate that our aligner outperforms most of the existing designs apart from [13] and [6]. This showcases the benefits of our design where the addition of computation logic for quality sums does not affect alignment time. On the other hand, our aligner is slower than [13] and [6] mainly because of the availability of DIMMs on the respective devices. Their platforms are based on Maxeler MAX3 where there exist seven memory DIMMs onboard. With the number of memory channels more than double

Table 3.3: Performance comparison with previous hardware accelerators that are based on similar algorithms.

| **Year** | **Work** | **Algorithm & Method** | **Platform** | **Device** | **Read Count** (Million) | **Read Length** ($bp$) | **Align Time** | **M**$bps^{-1}$ |
|---|---|---|---|---|---|---|---|---|
| 2012 | [81] | FM-index | Convey HC-1 | Virtex-5 LX330×4 | 18 | 101 | 138 s | 3.29 |
| 2013 | [13] | FM-index + Smith-Waterman | Maxeler MAX3 | Virtex-6 SX475T | 82 | 90 | 49.0 s | 151 |
| 2013 | [6] | FM-index | Maxeler MAX3 | Virtex-6 SX475T | 18 | 75 | 13.8 s | 97.8 |
| 2015 | [84] | FM-index | Maxeler MPC-X1000 | Stratix-V×8 | 10 | 75 | 11.3 s | 66.4 |
| 2019 | [83] | Bowtie2 + Smith-Waterman | Maxeler MAX5C | Virtex Ultrascale+ VU9P | 60 | 100 | 5830 s | 1.03 |
| 2021 | Ours | FM-index | Maxeler MAX5C | Virtex Ultrascale+ VU9P | 300 | 101 | 779 s | 38.5 |

compared to Xilinx VU9P, the alignment speed is therefore more significant in their work. It is also important to note that their evaluation methods are based on the theoretical upper bound estimation on each respective device. Hence, the actual performance of their implemented designs might be poorer, as the routing congestion and fan-out are not taken into consideration. Finally, [13] only consists of two stages in the alignment pipeline, where the approximate matching is performed with the linear Smith-Waterman algorithm. Therefore, the alignment speed can be faster than the proposed aligner as we exhaust all possible mismatch locations using backtracking FM-index.

### 3.3.4 Resource Consumption

Based on the proposed reconfigurable architecture with $M_j = 3$, Table 3.4 indicates the corresponding resource consumption for the implementation of each strategy on VU9P, with the inclusion of the PCI-e and memory controller. With an adequate area remaining on the FPGA, more modules can be populated on the FPGA if the number of DIMMs increases. Note these numbers can only serve as a reference, as many factors can affect the final resource computation when migrated onto other platforms.

Table 3.4: Area cost of the final design on VU9P. Percentage values are relative to the available resources on target FPGA.

| | **LUT** | **Register** | **BRAM** | **DSP** |
|---|---|---|---|---|
| **Exact-match** | 101 092 (8.55%) | 159 260 (6.74%) | 754 (14.3%) | 9 (0.13%) |
| **1-Mismatch** | 140 726 (11.9%) | 301 540 (12.6%) | 1224 (23.18%) | 9 (0.13%) |
| **2-Mismatch** | 150 758 (12.8%) | 337 212 (14.3%) | 1312 (24.8%) | 9 (0.13%) |

## 3.4 Summary

In this chapter, we propose a novel, runtime reconfigurable architecture to accelerate exact-match and mismatch alignment strategies based on FM-index. The novel aspects of our proposed architecture are:

1. By leveraging the runtime reconfigurability of FPGA, individual strategies are used to align reads to the reference with a specific edit-distance, forming an alignment pipeline. This pipeline provides better performance and resource consumption for each strategy, which in turn achieves a fully optimized alignment pipeline.

2. The implementation of each strategy is optimized using the techniques: index compression, data interleaving, and bi-directional FM-index for mismatch alignment.

3. Complete biological information including quality metric and ambiguous characters are considered in our alignment pipeline for biologically accurate short read alignment. With alignment results from Bowtie acting as the ground truth, our design can achieve an accuracy of 97.7%.

In the following chapter, we explore different arrangements of alignment strategies and evaluate the corresponding accuracy for indel alignment.

# Chapter 4

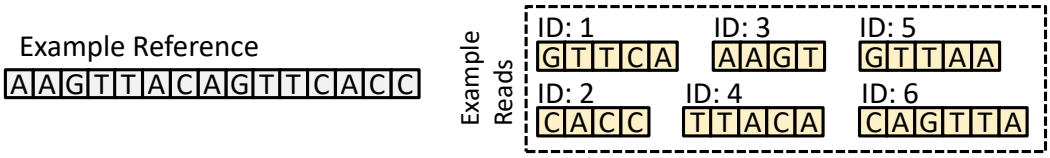
# Alignment with Indels

Insertion–deletion mutations (indels) refer to the insertion and/or deletion of nucleotides into sampled DNA and are often less than $1kbp$ in length [111]. Alignment with indels is important for the downstream analysis of NGS data, such as variant calling that identifies variants in the sequenced data. In particular, indels are usually implicated as the driving mechanism for many constitutional diseases such as cancer.

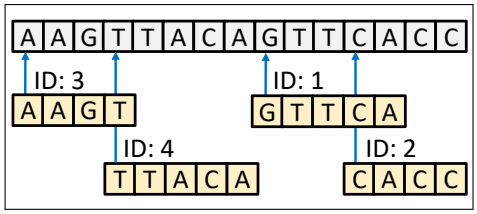
Indexing scheme, such as FM-index, is quite inefficient to perform gapped alignment to locate indels. These gaps substantially increase the search space and reduce the benefits of using a bi-directional FM-index, thereby slowing the aligner built upon only the index-based alignment.

State-of-the-art software generally extends the index-based aligner to enable gapped alignment by applying the seed-and-extend strategy. Essentially, the seeding stage extracts substrings, *i.e.* seeds, from a read and they are aligned to the reference contiguously using the index. Based on the obtained positions, the seeds are extended into full alignments using dynamic programming such as the Smith-Waterman algorithm. To further improve the alignment sensitivity, software such as Bowtie2 employs filtered search by applying the exact-match and even one-mismatch strategies before the seed-and-extend strategy.

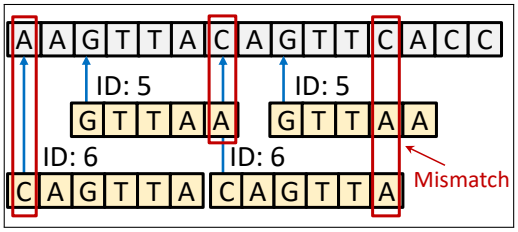
FPGAs have been used to speed up indel alignment by accelerating individual alignment strategies such as exact-match, seed-and-extend, etc as illustrated in Figure 4.1, and are combined and arranged into a consecutive, sequential, statically or runtime reconfigurable pipeline similar to the alignment pipeline elaborated in the previous chapter.



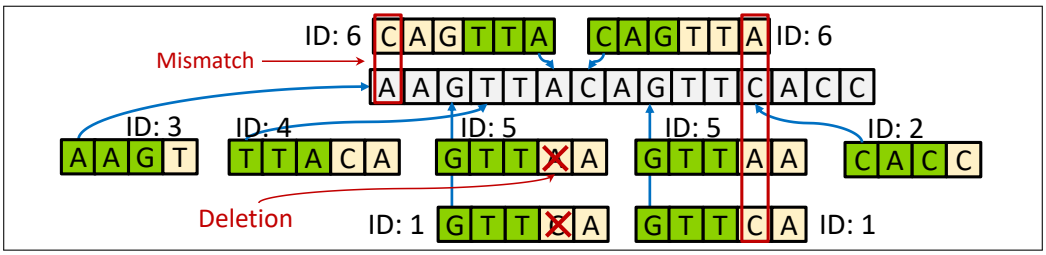

(a) An example reference genome and reads for illustration.




(b) **Exact-match strategy (EM)**. Reads 1-4 are exactly mapped to the reference. Reads 5 & 6 contain mismatch characters and therefore they cannot be aligned.




(c) **One-mismatch strategy (OM)**. Reads 5 & 6 can be successfully aligned to the reference. Each read can have multiple mismatch locations, which contribute to several aligned results per read.




(d) **Seed-and-extend strategy (SE)**. Reads are split into subsequences known as seeds. The seed (green) is EXACTLY aligned to reference to obtain all possible matching locations. Each location is then extended to identify the possible mismatches, deletions or insertions.

Figure 4.1: An illustration of different strategies. Usually, strategies in (b), (c) are based on an indexing algorithm such as FM-index to enable fast substring matching. Strategy in (d) is based on indexing with dynamic programming algorithm such as the Smith-Waterman to allow alignment with mismatches, insertions or deletions.

Despite the promising performance, FPGA-accelerated aligners are still

rarely adopted in genomics research and practice. This is because previous FPGA designs only report the speed-up, but rarely consider and investigate the corresponding alignment accuracy. For example, [13] proposes an alignment pipeline with an exact-match strategy, followed by a seed-and-extend strategy using the linear Smith-Waterman algorithm. In spite of a $72.2\times$ speed-up, the rationale behind the arrangement and selection of these strategies is not explained. Without an adequate explanation of the choice of strategies and the accuracy, genomics researchers have no clue if the accelerator is sufficiently accurate for realistic genomics practice.

In the chapter, we explore different **arrangements** of strategies by exploiting runtime reconfigurability of FPGA. For each arrangement, a few alignment strategies are executed in order, forming an **alignment pipeline**. Reads aligned by a strategy are reported immediately while unaligned reads are directed to subsequent strategy. Each strategy can be formed by one or two alignment **algorithms**, in which each algorithm is implemented as an individual FPGA configuration. Further information is explained in Section 4.1 and Table 4.1. Our main goal is to investigate the relationship between speed-up and accuracy of each alignment pipeline to provide guidance for genomics scientists.

Ultimately, we aim to achieve the contributions 2 and 3 stated in Chapter 1 through the following methods:

1. We extend the FM-index implementation of the exact-match strategy to perform seeding. We also develop a Smith-Waterman implementation with the affine-gap model, and complete biological information is considered in these implementations.

2. We explore different arrangements of alignment strategies (pipelines) for short read mapping based on the target platform Xilinx VU9P. For each alignment pipeline, the relationship between speed-up and accuracy is obtained using a real dataset.

3. We compare the performance between our optimal pipeline and state-of-the-art software Bowtie2, together with comparisons against some of the existing FPGA solutions.

The rest of the chapter is organized as follows: in Section 4.1 we introduce three different arrangements of alignment strategies, which are based on the workflows proposed by some of state-of-the-art software and previous FPGA designs; in Section 4.2 we present the implementations of the FM-index for the seeding stage and the Smith-Waterman algorithm with the affine-gap model for the extension stage; in Section 4.3 we present the performance and accuracy comparison of our proposed alignment pipelines versus other designs.

## 4.1 Selected Pipelines / Selected Arrangements of Alignment Strategies

The term alignment strategy generally refers to an alignment stage in state-of-the-art software such as Bowtie, Bowtie2. For example, by default, Bowtie2 first processes the reads using an exact-match strategy in the first stage. In the second stage, it uses one-mismatch strategy to process the reads that fail to align previously, *i.e.* a filtered search. Finally, it relies on a seed-and-extend strategy to process the remaining reads. As explained in Figure 4.1, each strategy is usually formed by one or two alignment algorithms such as FM-index and/or the Smith-Waterman algorithm.

According to Chapter 3, the overall alignment efficiency can be significantly increased by leveraging the runtime reconfigurability of an FPGA. It also provides better flexibility where genomics scientists can have greater control over the alignment parameters. Strategies can be added, re-ordered, or removed at runtime, making this architecture completely modular. Because of these benefits, we decide to adopt this approach to explore different alignment pipelines/different arrangements of alignment strategies.

Table 4.1: Selected alignment pipelines for speed and accuracy exploration. The arrow indicates the execution flow of the strategies. The rightmost four columns indicate the algorithms needed and the corresponding strategies realized are described in the square brackets underneath.

| No. | Selected Pipelines | Algo. I | Algo. II | Algo. III | Algo. IV |
|---|---|---|---|---|---|
| P1 | Exact-match (EM) → Seed-and-extend | FM-index [EM / Seed] | Smith-Waterman [Extend] | - | - |
| P2 | Seed-and-extend | FM-index [Seed] | Smith-Waterman [Extend] | - | - |
| P3 | Exact-match (EM) → 1-mismatch (OM) → Seed-and-extend | FM-index [EM] | Backtracking FM-index [OM] | FM-index [Seed] | Smith-Waterman [Extend] |

We use the reconfigurable architecture to configure the FPGA with different algorithms in turn, so as to apply alignment strategies in the required order. If two consecutive strategies require the same algorithm, the same FPGA configuration is reset and reused to avoid unnecessary reconfiguration. We also apply the filtered search approach in some selected pipelines to ensure accuracy. Only the reads that fail to be aligned by the current strategy will be handled by the subsequent one, which decreases the number of reads handled in each strategy along the pipeline.

In the rest of this chapter, we investigate the most promising pipelines based on the alignment workflows proposed by state-of-the-art software and previous FPGA design [8], in order to understand their implications on alignment speed and accuracy. Table 4.1 details the selected pipelines and the corresponding algorithms required, and Algorithm 6-8 explain the workflow of each pipeline. Note that we use Bowtie2 version 2.4.1 as the default version in the following discussion.

```
1 Function Execute(Data):
2     do in parallel
3         stream Data and/or Reference from the host
4         process the transferred Data
5         stream processed Data back to the host
6     end
```

**Algorithm 5:** The common function used in all pipelines.

```
1 //EM Strategy
2 load configuration FM-index
3 Execute(all the reads)
4 //Seeding for SE Strategy
5 reset configuration
6 Execute(unaligned reads R_na2 from line 3)
7 //Extend for SE Strategy
8 load configuration Smith-Waterman
9 Execute( [ R_na2, seed locations from line 6 ] )
```

**Algorithm 6:** P1 Workflow.

```
1 //Seeding for SE Strategy
2 load configuration FM-index
3 Execute(all the reads R)
4 //Extend for SE Strategy
5 load configuration Smith-Waterman
6 Execute( [ R, seed locations from line 3 ] )
```

**Algorithm 7:** P2 Workflow.

```
1 //EM Strategy
2 load configuration FM-index
3 Execute(all the reads)
4 //OM Strategy
5 load configuration backtracking FM-index
6 Execute(unaligned reads R_na3 from line 3)
7 //Seeding for SE Strategy
8 load configuration FM-index;
9 Execute(unaligned reads R_na4 from line 6)
10 //Extend for SE Strategy
11 load configuration Smith-Waterman;
12 Execute( [ R_na4, seed locations from line 9 ] )
```

**Algorithm 8:** P3 Workflow.

### 4.1.1 Pipeline `P1`

This is inspired by [13] where the proposed architecture achieves a theoretical speed-up of 293 × on Maxeler MAX3 platform. This extraordinary speed-up benefits from an alignment workflow composed of an exact-match strategy followed by a seed-and-extend strategy with the linear Smith-Waterman algorithm. We extend this workflow with the affine gap model, so that the pipeline is consistent with one of the provided alignment workflows in Bowtie2 when the option `-no-1mm-upfront` is specified.

Essentially, the FPGA is initially loaded with an FM-index implementation that performs exact-match alignment and also the seed location for the seed-and-extend strategy. The hardware first attempts to exactly align all the reads using Algorithm 3 in Section 3.2.1, completing the exact-match alignment strategy. The next strategy, which is based on seed-and-extension, begins with processing the reads that fail to align previously. By utilizing the same implementation, the unaligned reads are transferred back to the FPGA from the host. Reads transferred are partitioned into fixed-length seeds and are exactly matched to locate all possible matching locations on the FPGA. These locations are transferred to the host for temporary storage. Since we cannot predict the total size of the matching locations, storing these data using on-board memory is not a viable solution. The FPGA is reconfigured with the Smith-Waterman hardware that implements the affine-gap model. The locations and the reads are transferred back to the FPGA to complete approximate alignment, finishing the second strategy.

### 4.1.2 Pipeline `P2`

This arrangement is used by the initial version of Bowtie2 and it outperforms Bowtie in terms of accuracy but slackens off the alignment speed. This pipeline can report the reads that can be exactly and approximately mapped to the reference simultaneously.

Implementing exclusively the seed-and-extend strategy requires the same set of hardware used in `P1`. With runtime reconfiguration, the FM-index hardware is used to identify the possible locations of the seeds. Then the Smith-Waterman hardware is configured to perform extension.

### 4.1.3 Pipeline `P3`

According to [112], single nucleotide polymorphisms (SNPs) are the most abundant genetic variations in the human genome. Basically, SNP refers to a substitution of a single nucleotide at a specific position in the genome, which is present in a sufficiently large fraction of the population. With the aim to identify SNPs precisely, we use backtracking FM-index to identify reads with one-mismatch similar to the default workflow in Bowtie2. It also improves alignment speed thereafter as it filters out the reads with one-mismatch, and subsequently decreases the number of reads to process using the seed-and-extend strategy.

Similar to `P1`, this pipeline starts off with the exact-match strategy using the FM-index implementation. Then intermediate results, such as SA index for successful mapping and unaligned reads reported, are transferred back to the host. The FPGA is then reconfigured with the backtracking FM-index implementation to achieve the one-mismatch alignment strategy. Reads unmapped by the exact-match alignment are redirected to this implementation. Finally, the FM-index implementation for the exact-match strategy is reloaded again to carry out seeding, and the Smith-Waterman hardware is loaded onto FPGA afterward to perform extension. This final step is intrinsically identical to `P2`.

Figure 4.2 illustrates the runtime reconfigurable architecture for `P3` where each module consists of a homogeneous array of **computational modules** with the same alignment algorithm. Since runtime reconfiguration allows the decoupling of each algorithm, we can optimize the population of the under-

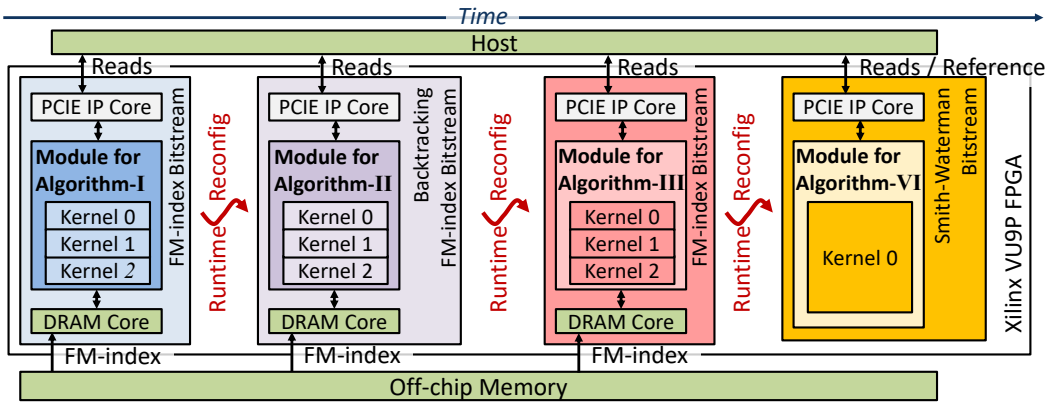



Figure 4.2: Multi-configuration alignment pipeline for P3. Each bitstream implements a specific algorithm composed of an array of computational modules.

lying computational modules based on one intended work, resulting in higher achievable parallelism.

### 4.1.4 Design Space and Performance

As mentioned, due to the decoupling of the alignment algorithms, the FPGA circuit is intended for one specific task at each runtime. This allows us to optimize the underlying computational modules based on one intended work, results in better module uniformity and higher achievable parallelism. Here we analyze the design space for the Smith-Waterman implementation with the affine gap model and provide back of envelope estimations for the performance. The key parameters used in this analysis are defined in Table 4.2.

One important advantage of the seed-and-extension strategy is the substantial reduction in the search space for the Smith-Waterman algorithm. The calculation of the scoring matrix $s_{i,j,\rightarrow}$, $s_{i,j,\downarrow}$, $V$ is now reduced to the characters in $R$ that is adjacent to the matching locations, and the read itself. This exhibits spatial locality and hence we exploit the CPU and its cache to aggregate the transfer for the reads and segments of $R$.

The Smith-Waterman circuit is composed of several matrix filling kernels. Each kernel consists of systolic arrays that calculate the score for $s_{i,j,\rightarrow}$, $s_{i,j,\downarrow}$, $V$. Since we don't need to access the onboard DRAM, the replications of the

Table 4.2: Key system parameters.

| | |
|---|---|
| $R_T$ | Total available resource on the target FPGA |
| $R_{PCI}$ | Resources for PCI-e controller |
| $r_s$ | Resources required by a module of Smith-Waterman algorithm |
| $K_s$ | Number of modules for Smith-Waterman algorithm |
| $t_{re}$, $t_{cs}$ | Time for runtime reconfiguration, and time for data communication for Smith-Waterman algorithm |
| $t_s$ | Time to align one read for Smith-Waterman algorithm |
| $N_s$ | Number of reads processed by Smith-Waterman algorithm |

Smith-Waterman modules are restricted by the FPGA resources only. This is given by:

$$K_s = \frac{R_T - R_{PCI}}{r_s} \tag{4.1}$$

The performance of the Smith-Waterman circuit can be modeled by (4.2). Similar to the FM-index implementation, data communication and reconfiguration overhead are negligible as the time for matrix computations is the dominant factor.

$$T = t_{re} + \max\left(t_{cs}, \frac{N_s t_s}{K_s}\right) \tag{4.2}$$

## 4.2 Module Designs and Optimization

This section discusses the FPGA implementations of the FM-index for the seeding stage, and of the Smith-Waterman algorithm with the affine gap model for the extension stage, targeting a single FPGA device. In particular, the FM-index implementation is extended from the exact-match strategy in the last chapter and is now capable of seeding and aligning reads exactly.

### 4.2.1 Seeding: FM-index Implementation

The FPGA configuration that implements the seed extraction and seed alignment using FM-index is an extension of the implementation of the exact-match strategy. Basically, the seeding process begins with streaming the reads from the host. By following a similar seeding strategy in Bowtie2, 26 nucleotides are extracted as seeds for every 15 nucleotides in each read, given that each read is 101$bp$. Then every seed is exactly aligned to the reference using FM-index with the pointers $Top$ and $Bottom$. The pointers $Top$ and $Bottom$ are updated based on the current character in the seed, and the correlated $i(x)$ and $c(\iota, x)$ values from the onboard memory. The final values for these pointers indicate the SA range for the seed. For each read, we maintain a priority list such that seeds with a smaller SA range are given higher priority in the extension stage. Figure 4.3 displays a simplified top-level diagram that implements seed extraction and seed alignment using FM-index.

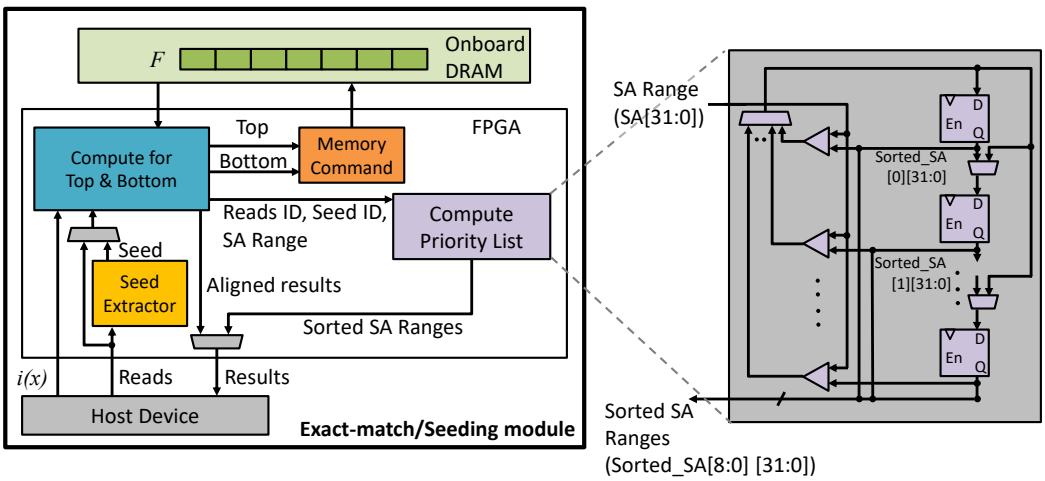



Figure 4.3: Simplified top-level diagram for exact-match/seeding module which contains the compute block for the priority list. For readability, some data and control paths are omitted.

### 2-bit Representation

Despite the DNA sequence produced by the NGS machines can be abstracted into {A, C, G, T, N}, statistically, only a small portion of reads contains N characters. Based on our observation on accession ERR194147 [113] generated by Illumina HiSeq 2000 sequencing machine, less than 4 % of the reads consist of N characters. With the aim to optimize the hardware efficiency, each nucleotide is represented in 2 bits in the FM-index circuit. Reads containing N characters are not transferred to FPGA and are instead processed by CPU for seed extraction and seed alignment. Given that only an inconsiderable amount of reads contains the N character, the seeding on the processor can be completely overlapped by the execution of FPGA. Note that seeds containing N characters are considered unaligned.

### Hardware Priority List

As shown in Figure 4.3, a set of comparators and shift registers are used to maintain the priority list. By default, 9 registers where each of them is 32-bit are cascaded to form the required shift register so as to cache all possible SA ranges for a read. Note that the number of registers can be increased by changing a parameter to accommodate longer reads.

When a seed is successfully aligned to the reference, the SA range, i.e. the difference between the final *Top* and *Bottom*, is obtained and compared against the existing SA ranges on the priority list. An insertion point can be found by comparing the SA range at each shift registers output simultaneously. The SA ranges larger than the current one are shifted in the shift registers so that seeds from smaller ranges receive a higher priority. Similar to the on-the-fly sorter in Chapter 3.2.3, the underlying logic is fully pipelined so that it can receive the aligned SA range from the compute block for each seed every cycle, even the comparison and the shifting operations can take a few cycles.

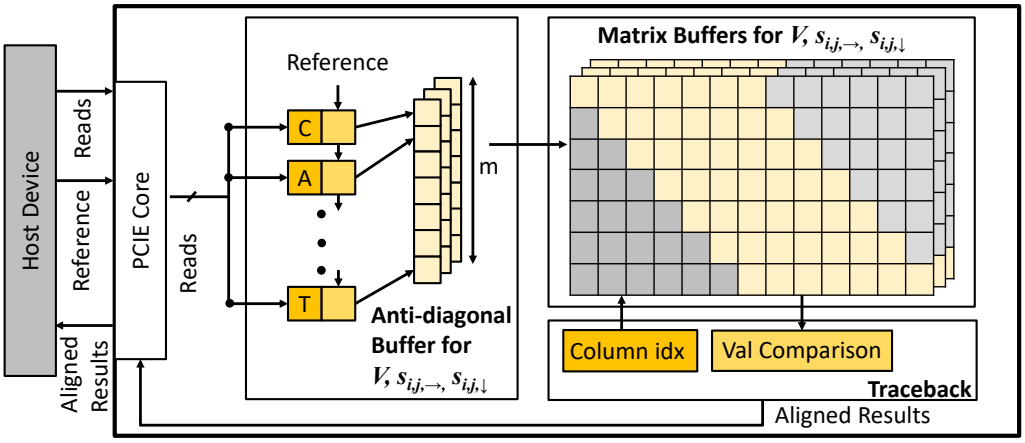



Figure 4.4: Simplified top-level diagram for the Smith-Waterman module with the affine-gap model. For readability, some data and control paths are omitted.

### 4.2.2 Extension: Smith-Waterman Implementation with Affine-Gap Model

Figure 4.4 displays the top-level architecture of the implementation of the Smith-Waterman module. It is composed of three major parts: systolic arrays that fill up $s_{i,j,\rightarrow}$ and $s_{i,j,\downarrow}$ and V, three matrix buffers, and a traceback unit. Our design draws inspiration from the work in [83], however, we rely on our FM-index circuit to locate the seed instead of using the software Bowtie2. Basically, accelerating both the seeding and extension steps provide a more superior speed-up, as the acceleration of only one step brings limited improvement based on Amdahl's law.

The systolic array consists of a pipeline of cells where each of them computes one score for the matrices. The cells are parallel-loaded with one base of the short read and its Phred quality, while the reference extracted in the proximity of the matching position is shifted through the array. This allows the calculation of the anti-diagonal of the matrix in parallel, which decreases the time complexity from $O(mn)$ to $O(m+n)$. The result of the current anti-diagonal is buffered with registers and BRAM for data interleaving, and it is reused in the computation of the next anti-diagonal.

In the second part, we use block memory to create matrix buffers in order to store up every score for $s_{i,j,\rightarrow}$ and $s_{i,j,\downarrow}$ and V during the anti-diagonal calculation. Despite the calculation does not require buffering of every value in the matrices, the complete storage reduces the efforts of the traceback unit. It can reconstruct the alignment path by traversing the matrix in reverse order, instead of recalculating the values and refilling the matrices.

#### Consideration for all NGS Data

The quality metrics and N characters are processed completely in the Smith-Waterman implementation. Compared to FM-index, the N characters and Phred quality are handled slightly differently. The reference and the reads are represented in 3 bits. The Phred quality, which is transferred alongside the reads, is loaded in the systolic array and used in the function $\sigma(P[i], R[j])$ for mismatch calculation. Despite the increase in the data transfer, the entire operation is highly computationally intensive and is therefore bound by the matrix computation. Note that based on Bowtie2, the $\sigma$ function is given by: $2 + floor(4 \times min(q, 40.0)/40)$.

## 4.3 Evaluations and Discussion

In this section, we evaluate the selected alignment pipelines and discuss their runtime and alignment accuracy. Then, we select the one that provides optimal runtime and accuracy and compares it to other FPGA-based alignment frameworks available.

### 4.3.1 Experimental Parameters

Similar to the last chapter, all the implementations are experimented on Xilinx Virtex UltraScale+ VU9P with three DIMMs of 16 GB onboard DRAM. The FPGA runs at 200 MHz while the host runs with Intel Xeon E5-2643 processor

at 3.4 GHz and 64 GB DDR4-2400 memory. Vivado 2017.4 is used for synthesis and implementation and the host is run with Centos 7.0. PCI-e 2.0 ×8 is used to transfer the data between the host and FPGA. As the alignment is bottleneck by either the onboard DRAM in the FM-index implementation or the matrix computations in the Smith-Waterman circuit, PCI-e 2.0 is already sufficient.

Our compressed FM-index is also constructed with the same parameters similar to the last chapter (burst size = 64 B, $d = 192$). The Smith-Waterman implementation, on the other hand, supports a read length of $m = 101$ and a reference length of $n = 150$. We need a larger segment from the reference to enable gapped alignment at the start and the end of a read. We also use the same sequence read dataset (`human_100_300M` in [109]) and reference genome (GRCh38) to evaluate the performance and accuracy. Finally, we use Bowtie2 2.4.1 in this evaluation. To ensure a fair comparison, we use the same normalized metric, base pairs aligned per second ($bps^{-1}$), given by Equation 2.6 in Section 2.2.4.

### 4.3.2 Alignment Runtime and Accuracy

Figure 4.5 presents the alignment time of the pipelines `P1`-`P3`. Also displayed is the result of the default runtime of Bowtie2 on Xeon Silver 4110@ 2.1 GHz using 16 threads and 192 GB RAM. We also show the result of Bowtie2 in default settings with the parameter `-no-1mm-upfront` specified. This option prevents Bowtie2 from searching for one-mismatch alignments before using the seed-and-extend strategy.

Compared to Bowtie2, pipeline `P3` provides the best acceleration with a maximum speed-up up to 5.94 × when 300 M reads are processed. `P1` also provides a decent acceleration with a maximum speed-up of 4.46 ×. The speed-up provided by `P2`, however, is only around 2.28 ×. Essentially, the extension step in `P2` considers any possible matching positions in all the reads, while

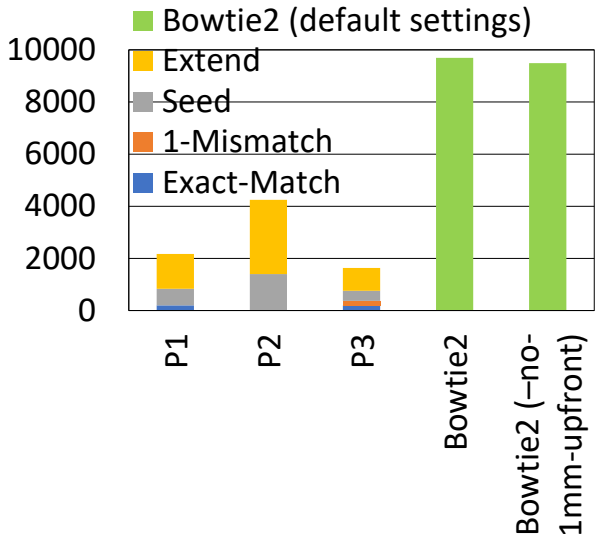



Figure 4.5: The alignment speed for the selected alignment pipelines and the default settings of Bowtie2 with and without the parameter `-no-1mm-upfront` specified.

`P1`, `P3` and Bowtie2 employ the exact-match and even one-mismatch strategies to reduce the search space. Therefore `P2` is less efficient compared to other pipelines, especially the extension step suffers from the most prolonged latency in all the selected pipelines.

Table 4.3 displays the comparison between the set-up and alignment accuracy of the different alignment pipelines and software. Again, accuracy is defined as the fraction of correct alignment and un-alignment from the proposed alignment pipeline among all of the reads. Correct alignment refers to reads aligned to the same locations as Bowtie2. Correct un-alignment means that if the reads are not aligned by Bowtie2, they should not be aligned by the proposed aligner also.

Since we have two separate Bowite2 runtime in different settings, the accuracy values are obtained with the following:

- The accuracy of `P1` is calculated based on the results of Bowtie2 with the parameter `-no-1mm-upfront` specified, as P1 is modeled upon the default settings of Bowtie2, but without searching for one-mismatch alignment.

- The accuracy of `P3` is obtained based on the default setting of Bowtie2, as the alignment pipeline of `P3` is basically identical to the default workflow of Bowtie2.

- The accuracy of `P2` is calculated based on the default settings of Bowtie2 with and without the parameter `-no-1mm-upfront` specified. Since it is unable to perform the seed-and-extend strategy exclusively in the current version of Bowtie2, we obtain two sets of accuracy numbers according to our Bowtie2 runtime and then we take the average.

Table 4.3: A comparison between set-up and alignment results of the selected alignment pipelines and the default settings of Bowtie2 with and without the parameter `-no-1mm-upfront` specified.

| | `P1` | `P2` | `P3` | **Bowtie2 (w/o –no-1mm-upfront)** | **Bowtie2 (w –no-1mm-upfront)** |
|---|---|---|---|---|---|
| **Device** | Xilinx Virtex UltraScale+ VU9P | | | Intel Silver 4110 (16 threads) | |
| **Frequency** | 200 MHz | | | 2.1 GHz | |
| **Lithography** | TSMC 16 nm | | | Intel 14 nm | |
| **Align time** | 2170 s | 4250 s | 1630 s | 9690 s | 9490 s |
| $bps^{-1}$ (million) | 13.9 | 7.14 | 18.6 | 3.13 | 3.19 |
| **Accuracy** | 93.9 % | 90.6 % | 95.2 % | – | – |

With the consideration of complete biological information in the read dataset, all the selected pipelines can achieve more than 90 % alignment accuracy. Notably, `P3` can achieve accuracy up to 95.2 %. This showcases that the identification of SNPs using a particular backtracking FM-index implementation can substantially improve the overall accuracy. It also improves the overall alignment speed where around 76 % of the reads are filtered out with the exact-match and one-mismatch strategies.

The discrepancy in accuracy, between our pipelines and Bowtie2 (100 % accuracy for the ground truth), is due to two reasons. First, Bowtie2 employs a heuristic to determine the mapping locations for the exact-match and one-

mismatch strategies, especially when the reads have multiple aligned results. Some of the reads that can be exactly aligned or aligned with one-mismatch by Bowtie2 are mapped to a different location in our proposed pipelines. Second, Bowtie2's search for alignments for a given read is randomized. That means that when Bowtie2 encounters a set of equally good alignment choices, it uses a pseudo-random number to choose. Therefore, our proposed pipelines align some of the reads to different locations in the seed-and-extend strategy. Furthermore, because of some minor differences in alignment parameters such as reseeding offset, extension attempts, and search limits, around 0.224 % of the reads are aligned in the proposed pipelines but not in Bowtie2, and vice versa.

Given that `P3` provides the best results in terms of alignment speed and accuracy, we select `P3` as the most optimal pipeline. It decreases the alignment time from around **2 hours 41 minutes** in software to around **27 minutes** on FPGA. In the next evaluation, we compare our optimal pipeline with some of the existing FPGA-accelerated aligners.

### 4.3.3 Performance Comparison with Existing Accelerators

Table 4.4 presents the performance comparison between the proposed optimal pipeline and previous accelerators from recent years. We select the aligners that are mostly based on FM-index, or the Smith-Waterman algorithm, or both for a fair comparison. Similar to the last chapter, we do not compare the alignment accuracy in this evaluation because different designs conduct their evaluations using different datasets.

The million $bps^{-1}$ values in Table 4.4 indicate that our aligner outperforms most of the previous work even we use runtime reconfiguration to deduce and run the optimal architecture. It also showcases the advantages of our architecture where the addition of computation logic for processing quality metric in the FM-index and the Smith-Waterman implementation does not

Table 4.4: Performance comparison with previous hardware accelerators that are based on similar algorithms.

| **Year** | **Work** | **Algorithm & Method** | **Platform** | **Device** | **Read Count** (Million) | **Read Length** ($bp$) | **Align Time** | **M**$bps^{-1}$ |
|---|---|---|---|---|---|---|---|---|
| 2013 | [13] | FM-index + Smith-Waterman | Maxeler MAX3 | Virtex-6 SX475T | 82 | 90 | 49.0 s | 151 |
| 2016 | [88][89] | Smith-Waterman | Alpha Data ADM-PCIE-7V3 | Virtex-7 VX690T-2 | 8 | 150 | 289 s | 4.1 |
| 2019 | [87] | Dynamic Programming | IBM Power8 S824L | Virtex-7 VX690T | 100 | 128 | 5112 s | 2.5 |
| 2019 | [83] | Bowtie2 + Smith-Waterman | Maxeler MAX5C | Virtex Ultrascale+ VU9P | 60 | 100 | 5830 s | 1.03 |
| 2020 | Ours | FM-index + Smith-Waterman | Maxeler MAX5C | Virtex Ultrascale+ VU9P | 300 | 101 | 1630 s | 18.6 |

affect alignment time.

The work [13] outperforms ours because of the following two reasons. First, their implementation of the dynamic programming is based on the linear Smith-Waterman algorithm, which involves only one matrix filling kernel. Our Smith-Waterman implementation is based on the affine-gap model which is composed of three matrix filling kernels and hence requires more compute cycles. Moreover, [13] only consists of two stages in the alignment pipeline while our alignment pipeline `P3` consists of three stages to improve the alignment accuracy.

### 4.3.4 Resource Consumption

Based on the proposed alignment pipeline `P3`, Table 4.5 indicates the corresponding resource consumption for each implementation on VU9P, with the inclusion of the PCI-e and memory controller. The resource consumption of the exact-match strategy is slightly different from the one in Chapter 3, as it is extended to support the seeding stage. Similar to the last chapter, with an adequate area remaining on the FPGA for the FM-index implementation, more modules can be populated on the FPGA if the number of DIMMs increases. The Smith-Waterman implementation, however, consumes less than 25 % of the resources because of the timing and fan-out issues. When the

Smith-Waterman module is replicated, a significant amount of logic spans across different dies on the target FPGA VU9P, which fails the timing. Future research will involve fine-tuning the Smith-Waterman implementation to minimize cross-die connection. Based on Equation 4.1, we can theoretically populate four Smith-Waterman modules on the target FPGA. This will bring further speed-up to the extension stage according to Equation 4.2 and theoretically contributes to a maximum 9 × speed-up of `P3` compared to Bowtie2.

Table 4.5: Area cost of the final design on VU9P. Percentage values are relative to the available resources on target FPGA.

| | **LUT** | **Register** | **BRAM** | **DSP** |
|---|---|---|---|---|
| **Exact-match/Seeding** | 127 442 (10.8%) | 239 262 (10.1%) | 1000 (18.9%) | 36 (0.53%) |
| **1-Mismatch** | 140 726 (11.9%) | 301 540 (12.6%) | 1224 (23.18%) | 9 (0.13%) |
| **Smith-Waterman** | 279 792 (23.7%) | 452 345 (19.1%) | 530 (10.0%) | 412 (6.02%) |

## 4.4 Summary

In this chapter, we present different arrangements of alignment strategies and evaluate their alignment speed and accuracy, by exploiting the runtime reconfigurability of FPGA for acceleration. The novel aspects of this architecture are:

1. By leveraging the runtime reconfigurability of FPGA, individual strategies can be combined and re-ordered to align reads to the reference. This pipeline enables evaluations of different arrangements of alignment strategies.

2. Different arrangements of strategies are explored and an optimal pipeline is obtained. It is composed of exact-match, one-mismatch, and seed-and-extend strategy in sequential order.

3. The exact-match strategy is extended to support the seeding stage, and the Smith-Waterman algorithm with the affine-gap model is implemented.

4. Complete biological information including quality metric and ambiguous characters are considered along the optimal alignment pipeline. With alignment results from Bowtie2 acting as the ground truth, our design can achieve an accuracy of 95.2%.

In the following chapter, we introduce an automatic merger that combines multiple versions of a design project into a single hardware implementation, with a case study on Binomial Filters and one of the proposed FPGA aligners.

# Chapter 5

# Automated Design Analyzer and Merger

FPGA accelerators have shown to be promising candidates to improve system performance and power efficiency for more than two decades [114, 115]. However, the low productivity in developing FPGA-based applications and the design portability compared to software development remains a huge obstacle that hinders widespread utilization of FPGA devices in mainstream systems [116].

One of the major challenges when designing applications on FPGA devices is the lack of efficient implementation, optimization and debugging facilities [117]. In particular, compiling a hardware design using standard design tools could involve a tremendous amount of time. This long compilation time limits the amount of implement/optimize-debug-edit cycles [118] per day and, as a consequence, hinders the productivity of the designers.

During the optimization process, it is usual to have multiple versions [119] of an FPGA design project being experimented on actual and specific hardware to test the functional correctness or to evaluate their accuracy. In particular, **there exist many real-life applications that perform optimization by fine-tuning each version of the design**, where the derivation of each ver-

sion can be independent of each other and the results from one version are not required for deriving the other versions. For instance, Aubury and Luk [120] propose the use of binomial filters to implement and approximate Gaussian filtering on FPGA. The depth of the binomial filter structure can be adjusted in each version to determine the accuracy and the frequency response. Also, Targett *et al.* [121] carry out a precision and resolution exploration for shallow water equations for climate modeling. This study includes reducing the bitwidth of mantissa length of variables in each version of design to balance the tradeoff between precision and accuracy. These fine-tuning activities can improve the resulting design significantly, but can be time-consuming due to the repeated and prolonged process of placing and routing for each of the design versions.

To address the above design optimization challenge, we propose the use of an automatic merger that combines multiple versions of a design project into a single hardware implementation. The proposed merger can identify common computational kernels between versions, perform the necessary merging and generate a final hardware design in linear time. Instead of placing and routing each individual version every time separately as shown in Figure 5.1, the developer can implement the generated hardware once and hence improve optimization productivity. We note that this approach is still useful for the scenario where the derivation of each version is dependent on a former one because developers can sometimes predict the possible parameters for the succeeding versions.

Furthermore, based on the statistics from the Proceeding of International Conference on Field Programmable Technology (ICFPT) 2015 and 2016, 75% of the full papers in the application sections utilize less than half of the resources on FPGA. In other words, **there remains adequate area on chip for insertion of extra logic with the proposed merger, especially when there are only minor discrepancies between each version of the design.** We collect the statistic from ICFPT instead of other conferences because

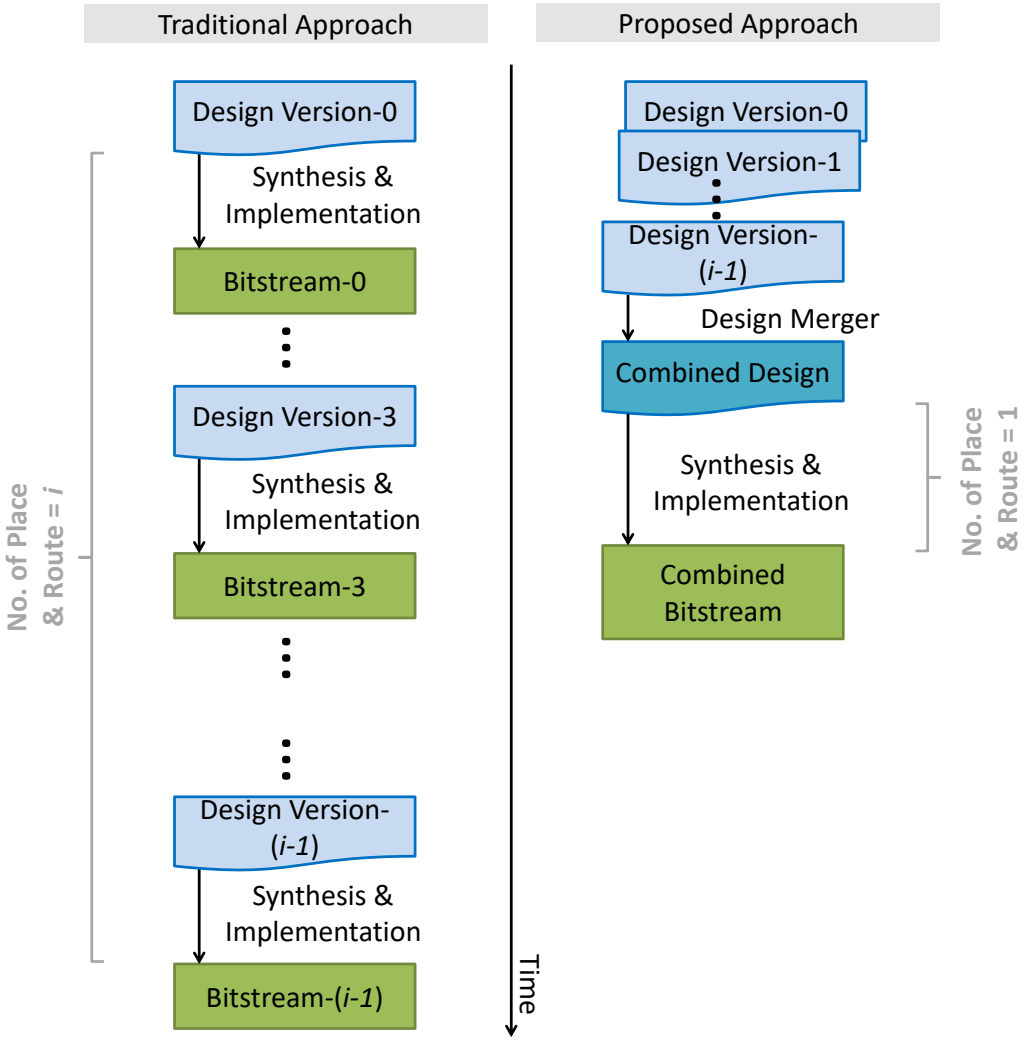



Figure 5.1: Comparison between the traditional approach and the proposed approach during the optimization phase of an FPGA design. Traditional optimization flow (left) requires placing and routing the design $i$ times, while the proposed approach (right) reduces the number of compilations but requires the use of a design merger.

there are more application-based contributions in this conference. Finally, **by relaxing the timing constraints, the proposed merger enables designers to focus on checking functional correctness in hardware which is faster and more accurate than software simulation**.

This chapter presents DAM, an automated **D**esign **A**nalysis and **M**erging approach to improve the design optimization process. Given $i$ versions of a design, this approach first parses each of them and generates the dataflow graphs based on their respective datapath. Then a maximum subgraph algorithm for dataflow graphs is applied to determine the common computational kernels among them with linear time complexity. Common signals with different bitwidths across the versions are also analyzed and merged if possible in order to further minimize resource consumption. Finally, the users can select a particular version of design by providing appropriate control signals to the generated hardware implementation. The proposed approach can also be applied to merge unrelated designs targeting a large FPGA.

Since the users do not need to follow the low-level details of the generated hardware implementation, the proposed approach can be considered as an ***overlay*** [118] where a virtual programmable intermediate architecture is overlaid on top of the physical fabric as a way to address the productivity and portability challenge.

Ultimately, we aim to achieve the contribution 4 stated in Chapter 1 through the following methods:

1. We propose a novel approximate maximum common subgraph detection algorithm for dataflow graphs with a linear time complexity that maximizes the sharing of resources for merging of different design versions.

2. We develop a prototype tool implementing a common subgraph detection algorithm for dataflow graphs derived from Verilog designs, which in addition generates the appropriate control circuits to enable the selection of each design version at runtime.

3. We provide a comprehensive analysis of compilation time versus degree of similarity to identify the optimized user parameters for the proposed approach.

The rest of the chapter is organized as follows: in Section 5.1 we present the details of the proposed framework; in Section 5.2 we describe the prototype tool and evaluate the performance of DAM; in Section 5.3 we provide a comprehensive analysis of compilation time versus degree of similarity; in Section 5.4 we describe some previous work similar to the proposed framework.

## 5.1 The DAM Framework

This section provides a comprehensive overview of DAM. Figure 5.2 illustrates the complete workflow of the design merger. To begin with, we consider the dataflow graphs for multiple versions of a design. As there are only minor discrepancies between each version, every dataflow graph will look remarkably similar. To identify the common subgraph between versions, a maximum common subgraph algorithm for dataflow graphs is launched and the corresponding nodes are merged, including the nodes of the common signal that can be different in bitwidth across different versions of design. Then the combined dataflow graph is directed to the compiler to generate a final hardware design.

The proposed approach has two novel aspects. First, DAM supports merging of multiple design versions in linear time based on an approximate maximum common subgraph algorithm for dataflow graphs. Second, it covers merging of common variables that have different bitwidths across versions. In the following subsections, we describe each of the modules within the merger and their interactions in detail.

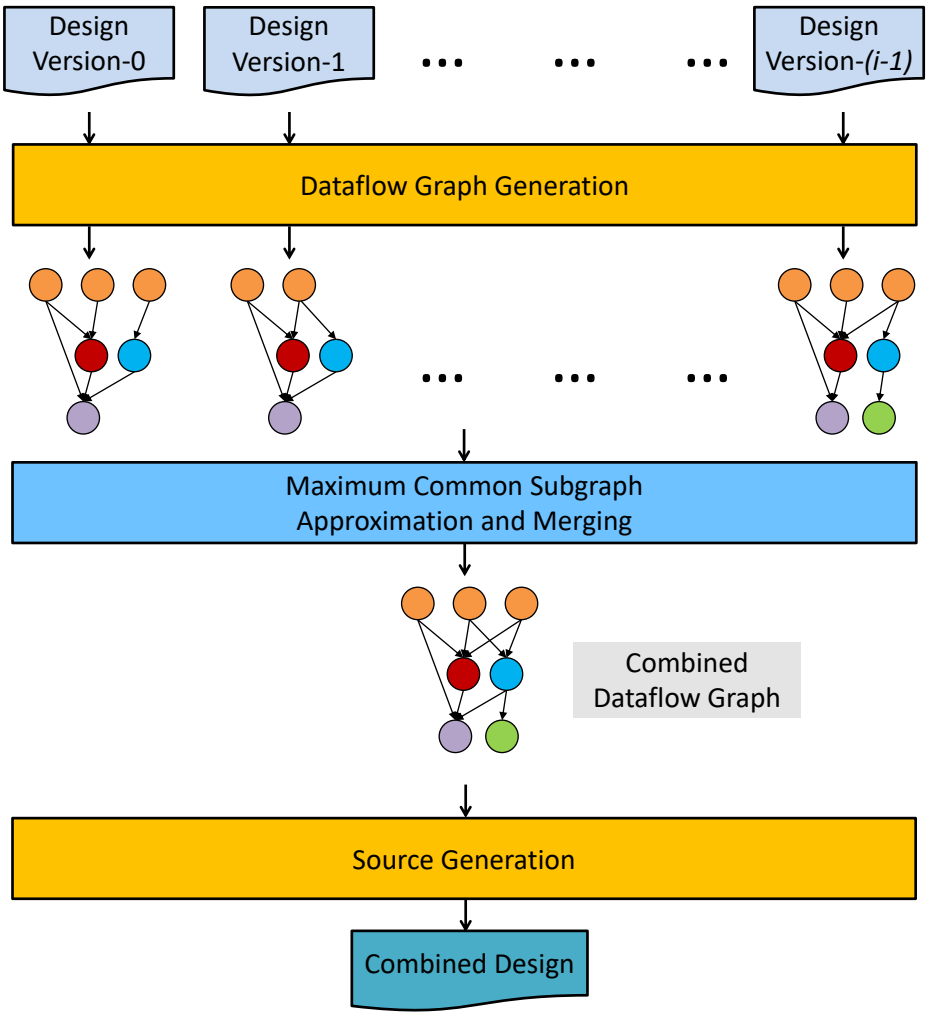



Figure 5.2: The workflow of the proposed DAM approach.

### 5.1.1 Dataflow Graph

A dataflow graph is a directed graph that represents the datapath of a circuit, where the nodes represent the basic operations and variables of a design while the edges between them represent specific paths that data elements follow [122]. Every hardware circuit can be translated into a dataflow graph and vice versa, since every node in the graph corresponds to a hardware unit that can be allocated on the chip surface and every edge represents a wire between two units.

In the proposed approach, a dataflow graph is first abstracted from each version $v$, where $v = 0, 1, ..., i-1$ of the hardware design with a source-to-source compiler. Then, in order to recognize the common computational kernels, a maximum subgraph algorithm is subsequently applied between every version of dataflow graphs $G_0, G_1, ...., G_{i-1}$ to identify the maximum amount of connected hardware elements that can be merged and shared.

### 5.1.2 Maximum Common Subgraph Algorithm for Dataflow Graph

Essentially, precise detection of maximum common subgraph (MCS) in random and arbitrary graphs is an NP-complete problem. Existing algorithms such as McGregor or Durand-Pasari suffer from prolonged execution latency because of their exponential time complexity [123]. The matching of multiple dataflow graphs, specifically their vertices, is a polynomial-time problem, as it can be modeled as a maximum weight bipartite matching. However, the matching of the edges to minimize the interconnection area is an NP-complete problem, since their matching must depend on the mapping of the adjacent vertices [124]. Consequently, precise algorithms are inappropriate for the proposed design merger.

### Approximate Algorithm for MCS Detection

It is noticed that the dataflow graph extracted from hardware circuits carries certain properties that can aid in the quick search for MCS. In general, nodes are connected by a few edges since most operators consist of only one or two parents and one output, and the majority of the nodes are normally labels such as signal or port names. As a result, the dataflow graph extracted is so sparse that an approximate algorithm such as [125] (time complexity: $O(n)$, where $n$ is the number of nodes) can be used to obtain a set of MCS with decent quality.

To approximate the MCS between two graphs $\boldsymbol{G_a}$ and $\boldsymbol{G_b}$, a mapping $M_{ab}$ is constructed from the vertices $v_a \in V_a$ of graph $\boldsymbol{G_a}$ onto the equivalent vertices $v_b \in V_b$ of graph $\boldsymbol{G_b}$. In [125], Rutgers *et al.* present a greedy algorithm which uses best-first search to traverse the graph $\boldsymbol{G_a}$ and $\boldsymbol{G_b}$. In each round of search, a vertex $v_a$ is heuristically chosen from $\boldsymbol{G_a}$ so as to find a mapping to a vertex $v_b$ of $\boldsymbol{G_b}$. For every possible $v_b$, the best candidate to choose from is determined by the following heuristic. To begin with, vertices in $V_a$ with fewer possible mapping candidates in $V_b$ are handled first, as the probability of selecting an incorrect vertex decreases with a fewer number of candidates. After a vertex $v_a$ is chosen from $V_a$, the selection of the corresponding $v_b$ depends on the similarities of $v_a$ and $v_b$ neighbors. Lastly, when a round of search completes, the vertex $v_a$ is finished and will not be selected again regardless of the search result.

To initiate the above MCS algorithm, the set of inputs $I_a$ and outputs $O_a$ of $\boldsymbol{G_a}$ are matched against the set of inputs $I_b$ and outputs $I_b$ of $\boldsymbol{G_b}$ respectively, and this constructs the initial common vertices in $M_{ab}$. Since every version of the same design is highly similar during the design optimization process, the input/output interface of each version must share some common signals such as the clock or reset input. After initialization, the above heuristic, denoted by $Rutgers(\boldsymbol{G_a}, \boldsymbol{G_b}, M_{ab})$, is subsequently launched until all the vertices in $V_a$

are exhausted so as to return the MCS $M_{ab}$. For further information about the approximation algorithm, please refer to [125].

Obviously, there are several conditions to check before two nodes are identified as common. First, both nodes need to implement the same operation, and they also have to operate on the same data type. Furthermore, associative operations such as $(a+b)+c$ and $a+(b+c)$ must be extracted before performing MCS detection, and commutative operations such as $a+b$ and $b+a$ must also be recognized as the same operation to minimize the area cost of the final implementation.

### MCS Algorithm for Multiple Graphs

Since [125] can only determine the set of MCS between two dataflow graphs, the algorithm has to be launched iteratively until a final set of MCS for every version is obtained. The set of notations adopted in this chapter is given by:

- $i$ is the total number of versions for a given design;
- $G_p$ is the dataflow graph for each version, where $p = \{0, 1, ..., i-1\}$;
- $C_q$ is the set of MCS between every $G_p$, where $q = \{0, 1, ..., i-2\}$;
- $\boldsymbol{C} = C_{i-2}$ is the set of MCS between every version of dataflow graphs;
- $\overline{\boldsymbol{C}}$ is the negation of $\boldsymbol{C}$ which contains all the uncommon subgraphs;
- $find\_MCS((G_0, G_1, ..., C_{j-3}), G_{j-1})$ refers to the algorithm that identifies the set $C_q$, where $2 < j \leq i$.

To identify the set of MCS between every version of dataflow graphs, an initial set of MCS $C_0$ is obtained by comparing $G_0$ and $G_1$. This newly calculated $C_0$, together with $G_0$ and $G_1$, are matched against $G_2$ to compute $C_1$. This process repeats $i-2$ times until the final $C_{i-2}$ is obtained. Note that the set $C_{i-2}$, which is equivalent to $\boldsymbol{C}$, contains every set of MCS across all $i$

versions of design. Other nodes that are not in any of the MCS fall into the set $\overline{\boldsymbol{C}}$.

```
Data: G_0, G_1, ..., C_{j-3}, G_{j-1}, where 2 < j ≤ i
Result: C_{j-2}
G_a = {G_0, G_1, ..., G_{j-2}}
G_b = {G_{j-1}}
M_ab = {}
for g in G_A do
    M_ab = M_ab ∪ MATCH_IO( g, G_b )
end
M_ab = Rutgers(G_a, G_b, M_ab)
C_{j-2} = C_{j-3} ∪ M_ab
```

**Algorithm 9:** Pseudocode of a single iteration of MCS approximation for multiple dataflow graphs. Assume that $C_0$ is already computed for consistent input data format.

A simple illustration of each iteration of the merging process is displayed in Algorithm 9. Each iteration is denoted by $find\_MCS((G_0, G_1, ..., C_{j-3}), G_{j-1}))$. In each iteration, $\boldsymbol{G_b}$ is initialized with the graph to be matched against, while $\boldsymbol{G_a}$ is composed of multiple dataflow graphs across versions, which can be conceptually considered as a single dataflow graph with numerous unconnected subgraphs. After that, the common input and output ports are mapped and inserted into $M_{ab}$, and $Rutgers(\boldsymbol{G_a}, \boldsymbol{G_b}, M_{ab})$ is then executed to compute a partial MCS. Finally, the information about the current MCS and the MCS from the previous iteration are joined to obtain a complete MCS. This final step is crucial because only one vertex in $\boldsymbol{G_a}$ can be mapped to a candidate in $\boldsymbol{G_b}$ based on Rutgers *et al.* Yet in reality multiple vertices can be matched because $\boldsymbol{G_a}$ is composed of dataflow graphs from every version. The MCS formulated in the previous version provides the information about the rest of the mapped vertices, and hence the union of $C_{j-3}$ and $M_{ab}$ contributes to a complete search result.

Since the number of versions $i$ is relatively small, the overall time complexity is given by:

$$
\begin{aligned}
& O(find\_MCS(G_0, G_1)) + O(find\_MCS((G_0, G_1, C_0), G_2)) + \\
& ... + O(find\_MCS((G_0, ..., G_{i-2}, C_{i-3}), G_{i-1})) \\
= & O(n) + O(2n + n) + ... + O(i \times n + n) \\
= & O(i^2 n) \\
= & \boldsymbol{O(n)},
\end{aligned}
$$

which means such algorithm is acceptable for the proposed design merger because of its linear time complexity.

#### Final Dataflow Graph Generation

In order to generate the final hardware which is logically identical to the originals, every MCS in $\boldsymbol{C}$ is first combined to generate a merged dataflow graph. The inputs and outputs of the merged graph are reconnected to the nodes in $\overline{\boldsymbol{C}}$ as well.

Essentially, the inputs to the MCS in $\boldsymbol{C}$ are multiplexed and the `sel` signal is fed to the output interface. To activate a particular version of the original design, an associated value is asserted at `sel` so that a correct signal from $\overline{\boldsymbol{C}}$ can be directed to the merged hardware. The outputs of the merged node, on the other hand, have to be connected back to the nodes of the versions that originally use the results. Figure 5.3 displays an example that explains the process of multiplexing.

### 5.1.3 Analysis and Merging of Common Signals / Variables with Different Bitwidths

As our goal is to minimize the resource consumption for the combined implementation, we are also interested in merging the common signals even though they are different in bitwidth across various versions of a design.

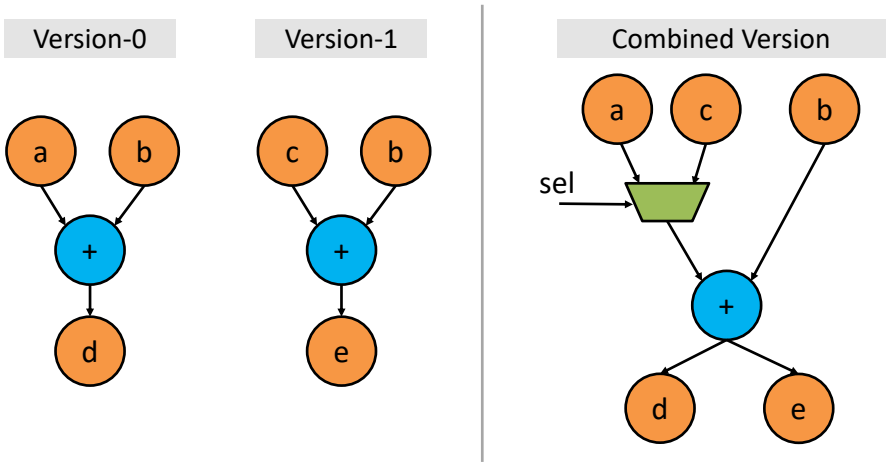



Figure 5.3: Example of multiplexing when dataflow graphs of two separate versions are combined and merged.

In the above MCS search, the common signals mentioned are considered to be non-identical because of their discrepancies in bitwidth. In order to merge these signals, the maximum bitwidth of every common signal is first obtained and the value is used to update every node that carries the same variable. After that, the same set of MCS algorithms is applied on $\overline{\boldsymbol{C}}$, which identifies a new group of MCS $\boldsymbol{C'}$ composed only of the newly-formed common signals. To provide a clear explanation, another set of notations is adopted in this subsection and they are defined as:

- $\boldsymbol{C'}$ is the set of newly obtained MCS which is composed of the common signals with different bitwidths;
- $\overline{\boldsymbol{C'}} = \overline{\boldsymbol{C}} - \boldsymbol{C'}$ contains all the graphs in various versions that cannot be combined or merged with any of the methods proposed.

### Assignment Nodes

Since every common signal in $\boldsymbol{C'}$ is unique originally, extra hardware node is inserted in the dataflow graph during the merging phase of $\boldsymbol{C'}$ to ensure correctness. This includes appending multiplexers, partially-selecting and sign/zero-extending the low-level bit when the signal appears as an assignment node. A

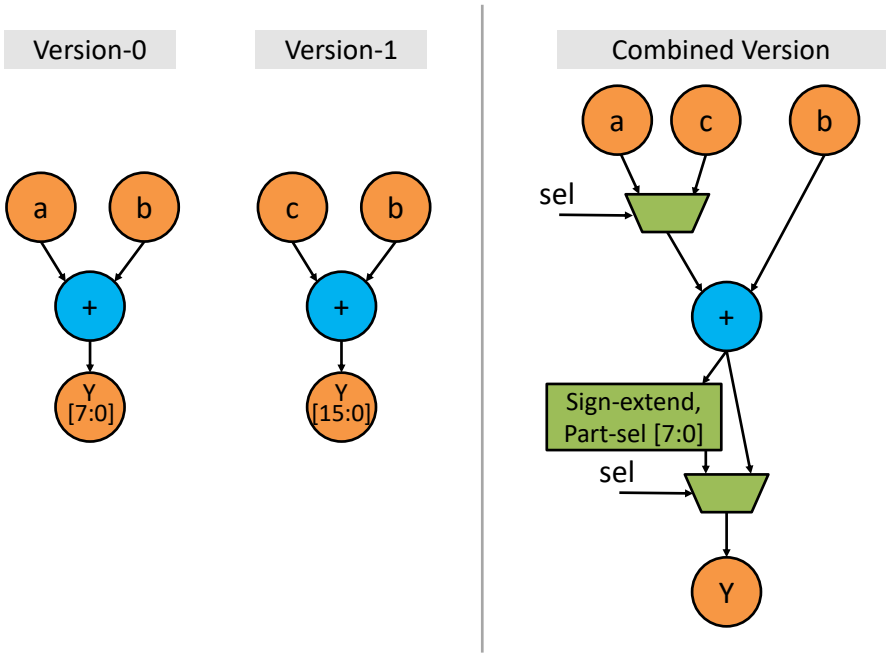



Figure 5.4: Example of appending multiplexers, partially-selecting and sign-extending the low-level bit when the common signal with multi-bitwidth is merged.

signal or a variable is assigned when it is either attached to the output of an operator, or directly connected to another signal in the dataflow graph.

Figure 5.4 shows an example of the above process when the output signals are attached to an addition operator in two separate versions. Initially, the common signal `Y` is of width 8-bit and 16-bit in version-0 and version-1 respectively. Then, the operator is merged and its output is partially-selected and sign-extended. This enables the 16-bit signal to imitate an 8-bit signal and contributes to the same computational result.

As illustrated in the above example, different number of bits should be selected and different values should be appended in regard to the signal type and the operators attached. Normally, sign extension is applied when the signal adopts signed number representation while for unsigned number representation zero extension would suffice.

### Comparison Operator

Furthermore, for every common signal that is connected to a comparison operator (e.g. $== < \leq > \geq$), partially-selecting the low-level bit is required. This is due to the fact that comparison is based on the left-to-right evaluation, and the sign/zero-extension process performed above will incur an incorrect comparison result if left unattended.

### Connection to the MCS in $\boldsymbol{C}$

Depending on the original structure of the dataflow graph, the inputs or outputs of each MCS in $\boldsymbol{C'}$ can be connected to the previously formed MCS in $\boldsymbol{C}$, or simply connected to the nodes in $\overline{\boldsymbol{C'}}$. The following description summarizes all possible combinations and provides a detailed explanation for each scenario.

1. *$\boldsymbol{C'}$ and $\boldsymbol{C}$ Unconnected* — In this scenario, every input and output of an MCS in $\boldsymbol{C'}$ are connected to the uncommon subgraphs in set $\overline{\boldsymbol{C'}}$. Similar to the multiplexing mechanism as shown in Figure 5.3, the inputs are multiplexed and the `sel` signal is fed to the control interface. Also, the outputs of the merged graph must be connected back to the nodes in the uncommon graphs that use the calculated results.

2. *Outputs of $\boldsymbol{C'}$ connected to Inputs of $\boldsymbol{C}$* — This is the case where the outputs of an MCS in $\boldsymbol{C'}$ are connected to any input nodes of an MCS in $\boldsymbol{C}$. To link both MCS together, the outputs are first partially-selected and sign/zero-extended, which is similar to the example in Figure 5.4. The multiplexers inserted in Section 5.1.2 are also slightly modified. The inputs of the original multiplexer in $\boldsymbol{C}$ are disconnected so that the sign/zero-extended outputs and the unmodified outputs can connect to them.

3. *Inputs of $\boldsymbol{C'}$ connected to Outputs of $\boldsymbol{C}$* — In this case, the outputs of an MCS in $\boldsymbol{C}$ can be connected to the inputs of an MCS in $\boldsymbol{C'}$ di-

rectly, without the need to introduce extra hardware. This is because the partial-selection and bit-extension process during assignment can always guarantee that a common signal will carry a correct value.

Currently, the merging of common signals with multi-bitwidth always takes place regardless of hardware cost, which may be less desirable for low-cost operations such as addition. In the future, we can extend the merging heuristic by considering multiplexing versus operator savings to further minimize the final resource consumption.

### 5.1.4 Optional Timing Optimization

Based on [122], the throughput of a hardware mainly depends on the number of data items that the design can process in one cycle, and also the maximum clock frequency that the design can support. Therefore, the proposed design merger provides an optional mechanism for users to perform certain re-pipelining if the dataflow graph is direct acyclic.

It is often hard to fulfill timing constraints when an output signal is connected to many hardware nodes. Since it is difficult for the synthesis and implementation tool to place the hardware nodes in close proximity, the resulting wire length will consequently increase. To address this challenge, the proposed approach can insert registers in a tree-like fashion such that each register only consists of a limited amount of outputs if the timing optimization mechanism is activated by the users.

Algorithm 10 shows the simplified timing optimization process where the final dataflow graph is parsed two times. The first round determines the high fan-out nodes and calculates the number of registers needed to generate the tree-like structure. To ensure the correctness of the results after the insertion of registers, the increase in pipeline levels is also recorded for each node. After the first parsing, the second round begins with traversing and adding each vertex into the final graph. When a high-fan out node is encountered, a tree-like

register structure is added to the final graph. Finally, each of the children are checked to ensure correct pipelining by inserting extra registers when necessary.

### 5.1.5 Final Implementation Generation

After detecting and approximating the MCS and merging the common nodes with the methods proposed, the final dataflow graph, which is formed by $\boldsymbol{C}$, $\boldsymbol{C'}$ and $\overline{\boldsymbol{C'}}$, can be supplied to the source-to-source compiler to generate the final implementation.

Usually, the compiler can produce the final hardware that is in the same language as the original design. However, depending on the needs of the designers, the compiler can be extended to produce the corresponding source code in another programming language such as Chisel [126] or Verilog to promote productivity.

## 5.2 Prototype Tool and Benchmarks

### 5.2.1 Prototype tool based on Pyverilog

With the objective to improve designers' productivity during the optimization process of FPGA implementations, the key goal of DAM is to merge every version of a design automatically while minimizing resource consumption. To demonstrate the feasibility and viability of DAM, we prototype the proposed approach with Pyverilog [127] to support the functionality mentioned in Section 5.1 as a proof of concept.

Pyverilog is an open-source toolkit that provides register transfer level design analysis and code generation of Verilog HDL. Written in the Python programming language, Pyverilog incorporates multiple libraries such as parser, dataflow analyzer and Verilog code generator that are useful to realize the proposed design merger. In our prototype, we use the given parser and dataflow analyzer to generate the dataflow graph for each version of a Verilog-based

```
Data: Final dataflow graph C_f, output connections limit l
Result: Timing optimized dataflow graph C_tf
1  //1st parsing to calculate registers needed
2  for vertex g in C_f do
3      if g.child_cnt > l then
4          //calculate the extra registers needed for the
5          //tree-like structure
6          g.tree_level = | ln(g.child_cnt) / ln(2) | - 1
7          for child h of g do
8              //Assume h.reg_level is initialized as 0
9              h.reg_level = max(g.reg_level + g.tree_level - 1,
10                               h.reg_level)
11         end
12     else
13         g.tree_level = 0
14         for child h of g do
15             h.reg_level = max(g.reg_level , h.reg_level)
16         end
17     end
18 end
19 //2nd parsing to add registers
20 for vertex g in C_f do
21     C_tf.addVertex(g)
22     if g.tree_level > 0 then
23         //add registers based on tree-like structure
24         C_tf.addRegTree(g, g.tree_level)
25     end
26     for child h of g do
27         reg_cnt = h.reg_level - g.reg_level − g.tree_level
28         //add registers to ensure correct pipelining
29         C_tf.addReg(g, h, reg_cnt)
30     end
31 end
```

**Algorithm 10:** Pseudocode of the timing optimization for the final dataflow graph. For simplicity the function contents for $addVertex()$, $addRegTree()$ and $addReg()$ are omitted.

design. Then, we approximate and combine the MCS iteratively in linear time using the algorithm presented by Rutgers *et al.* [125]. As the dataflow graphs between versions are based on the same design project and they look remarkably similar, Pyverilog is extended to recognize the common signals based on their literal names for fast prototyping. Therefore, the MCS obtained is an associative reduction where the resulting graph is independent of the input graph permutation. To further optimize the tool efficiency, we also perform the search and the necessary merging of the common signals that are different in bitwidth during the above iterative MCS detection. Optional timing optimization is performed by analyzing the number of fan-out of any outputs. The final dataflow graph is processed by the Verilog code generator to produce a final, Verilog-based hardware description.

Moreover, the decision to implement the proposed approach based on Verilog is mainly a consideration for design productivity and tool portability. Verilog is one of the most-used design languages to describe a digital circuit at the register transfer level for FPGA-based implementations. In addition, Verilog and VHDL are usually used as an intermediate representation for open-source or vendor EDA tools in modern high-level synthesis and next-generation HDL research [127]. Finally, we note that merging for block RAM is also supported in our prototype when the design implements inferred block RAM.

With Pyverilog extended to support the proposed approach, we run the design merger on HP EliteDesk 800 G2 Tower PC with Intel i7-6700 3.40GHz CPU and 32GB RAM, and the merged hardware is synthesized and implemented onto Xilinx Artix-7 AC701 Evaluation Platform using Vivado 2016.3 edition to recognize the overall performance and limitations.

### 5.2.2 Benchmarks from VTR

#### Experimental Setup

We select several parameterizable Verilog designs from the VTR Benchmarks [15, 16] and automatically combine them with the prototype merger in order to understand its implications in terms of real-life applications. These applications include bgm, LU8PEEng and array of diffeq1 which provide macros or parameters for users to explore different hardware structures and to offer multiple versions of a single design.

Table 5.1 illustrates the configuration details for these applications. In diffeq1, each version is obtained by adjusting the bitwidth of all signals, while for bgm and LU8PEEng the macros `BITS` and `PRECISION` are altered respectively so that different precision can be used to calculate the final results. As lowering the precision and changing the corresponding macros eliminate certain parts of the original circuit, the resulting dataflow graphs vary across different versions. Additionally, the adjustment of the macros changes the width of several signals, and hence creating common signals with different bitwidths for merging.

Finally, the generated hardware and original hardware are synthesized and implemented individually using Vivado with the default settings, and data about the area cost and compilation time are collected subsequently. Also, the maximum frequency for each implemented hardware is obtained by specifying different timing values in the constraint file and compiling separately until the tool fails to meet the timing constraint. Finally, optional timing optimization is not activated for these benchmarks.

#### Experimental Results

For each application, the area cost and the maximum frequency for every version, including the combined ones, are displayed in Table 5.1. The percentage values are relative to available resources of the targeted FPGA device.

Table 5.1: Resource consumption and maximum frequency of the generated hardware versus the originals for different applications.

| Application | Version | Difference | LUTs | | Registers | | BRAMs | DSPs | Max. Freq. |
|---|---|---|---|---|---|---|---|---|---|
| **diffeq1 Array (Array Size=64)** | 0 | for all signals: bitwidth = 8 | 5928 | 4.40 % | 5540 | 2.06 % | 0 | 0 | 125 MHz |
| | 1 | for all signals: bitwidth = 16 | 5638 | 4.19 % | 8256 | 3.07 % | 0 | 256 | 78.74 MHz |
| | 2 | for all signals: bitwidth = 24 | 17 497 | 13.0 % | 15 827 | 5.88 % | 0 | 384 | 60.24 MHz |
| | 3 | for all signals: bitwidth = 32 | 20 790 | 15.4 % | 20 608 | 7.65 % | 0 | 576 | 58.82 MHz |
| | **Merged** | – | 22 399 | 16.6 % | 20 544 | 7.63 % | 0 | 576 | 53.19 MHz |
| **bgm** | 0 | Macro: `BITS` = 8 | 7318 | 5.44 % | 2665 | 0.990 % | 0 | 22 | 78.13 MHz |
| | 1 | Macro: `BITS` = 16 | 7356 | 5.47 % | 3401 | 1.26 % | 0 | 22 | 78.13 MHz |
| | 2 | Macro: `BITS` = 32 | 11 518 | 8.56 % | 6030 | 2.24 % | 0 | 22 | 78.13 MHz |
| | **Merged** | – | 12 728 | 9.46 % | 8276 | 3.07 % | 0 | 22 | 70.42 MHz |
| **LU8PEEng** | 0 | Macro: `PRECISION` = 8 | 8369 | 6.22 % | 3903 | 1.45 % | 28 | 16 | 19.46 MHz |
| | 1 | Macro: `PRECISION` = 16 | 9946 | 7.39 % | 4136 | 1.54 % | 28 | 16 | 19.42 MHz |
| | 2 | Macro: `PRECISION` = 32 | 15 366 | 11.4 % | 4637 | 1.72 % | 28 | 16 | 19.30 MHz |
| | **Merged** | – | 15 910 | 11.8 % | 6048 | 2.25 % | 28 | 16 | 83.33 MHz |

As expected, the reduction in bitwidth of certain signals between versions contributes to a decrease in total resource consumption, and sometimes improves the maximum frequency of the implemented hardware. The generated hardware, on the other hand, shares similar properties in terms of area cost and timing when compared to the original implementations. The resources consumed are increased only by around 2 % with reference to the Artix-7 AC701 FPGA, which is one of the smallest FPGAs in the Xilinx 7-series. Moreover, the maximum frequencies in diffeq1 and bgm are reduced by 10 to 12 %, which are moderate given the functionality that DAM provides.

The maximum frequency supported by the merged hardware in LU8PEEng, on the contrary, is improved by around 4× when compared to the original implementation. This unexpected result arises from a similar timing and fan-out issue mentioned in Section 5.1.4. Originally, the register `recResult` in LU8PEEng is assigned by a wide multiplexer where the inputs are connected to repeating subsets of the same signal. Such assignment incurs a large fan-out and subsequently limits the maximum frequency of every version of implementation. Nevertheless, the bitwidth of `recResult` is defined with macro `PRECISION` and as a result, it is resolved as a common signal with different

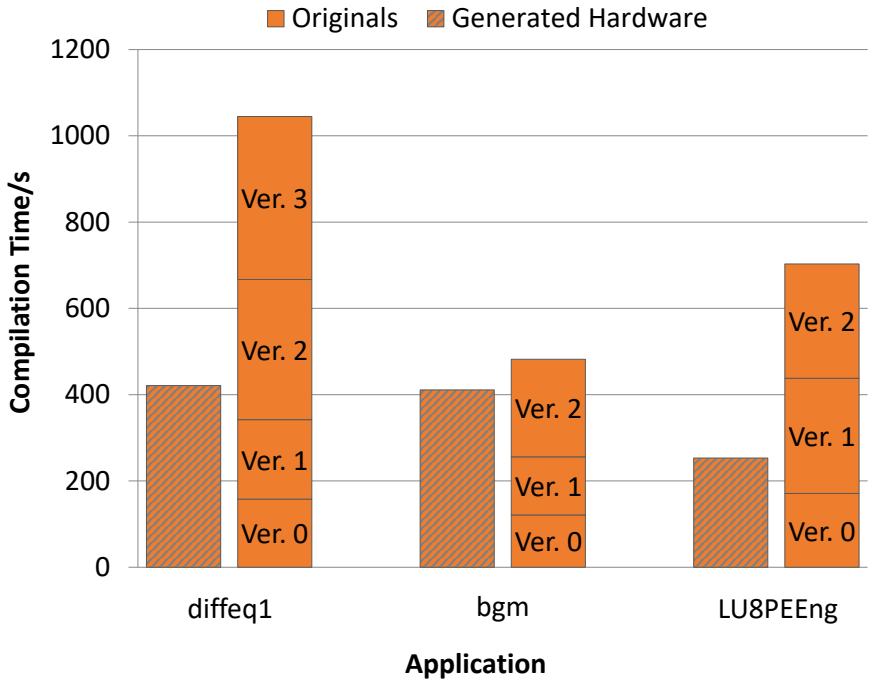



Figure 5.5: Compilation time of the generated hardware versus the combined compilation time of each hardware application.

bitwdiths during the merging process. The insertion of registers for zero-extension increases the number of driving gates for `recResult` and hence the number of fan-out is reduced, which in turn improves the overall timing. We note that such an improvement in the maximum frequency can be obtained by fan-out optimization [128], which can be applied in addition to dataflow graph merging.

Finally, Figure 5.5 shows the total compilation time of the generated hardware versus the sum of compilation time of each hardware application. The time recorded includes the duration of synthesis as well as implementation.

### 5.2.3 Case Study I: Binomial Filters

This subsection presents a case study on one of the applications mentioned at the beginning of this chapter: Binomial Filters. Such filters are efficient structures based on binomial coefficients to realize Gaussian filtering on FPGA.

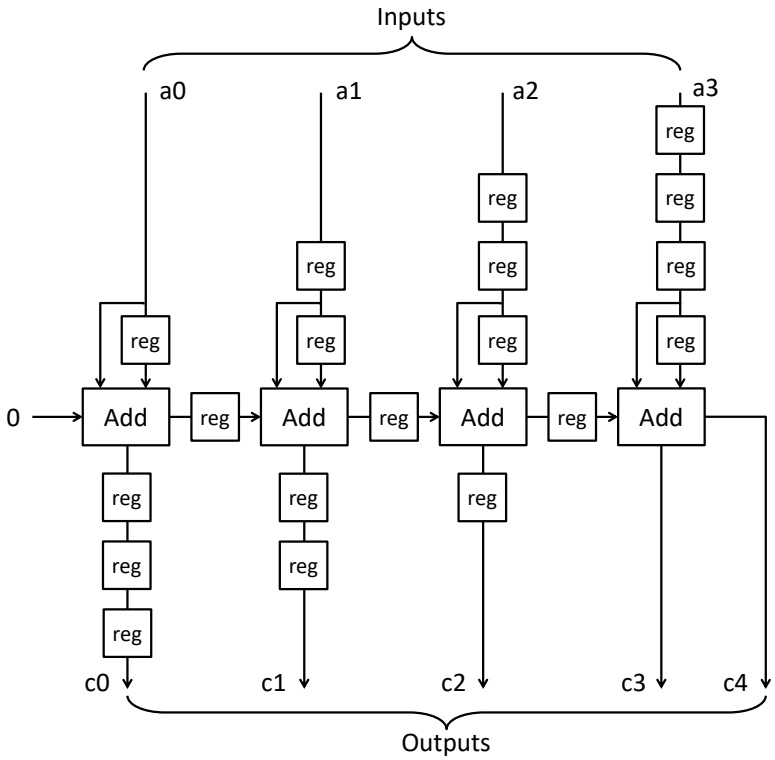



Figure 5.6: A fully-pipelined binomial filter where $n = 4$.

There are numerous possible variations of the basic binomial filter structure and therefore an analysis of the accuracy and frequency response is required when implemented on FPGA [120]. In particular, an analysis with actual hardware is important for such filters because it usually provides more accurate results such as frequency response with respect to signal inputs when compared to software simulation.

An example of a binomial filter used in this experiment is shown in Figure 5.6. The structure of the binomial filter is derived from the polynomial $(1 - z^{-1})^n$, and it can be implemented with a cascade of adders with one of the inputs delayed by a register. Such a cascade is arranged in a pipeline structure where the depth is given by the parameter $n$. The quality of the approximation to Gaussian filter depends on $n$ where the error is reduced to a small value for large filters. For further information about binomial filters, please refer to [120].

Table 5.2: Resource consumption of the generated hardware versus the original hardware for binomial filter. The usage of BRAMs and DSPs are not displayed because they are not used.

| **Version** | **Difference** | **LUTs** | | **Registers** | |
|---|---|---|---|---|---|
| 0 | $n = 14$ | 40 763 | 30.3 % | 67 632 | 25.1 % |
| 1 | $n = 15$ | 43 549 | 32.4 % | 72 464 | 26.9 % |
| 2 | $n = 16$ | 46 335 | 34.4 % | 77 312 | 28.7 % |
| 3 | $n = 17$ | 49 887 | 37.1 % | 82 176 | 30.5 % |
| 4 | $n = 18$ | 52 929 | 39.3 % | 87 056 | 32.3 % |
| 5 | $n = 19$ | 55 715 | 41.4 % | 91 952 | 34.1 % |
| 6 | $n = 20$ | 58 501 | 43.5 % | 96 864 | 36.0 % |
| **Merged** | - | 61 536 | 45.7 % | 101 420 | 37.7 % |

### Experimental Setup

We populate multiple binomial filters on the FPGA to allow parallel processing where each of them supports 64-bit calculation. The FPGA is populated with 32 filters so that the total resource consumption is around 40%. As mentioned above, the depths of the filters need to be fine-tuned to determine the accuracy of the binomial filters. Therefore, in this experiment, the depth of the filters is varied in each version while the target frequency is fixed at 100 MHz. Similar to the previous benchmarks, optional timing optimization is not activated in this case study, and the generated hardware and the original design versions are synthesized and implemented individually using Vivado with the default settings. Table 5.2 illustrates the configuration details for every version and also the corresponding implementation results.

### Experimental Results

The area cost of every version and of the combined hardware are shown in Table 5.2. Obviously, the reduction of depth contributes to a decrease in total area cost in each version, while the resource consumption of the combined

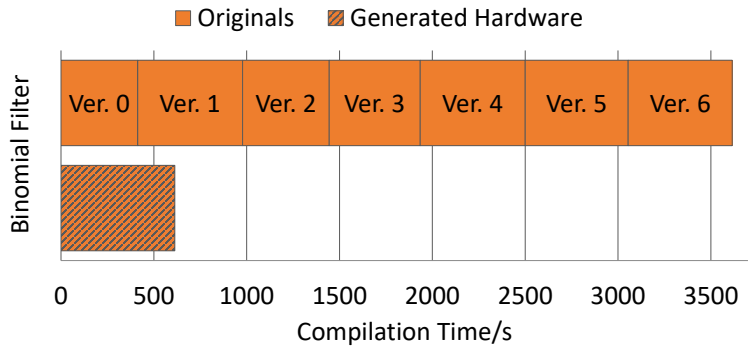



Figure 5.7: Compilation time of the generated hardware versus the combined compilation time of the originals for the binomial filter.

hardware remains competitive compared to the originals. The LUTs and registers consumed are only increased by around 2% with reference to the target Artix-7 AC701 FPGA. This clearly showcases the efficiency of the MCS approximation algorithm since the bitwidth is set to be identical across versions, and the merging of common signals with different bitwidths is not executed in this case study.

The total compilation time, on the other hand, is presented in Figure 5.7. Similar to the VTR benchmark, the time recorded includes the duration of synthesis as well as implementation. From the figure, it can be seen that the speed-up in compilation time is around 5.9× when compared to the combined compilation time of all the originals. In particular, the overall compilation time is reduced from 1 hour to around 10 minutes. Such a significant result is due to the increase in version counts and also the relatively high similarity between versions. It shows that the MCS algorithm proposed in Section 5.1 is able to identify most of the common vertices among all the dataflow graphs, and this contributes to a promising speed-up in compilation time with only a minor increase in resource consumption. Also, it is expected that the overall compilation speed-up will be more significant if more versions are supplied to the proposed design merger. Finally, the execution time for MCS detection and dataflow graph merging is only 1.43 seconds and is insignificant compared to the synthesis and compilation time.

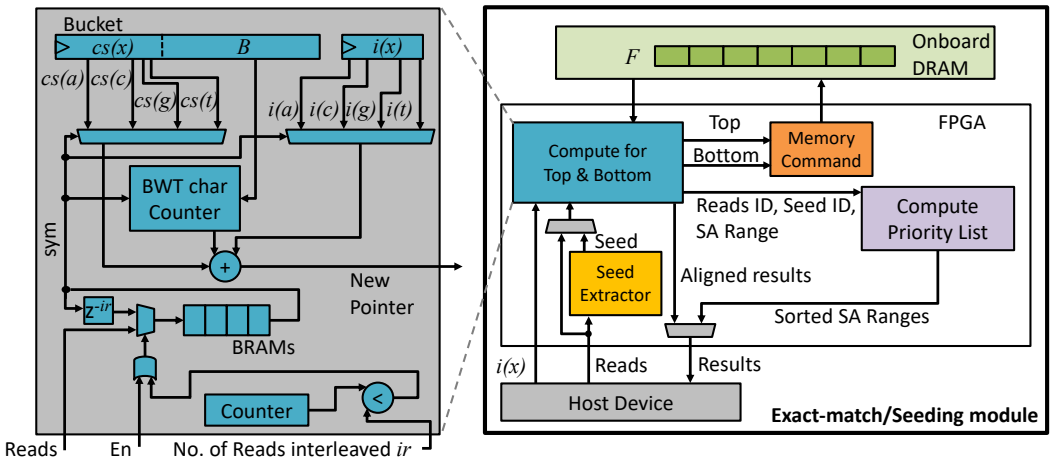



Figure 5.8: Simplified top-level diagram for an exact-match/seeding module. The BRAM and counter in the bottom left of the figure determine the number of reads interleaved. For readability it only displays the logic for one pointer calculation, and some data and control paths are omitted.

### 5.2.4 Case Study II: FM-index

This subsection presents another study that is based on the FM-index implementations presented in Chapter 3 and 4. In particular, we look at the implementation of the exact-match/seeding strategy in this case study. As mentioned, we introduce a data interleaving scheme to optimize our FM-index implementation. By processing multiple reads concurrently, it can negate the memory access latency and enable almost full utilization of FPGA at every cycle.

As shown in Figure 5.8, BRAMs are used to store the interleaved reads. Another counter, which stops at its maximum, is implemented to control the multiplexer at the input of the BRAMs. These two components determine the number of reads that are processed concurrently to mask the memory access latency, which varies between platforms and DDR technologies.

#### Experimental Setup

The initial implementation of the FM-index comes with five versions where each of them concurrently processes a different number of reads. This number

Table 5.3: Resource consumption of the generated hardware versus the original hardware for the FM-index implementation (exact-match/seeding). The usage of DSPs is not displayed because all the hardware uses 36 (0.53 %) of them. Percentage values are relative to the available resources on target Ultrascale+ VU9P FPGA.

| **Version** | **No. of Reads Interleaved** | **LUTs** | | **Registers** | | **BRAMs** | |
|---|---|---|---|---|---|---|---|
| 0 | 352 | 127 584 | 10.8 % | 239 313 | 10.1 % | 997 | 18.9 % |
| 1 | 368 | 127 442 | 10.1 % | 239 262 | 10.1 % | 1000 | 18.9 % |
| 2 | 384 | 127 378 | 10.8 % | 239 272 | 10.1 % | 1000 | 18.9 % |
| 3 | 400 | 127 079 | 10.8 % | 239 196 | 10.1 % | 1003 | 19.0 % |
| 4 | 416 | 127 380 | 10.8 % | 239 228 | 10.1 % | 1003 | 19.0 % |
| **Merged** | - | 134 596 | 11.4 % | 236 662 | 10.0 % | 1213 | 23.0 % |

affects the depth of the BRAMs and the maximum value of the counter. To determine the version that provides the optimal alignment performance, we apply the proposed DAM approach available as an automated dataflow graph merger in MaxCompiler 2018.2 [122], and synthesize and implement the merged design using the default settings of Vivado 2018.2. We also synthesize and implement each individual version to obtain the original compilation time and resource consumption. Table 5.3 illustrates the configuration details for every version and also the corresponding implementation results.

The frequency is fixed at 200 MHz for all the versions and the merged hardware. Similar to Chapter 3 and 4, Xilinx Ultrascale+ VU9P is used as the target FPGA connected to three DIMMs of 16 GB onboard memory. We run the synthesis and implementation on Dell PowerEdge R740XD Server with Intel Xeon Gold 6154 3.40GHz CPU and 754GB RAM.

### Experimental Results

The area cost of every version and the combined hardware are shown in Table 5.3. For each individual version, the increase in the number of reads interleaved contributes to an increase in LUTs, Registers or BRAMs, while the

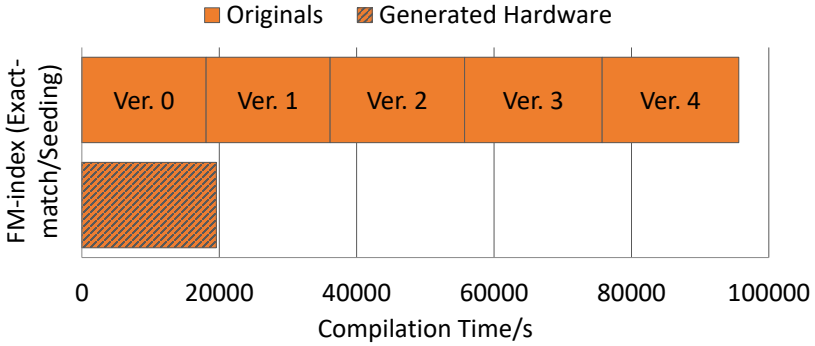



Figure 5.9: Compilation time of the generated hardware versus the combined compilation time of the originals for the FM-index implementation (exact-match/seeding).

resource consumption of the combined hardware is fairly similar to the original versions. The LUTs and BRAMs consumed are increased by around 1.3% and 4.1% respectively with reference to the target Ultrascale+ VU9P FPGA. The registers consumed, however, are decreased because some registers are inferred to BRAMs in Vivado synthesis and implementation.

The total compilation time is displayed in Figure 5.9, and it can be seen that the speed-up in compilation time is around 4.89× when compared to the combined compilation time of all the originals. The overall compilation time is reduced from 26.6 hours to around 5.43 hours. Note the time recorded in this experiment is the duration of MaxCompiler compilation which consists of Vivado synthesis and implementation.

Finally, the alignment time of the exact-match and seeding strategy is shown in Figure 5.10. For the exact-match alignment, the alignment time is decreased from 304 s to 198 s when the interleaved counts are increased from 336 to 368 reads. The alignment time saturates at 198 s despite the number of reads interleaved being increased further. We can also see a similar pattern for the seeding strategy. Based on this experiment, we can determine the number of reads required, *i.e* 368, to mask the memory access latency.

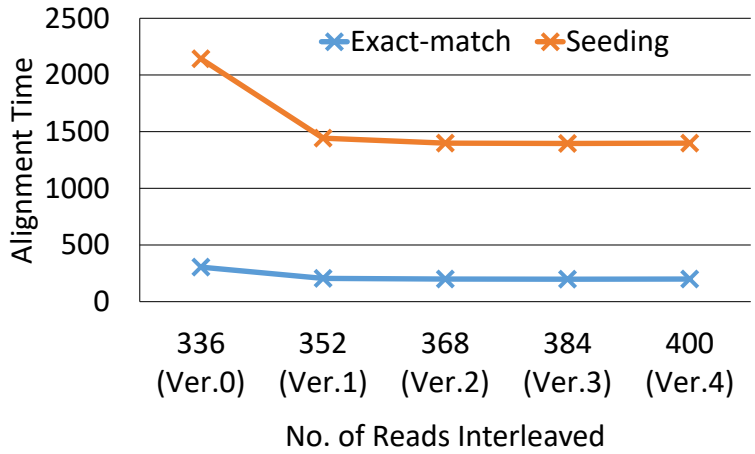



Figure 5.10: The alignment time of the exact-match and seeding strategy versus different number of reads interleaved.

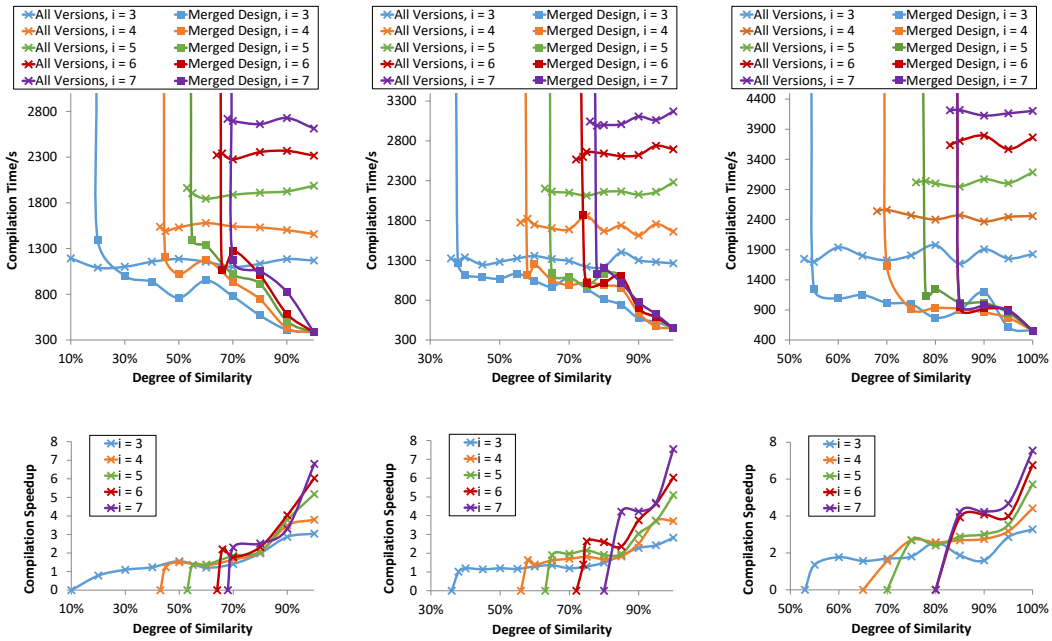
## 5.3 Evaluation

As the above experiment is based on the architecture of the applications provided by VTR Benchmarks, binomial filters and FM-index, the variations or the degree of similarity between versions cannot be adjusted randomly. In this evaluation, we explore the relationship between compilation time and degree of similarity by varying the number of design versions $i$ and its resource consumption on FPGA.

### 5.3.1 Evaluation Setup

We populate the FPGA with multiple diffeq1 modules so that 30 %, 40 % and 50 % of the FPGA slices are initially occupied by each unmerged design version. After that, we introduce discrepancies between versions by changing the signal names literally. This enables a fine-grained adjustment of the degree of similarity between versions. Then all the design versions are applied to the prototype merger to generate a merged design, which is passed to Vivado subsequently to record the compilation time. The original versions are also synthesized and implemented separately in order to make a comparison.

Essentially, the compilation time of the unmerged designs is the summa-




(a) Slices occupied = 30 % (b) Slices occupied = 40 % (c) Slices occupied = 50 %

Figure 5.11: The relationship between compilation time and degree of similarity.

tion of the synthesis and implementation time of all the versions. For the merged designs, the compilation time simply refers to its own synthesis and implementation time. The degree of similarity is defined as the proportion of computational hardware that is common between versions. We note that we do not use the context of dataflow graph for this definition because each node can represent different hardware types which contribute to different area costs.

### 5.3.2 Evaluation Results

Figure 5.11 displays a summary of the experimental results which demonstrates the scalability of the proposed approach. Basically, 100% similarity refers to the scenario that every design version is logically equivalent, and it indicates the scope of the maximum compilation speed-up. The soaring compilation time in the figure illustrates the failure of placement and routing in which the merged hardware is larger than the area of the given FPGA.

Originally, the total compilation time is linearly proportional to the number of versions as indicated by the flat lines in the figure, whereas the compilation time of the generated hardware is independent of the version counts. Since the synthesis and implementation time of the merged designs purely depends on the degree of similarity, extra logic is only introduced when there exist variations between versions. Thus, the compilation time increases with the decrease of similarity until the FPGA runs out of resources for the generated hardware.

It is also noticed that the compilation time of the generated hardware is largely similar regardless of the numbers of versions when every version is 85 % to 100 % in common. The compilation time is around 400 s to 1000 s which is at least 3× faster when there are seven versions of the same design. The compilation speed-up can be further improved to around 5× if the versions are 95 % similar. It is expected that, based on the assumption that there is adequate space on chip, the improvement will be larger when more designs are merged. Finally, we note that although the performance numbers are based on diffeq1, parameters such as relative compilation speed-up are important specifications for other applications when DAM is employed by designers to perform FPGA design optimization.

## 5.4 Previous Work

The concept of supporting multiple versions is described in [129] where conservation cores, i.e. specialized processors that focus on reducing energy, are designed to run both past and future versions of code. However, the notation of versions in [129] is different from our work since each version in the proposed approach is independent of each other. In other words, all the variants between versions are already known at the time when designers need to synthesize and implement the hardware.

Automated dataflow graph merging has been extensively studied in the

context of runtime reconfiguration, high-level synthesis and instruction set extension. Fazlali *et al.* [124, 130] propose a datapath merging algorithm based on approximating the maximum weighted clique to shorten the bitstream and to reduce reconfiguration time. Voss *et al.* [122] present a cost-driven heuristic to minimize the area cost within an HLS application. Other work such as [131–135] focuses on resource sharing of multiple instruction set extensions (ISEs) for extensible base processors. For example, a path-based heuristic approach is presented in [131] in which a set of ISEs is transformed to a hardware datapath. Maximal subsequences problem is then applied to maximize area reduction. Zuluaga and Topham later extend the work by introducing latency constraints in the merging process [134]. Similarly, a heuristic that uses the construction of a compatibility graph is proposed in [132] and a non-exact method is suggested to perform datapath merging. Such heuristic is also employed in [135] to increase area reduction by accounting for the cost of multiplexers.

Since the merging latency is not a prior concern in most of the work mentioned (from exponential to polynomial time complexity), the corresponding algorithms are less appropriate for DAM. A linear-time heuristic is proposed in our work to minimize the merging time because reducing the compilation time is an important objective of the proposed approach.

On the other hand, researchers have tackled the challenge of prolonged hardware implementation, optimization and debugging runtime in many different ways. For example, overlay architectures have been leveraged to offer faster compilation as well as improved programmability and runtime management. Recently, overlays with different granularity ranging from virtual FPGAs [136–140], soft processors [141–143] to CGRA overlays [116, 117, 144–146] and GPU-like overlays [147] have been proposed.

In addition, some have addressed the challenge from a design methodology's perspective. In [148, 149], the authors propose the use of pre-built hard macros and modular design flow to minimize the placement and routing process. A similar approach is also presented in [150] where a library of precom-

piled macros is constructed for HLS. Finally, some researchers have devoted their efforts to low-level FPGA EDA tools to improve implementation speed. In [151], the authors accelerate the placement and routing process by making quality-runtime tradeoffs. The implementation runtime can also be improved by parallelizing the placement algorithm [152, 153]. Dynamic partial reconfiguration is leveraged in [154] and [155] to shorten runtime by effectively reducing the user design size.

Compared to these contributions, the proposed approach represents a complementary solution to improve designers' productivity by eliminating the need to perform placement and routing for different design versions repeatedly. It is possible to use DAM together with the above optimization techniques to reduce the compilation time, and such opportunities are explored in the next chapter.

## 5.5 Summary

In this chapter, we propose a new approach DAM to merge multiple FPGA designs into a single one for rapid functional evaluation. The novel aspects of DAM are:

1. A novel approximate maximum common subgraph detection algorithm with linear time complexity is proposed to maximize the sharing of FPGA resources after design merging.

2. A prototype tool is developed to realize the maximum common subgraph detection algorithm for dataflow graphs derived from Verilog designs. It also generates the appropriate control circuits to enable the selection of each design version at runtime.

3. A comprehensive analysis of compilation time versus degree of similarity is done, and the proposed approach can reduce the compilation time by 3× to 5×.

# Chapter 6

# Conclusion

In this work, we demonstrate the use of FPGA to accelerate biologically accurate short read alignment. This is achieved through accelerating FM-index-based alignment and the Smith-Waterman algorithm with the affine-gap model. We also propose an automatic design analyzer and merger to further optimize the FM-index implementation. Below we summarize the main contributions of this work:

#### Exact-match and mismatch alignment with FM-index

We develop and implement a four-stage alignment pipeline that follows a similar workflow in Bowtie [156, 157]. By leveraging the runtime reconfigurability of FPGA, exact-match, one-mismatch, two-mismatch and three-mismatch alignment strategies are configured in turn on the FPGA. We use FM-index to achieve biologically accurate alignment by considering complete biological information in the read data. Additionally, we maximize the performance using three major techniques: First, we compress and partition the FM-index into buckets. The length of each bucket is equal to an optimal multiple of the memory burst size. Second, we interleave the memory access with multiple short reads to negate the access latency. Finally, we apply bi-directional FM-index to reduce the search space for the one-mismatch and two-mismatch

alignment strategies. Experimental results indicate that our design maximizes alignment accuracy compared to the state-of-the-art software Bowtie, mapping reads 4.27 × as fast. Compared to the previous hardware aligner, our achieved accuracy is 97.7 % which reports 4.48 M more valid alignments with a similar speed.

#### Indel alignment with exploration of different alignment pipelines

We implement the seed-and-extend strategy on FPGA to support indel alignment that includes matches, mismatches, deletions and insertions [157, 158]. The exact-match implementation is extended to perform seeding while a Smith-Waterman implementation with the affine gap model is developed for the extension. We maximize their performance using two major techniques: First, a set of comparators and shift registers is implemented so as to maintain the hardware priority list in seeding. Second, data interleaving is used in the extension to facilitate the anti-diagonal calculation of the three matrics. Moreover, we arrange one or multiple alignment strategies into a few sequential pipelines. For each pipeline, we investigate the speed-up and accuracy to provide guidance for genomics scientists. Experimental results indicate that the pipeline: exact-match alignment + one-mismatch alignment+ seed-and-extend strategies provides a desirable performance (5.94 × speed-up) with comparable accuracy (95.2 %) based on Bowtie2.

#### Runtime reconfigurable alignment pipeline

We propose a multi-configuration alignment pipeline by exploiting the runtime reconfigurability of FPGA. Distinct hardware implementations are loaded in turn, so as to run the alignment strategies in the required order. This architecture can eliminate the data hazards between strategies, provide a balanced pipeline and promote portability as users can have better control over alignment parameters. Implementations can be re-arranged, removed, or added

straightforwardly to accommodate different sequenced data quality and experiments, similar to our exploration in Chapter 4.

### Automatic Design Analysis and Merging

We propose a design analyzer and merger to combine multiple FPGA designs into a single hardware design [159], so that multiple place-and-route tasks can be replaced by a single task to speed up functional evaluation of designs, especially during the development process. This approach has the following three key elements: First, we propose an approximate maximum common subgraph detection algorithm with linear time complexity to combine multiple hardware design versions. Second, we develop a prototype tool that implements this algorithm and it generates the appropriate control circuits to enable the selection of original design versions at runtime. Finally, we provide a comprehensive analysis of compilation time versus degree of similarity to identify the optimized user parameters. Experimental results show that this approach can reduce compilation time by around $5\times$ when each design is $95\,\%$ similar to the others, and the compilation time is reduced from 1 hour to 10 minutes in the case of binomial filters. With this approach, we can also obtain the optimal number for data interleaving in the exact-match and seeding implementation of the FM-index.

Overall, the work presented in this thesis has focused on accelerating biologically accurate short read alignment on FPGA, and speeding-up FPGA development with a design analyzer and merger. We demonstrate that FPGA can provide a substantial speed-up relative to state-of-the-art software tools with reasonable accuracy. We also show that the design analyzer and merger can reduce compilation time by $3\times$ to $5\times$, and provide further optimization to the FM-index implementation. In the following sub-section, we summarize the potential extensions and future work.

## 6.1 Future Work

1. For the time being, we have applied our work to accelerate single-end read alignment and obtained the aligned positions accurately. However, the NGS machines can also generate paired-end reads [160] where both ends of the DNA fragment are sequenced. Each pair is separated by a known inner distance. This enables much more accurate read alignment and higher sensitivity for detecting indels. Therefore, an important extension of the proposed aligner is to support paired-end read alignment. Initial research will explore the use of the host processor to resolve the final aligned position of each read pair. Subsequent research will investigate the parallel architecture on FPGA to accelerate the comparison process for all aligned positions within each read pair.

2. Most software aligners, such as Bowtie, Bowtie2 and BWA-MEM not only experiment with real genomics data, but also evaluate their speed, accuracy and sensitivity using simulated dataset. Read simulators such as Mason [161] or ART [162] can generate synthetic NGS reads with customized read error and quality parameters. These tools simulate reads based on the reference genome, hence the actual position of each read can be known when generated. Also, errors are introduced to the reads based on empirical distribution which is often measured using large training datasets. For this reason, we can extend the evaluation of the proposed alignment pipeline using simulated reads with different read lengths, error rates, or coverage. Accordingly, we can obtain the corresponding accuracy and sensitivity and compare them against state-of-the-art software.

3. Currently, our FM-index implementation is based on traditional DDR SDRAM architecture. In recent years we have seen advancements in memory technologies such as Hybrid Memory Cube or High Bandwidth

Memory. For example, the latest Xilinx Virtex FPGA is incorporated with both traditional DDR SDRAM and High Bandwidth Memory technologies, while Xilinx Virtex VU9P FPGA on Amazon F1 instance [163] is equipped with the DDR SDRAM and Hybrid Memory Cube. It would be interesting to see the performance difference when our FM-index implementation is ported onto these platforms. Our primary research will be the extension of the proposed alignment pipeline onto F1 instance. Subsequently, we can obtain the alignment speed, accuracy and sensitivity and compare them to Illumina DRAGEN Bio-IT Platform [164].

4. Currently, the human reference genome only represents a small number of individuals. This limits its usefulness for genotyping because it does not reflect the genetic diversity of large populations. Specifically, some sequences from humans are not included in the construction of the reference genome. These sequences can be aligned incorrectly when they originate from a region that varies from the reference. Therefore, a graph FM-index is proposed for short read mapping in recent years, as it captures the entire human genome along with a large number of variants [165]. An interesting future work will be the development of a graph FM-index accelerator since memory access is also a major bottleneck in this algorithm.

5. We would like to extend the automatic design analyzer and merger by improving the merging heuristic, and including more functionalities and evaluations. First, we will consider multiplexing versus operator savings to determine the optimal merging heuristic of common signals with multi-bitwidth. Second, we will extend the signal merging for floating-point numbers. Since a floating-point number consists of a sign, exponent and mantissa, the merging process is less trivial compared to integer representation. Lastly, we will evaluate the design analyzer and merger with additional applications, such as the graph FM-index hard-

ware when it is implemented, machine learning implementations with adjustable mantissa or fused convolution blocks designed for different domestic machine learning applications [166–168], or object detection and classification for high-speed asymmetric-detection time-stretch optical microscopy on FPGA [169–172].

# Bibliography


[1] Lex Nederbragt, "developments in NGS," 2021. [Online]. Available: https://figshare.com/articles/dataset/developments_in_NGS/100940

[2] Illumina, Inc., "NovaSeq 6000 System," 2021. [Online]. Available: https://emea.illumina.com/systems/sequencing-platforms/novaseq.html

[3] G. Lightbody *et al.*, "Review of applications of high-throughput sequencing in personalized medicine: barriers and facilitators of future progress in research and clinical application," *Briefings in Bioinformatics*, vol. 20, no. 5, pp. 1795–1811, 2019.

[4] B. Langmead *et al.*, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biology*, vol. 10, no. R25, 2009.

[5] N. B. Tsui *et al.*, "Maternal plasma RNA sequencing for genome-wide transcriptomic profiling and identification of pregnancy-associated transcripts," *Clinical Chemistry*, vol. 60, no. 7, pp. 954–962, 2014.

[6] J. Arram *et al.*, "Reconfigurable Filtered Acceleration of Short Read Alignment," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 438–441.

[7] N. A. Miller *et al.*, "A 26-hour system of highly sensitive whole genome sequencing for emergency management of genetic diseases," *Genome Medicine*, vol. 7, no. 100, 2015.

[8] S. Salamat and T. Rosing, "FPGA Acceleration of Sequence Alignment: A Survey," *arXiv preprint arXiv:2002.02394v2*, 2020.

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

[10] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[11] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, pp. 357–359, 2012.

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

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

[14] B. Langmead and A. Nellore, "Cloud computing for genomic data analysis and collaboration," *Nature Reviews Genetics*, vol. 19, p. 208–219, 2018.

[15] J. Rose *et al.*, "The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing," in *2012 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2012, pp. 77–86.

[16] J. Luu *et al.*, "VTR 7.0: Next Generation Architecture and CAD System for FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 7, no. 2, pp. 6:1–6:30, 2014.

[17] M. Burrows and D. Wheeler, "A Block-sorting Lossless Data Compression Algorithm," Digital Equipment Corporation, Tech. Rep., 1994.

[18] U. Manber and G. Myers, "Suffix Arrays: A New Method for On-Line String Searches," *SIAM Journal on Computing*, vol. 22, no. 5, p. 935–948, 1993.

[19] R. Durbin *et al.*, *Biological Sequence Analysis Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press, 1998.

[20] S. A. Guccione and E. Keller, "Gene Matching Using JBits," in *12th International Conference on Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, 2002, pp. 1168–1171.

[21] K. Puttegowda *et al.*, "A Run-Time Reconfigurable System for Gene-Sequence Searching," in *16th International Conference on VLSI Design*, 2003, pp. 561–566.

[22] C. W. Yu *et al.*, "A Smith-Waterman Systolic Cell," in *13th International Conference on Field-Programmable Logic and Applications*, 2003, pp. 375–384.

[23] T. V. Court and M. C. Herbordt, "Families of FPGA-based Algorithms for Approximate String Matching," in *15th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 2004, pp. 354–364.

[24] ——, "Families of FPGA-Based Accelerators for Approximate String Matching," *Microprocessors and Microsystems*, vol. 31, no. 2, pp. 135–145, 2007.

[25] X. Jiang *et al.*, "A Reconfigurable Accelerator for Smith-Waterman Algorithm," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 12, pp. 1077–1081, 2007.

[26] T. Oliver *et al.*, "Hyper Customized Processors for Bio-sequence Database Scanning on FPGAs," in *ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, 2005, pp. 229–237.

[27] M. Gok and C. Yilmaz, "Efficient Cell Designs for Systolic Smith-Waterman Implementations," in *2006 International Conference on Field Programmable Logic and Applications*, 2006, pp. 1–4.

[28] P. Faes *et al.*, "Scalable Hardware Accelerator for Comparing DNA and Protein Sequences," in *1st International Conference on Scalable Information Systems*, 2006, pp. 33–es.

[29] A. Boukerche *et al.*, "Reconfigurable Architecture for Biological Sequence Comparison in Reduced Memory Space," in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.

[30] I. T. Li *et al.*, "160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA)," *BMC Bioinformatics*, vol. 8, no. 185, 2007.

[31] P. Zhang *et al.*, "Implementation of the Smith-Waterman Algorithm on a Reconfigurable Supercomputing Platform," in *1st International Workshop on High-Performance Reconfigurable Computing Technology and Applications: Held in Conjunction with SC07*, 2007, pp. 39–48.

[32] K. Benkrid *et al.*, "High Performance Biological Pairwise Sequence Alignment: FPGA versus GPU versus Cell BE versus GPP," *International Journal of Reconfigurable Computing*, vol. 2012, no. 752910, 2012.

[33] M. Y. Gök *et al.*, "Programmable Hardware Based Short Read Aligner Using Phred Quality Scores," in *2013 International Conference on Social Computing*, 2013, pp. 864–867.

[34] K. Benkrid *et al.*, "A Highly Parameterized and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 4, pp. 561–570, 2009.

[35] S. Lloyd and Q. O. Snell, "Hardware Accelerated Sequence Alignment with Traceback," *International Journal of Reconfigurable Computing*, vol. 2009, no. 762362, 2009.

[36] B. Morgenstern *et al.*, "DIALIGN: Finding Local Similarities by Multiple Sequence Alignment," *Bioinformatics*, vol. 14, no. 3, pp. 290–294, 1998.

[37] A. Boukerche *et al.*, "A Hardware Accelerator for the Fast Retrieval of DIALIGN Biological Sequence Alignments in Linear Space," *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 808–821, 2010.

[38] N. Sebastião *et al.*, "Integrated Hardware Architecture for Efficient Computation of the $n$-Best Bio-Sequence Local Alignments in Embedded Platforms," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 7, pp. 1262–1275, 2012.

[39] X. Fei *et al.*, "FPGASW: Accelerating Large-Scale Smith-Waterman Sequence Alignment Application with Backtracking on FPGA Linear Systolic Array," *Interdisciplinary Sciences: Computational Life Sciences*, vol. 10, no. 1, pp. 176–188, 2018.

[40] R. P. Jacobi *et al.*, "Reconfigurable Systems for Sequence Alignment and for General Dynamic Programming," *Genetics and Molecular Research*, vol. 4, no. 3, pp. 543–552, 2005.

[41] O. Creţ *et al.*, "A Hardware Algorithm for The Exact Subsequence Matching Problem in DNA Strings," *Romanian Journal of Information Scient and Technology*, vol. 12, no. 1, pp. 51–67, 2009.

[42] Y. Liu *et al.*, "An FPGA-Based Web Server for High Performance Biological Sequence Alignment," in *NASA/ESA Conference on Adaptive Hardware and Systems*, 2009, pp. 361–368.

[43] D. Greaves *et al.*, "Synthesis of a Parallel Smith-Waterman Sequence Alignment Kernel into FPGA Hardware," in *Many-core Reconfigurable Supercomputing Conference*, 2009, pp. 1–9.

[44] S. F. Altschul *et al.*, "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs," *Nucleic Acids Research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[45] GenBank and WGS Statistics. Accessed on 7-April-2021. [Online]. Available: https://www.ncbi.nlm.nih.gov/genbank/statistics/

[46] Y. Yamaguchi *et al.*, "High Speed Homology Search Using Run-Time Reconfiguration," in *12th International Conference on Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, 2002, pp. 281–291.

[47] E. Sotiriades *et al.*, "FPGA based Architecture for DNA Sequence Comparison and Database Search," in *20th IEEE International Parallel Distributed Processing Symposium*, 2006, pp. 8 pp.–.

[48] ——, "Some Initial Results on Hardware BLAST Acceleration with a Reconfigurable Architecture," in *20th IEEE International Parallel Distributed Processing Symposium*, 2006, pp. 8 pp.–.

[49] E. Sotiriades and A. Dollas, "A General Reconfigurable Architecture for the BLAST Algorithm," *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 48, pp. 189–208, 2007.

[50] F. Xia *et al.*, "Families of FPGA-Based Accelerators for BLAST Algorithm with Multi-seeds Detection and Parallel Extension," in *2nd International Conference on Bioinformatics Research and Development*, 2008, pp. 43–57.

[51] X. Guo *et al.*, "A Systolic Array-Based FPGA Parallel Architecture for the BLAST Algorithm," *ISRN Bioinformatics*, vol. 2012, pp. 1–11, 2012.

[52] Y. Chen *et al.*, "Reconfigurable Accelerator for the Word-Matching Stage of BLASTN," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 4, pp. 659–669, 2013.

[53] P. Krishnamurthy *et al.*, "Biosequence Similarity Search on the Mercury System," *Journal of VLSI Signal Processing System*, vol. 49, pp. 101–121, 2007.

[54] J. D. Buhler *et al.*, "Mercury BLASTN : Faster DNA Sequence Comparison Using a Streaming Hardware Architecture," in *3rd Annual Reconfigurable Systems Summer Institute*, 2007.

[55] J. Lancaster *et al.*, "Acceleration of Ungapped Extension in Mercury BLAST," *Microprocessors and Microsystems*, vol. 33, no. 4, p. 281–289, 2009.

[56] S. Datta *et al.*, "RC-BLASTn: Implementation and Evaluation of the BLASTn Scan Function," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, 2009, pp. 88–95.

[57] P. Afratis *et al.*, "A Rate-based Prefiltering Approach to BLAST Acceleration," in *International Conference on Field Programmable Logic and Applications*, 2008, pp. 631–634.

[58] J. H. Park *et al.*, "CAAD BLASTn: Accelerated NCBI BLASTn with FPGA prefiltering," in *2010 IEEE International Symposium on Circuits and Systems*, 2010, pp. 3797–3800.

[59] S. Kasap *et al.*, "High Performance FPGA-based Core for BLAST Sequence Alignment with the Two-Hit Method," in *8th IEEE International Conference on BioInformatics and BioEngineering*, 2008, pp. 1–7.

[60] H. Abelsson *et al.*, "Accelerating NCBI BLAST FPGA Supercomputing Coming of Age," in *Proceedings of the CUG Conference*, 2007, pp. 1–5.

[61] B. C. Lam *et al.*, "BSW: FPGA-accelerated BLAST-Wrapped Smith-Waterman aligner," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2013, pp. 1–7.

[62] E. Sotiriades and A. Dollas, "Design Space Exploration for the BLAST Algorithm Implementation," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007, pp. 323–326.

[63] F. Xia *et al.*, "Hardware BLAST Algorithms with Multi-seeds Detection and Parallel Extension," in *4th International Workshop Reconfigurable Computing: Architectures, Tools and Applications*, 2008, pp. 39–50.

[64] ——, "FPGA-Based Accelerators for BLAST Families with Multi-Seeds Detection and Parallel Extension," in *2nd International Conference on Bioinformatics and Biomedical Engineering*, 2008, pp. 58–62.

[65] R. D. Chamberlain *et al.*, "The Mercury System: Exploiting Truly Fast Hardware for Data Search," in *the International Workshop on Storage Network Architecture and Parallel I/Os*, 2003, pp. 65–72.

[66] A. Jacob *et al.*, "FPGA-accelerated seed generation in Mercury BLASTP," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007, pp. 95–106.

[67] B. Harris *et al.*, "A Banded Smith-Waterman FPGA Accelerator for Mercury BLASTP," in *2007 International Conference on Field Programmable Logic and Applications*, 2007, pp. 765–769.

[68] A. Jacob *et al.*, "Mercury BLASTP: Accelerating Protein Sequence Alignment," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 1, no. 2, pp. 9:1–9:44, 2008.

[69] M. C. Herbordt *et al.*, "Single Pass, BLAST-Like, Approximate String Matching on FPGAs," in *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 217–226.

[70] ——, "Single Pass Streaming BLAST on FPGAs," *Parallel Computing*, vol. 33, no. 10-11, pp. 741–756, 2007.

[71] J. H. Park *et al.*, "CAAD BLASTP: NCBI BLASTP Accelerated with FPGA-Based Accelerated Pre-Filtering," in *17th IEEE Symposium on Field Programmable Custom Computing Machines*, 2009, pp. 81–87.

[72] K. Muriki *et al.*, "RC-BLAST: towards a portable, cost-effective open source hardware implementation," in *19th IEEE International Parallel and Distributed Processing Symposium*, 2005, pp. 8 pp.–.

[73] S. Aluru and N. Jammula, "A Review of Hardware Acceleration for Computational Genomics," *IEEE Design Test*, vol. 31, no. 1, pp. 19–30, 2014.

[74] J. D. Thompson *et al.*, "CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice," *Nucleic Acids Research*, vol. 22, no. 22, p. 4673–4680, 1994.

[75] T. Oliver *et al.*, "Multiple Sequence Alignment on an FPGA," in *11th International Conference on Parallel and Distributed Systems*, 2005, pp. 326–330.

[76] X. Lin *et al.*, "To Accelerate Multiple Sequence Alignment using FPGAs," in *Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, 2005, pp. 5 pp.–180.

[77] S. Lloyd and Q. O. Snell, "Accelerated large-scale multiple sequence alignment," *BMC Bioinformatics*, vol. 12, no. 466, pp. 1–10, 2011.

[78] E. Fernandez *et al.*, "Exploration of Short Reads Genome Mapping in Hardware," in *2010 International Conference on Field Programmable Logic and Applications*, 2010, pp. 360–363.

[79] Y. Chen *et al.*, "An FPGA Aligner for Short Read Mapping," in *22nd International Conference on Field Programmable Logic and Applications*, 2012, pp. 511–514.

[80] C. B. Olson *et al.*, "Hardware Acceleration of Short Read Mapping," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012, pp. 161–168.

[81] E. Fernandez *et al.*, "Multithreaded FPGA acceleration of DNA Sequence Mapping," in *IEEE Conference on High Performance Extreme Computing*, 2012, pp. 1–6.

[82] T. B. Preußer *et al.*, "Short-Read Mapping by a Systolic Custom FPGA Computation," in *IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012, pp. 169–176.

[83] K. Koliogeorgi *et al.*, "Dataflow Acceleration of Smith-Waterman with Traceback for High Throughput Next Generation Sequencing," in *29th International Conference on Field Programmable Logic and Applications*, 2019, pp. 74–80.

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

[85] W. Tang *et al.*, "Accelerating Millions of Short Reads Mapping on a Heterogeneous Architecture with FPGA Accelerator," in *IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012, pp. 184–187.

[86] P. Draghicescu *et al.*, *Inexact Search Acceleration on FPGAs Using the Burrows-Wheeler Transform.* Pico Computing, 2012.

[87] S. S. Banerjee *et al.*, "ASAP: Accelerated Short-Read Alignment on Programmable Hardware," *IEEE Transactions on Computers*, vol. 68, no. 3, p. 331–346, 2019.

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

[89] ——, "Power-Efficiency Analysis of Accelerated BWA-MEM Implementations on Heterogeneous Computing Platforms," in *International Conference on ReConFigurable Computing and FPGAs*, 2016, pp. 1–8.

[90] Y. T. Chen *et al.*, "A Novel High-Throughput Acceleration Engine for Read Alignment," in *IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 199–202.

[91] M. C. F. Chang *et al.*, "The SMEM Seeding Acceleration for DNA Sequence Alignment," in *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2016, pp. 32–39.

[92] A. D. Smith *et al.*, "Using quality scores and longer reads improves accuracy of Solexa read mapping," *BMC Bioinformatics*, vol. 9, no. 128, pp. 1–8, 2008.

[93] H. Li *et al.*, "Mapping short DNA sequencing reads and calling variants using mapping quality scores," *Genome Research*, vol. 18, no. 11, pp. 1851–1858, 2008.

[94] E. Fernandez *et al.*, "String Matching in Hardware Using the FM-Index," in *IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2011, pp. 218–225.

[95] ——, "FHAST: FPGA-Based Acceleration of Bowtie in Hardware," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 12, no. 5, pp. 973–981, 2015.

[96] Y. Chen *et al.*, "A hybrid short read mapping accelerator," *BMC Bioinformatics*, vol. 14, no. 67, pp. 1–14, 2013.

[97] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[98] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows–Wheeler transform," *Bioinformatics*, vol. 25, no. 14, p. 1754–1760, 2009.

[99] O. Knodel *et al.*, "Next-generation massively parallel short-read mapping on FPGAs," in *22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2011, pp. 195–201.

[100] T. B. Preußer *et al.*, "PoC-align: An open-source alignment accelerator using FPGAs," in *International Conference on ReConFigurable Computing and FPGAs*, 2014, pp. 1–6.

[101] J. Arram *et al.*, "Hardware Acceleration of Genetic Sequence Alignment," in *9th International Symposium on Reconfigurable Computing: Architectures, Tools and Applications*, 2013, pp. 13–24.

[102] Y. Chen *et al.*, "PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds," *Bioinformatics*, vol. 25, no. 19, p. 2514–2521, 2009.

[103] J. Cong *et al.*, "SMEM++: A Pipelined and Time-Multiplexed SMEM Seeding Accelerator for Genome Sequencing," in *28th International Conference on Field Programmable Logic and Applications*, 2018, pp. 210–214.

[104] N. Ahmed *et al.*, "Heterogeneous Hardware/Software Acceleration of the BWA-MEM DNA Alignment Algorithm," in *IEEE/ACM International Conference on Computer-Aided Design*, 2015, pp. 240–246.

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

[106] B. Ewing *et al.*, "Base-Calling of Automated Sequencer Traces Using Phred. I. Accuracy Assessment," *Genome Research*, vol. 8, no. 3, p. 175–185, 1998.

[107] A. A. Hernandez-Lopez *et al.*, "Toward a Dynamic Threshold for Quality Score Distortion in Reference-Based Alignment," *Journal of Computational Biology*, vol. 27, no. 2, pp. 288–300, 2020.

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

[109] B. Langmead *et al.*, "Scaling read aligners to hundreds of threads on general-purpose processors," *Bioinformatics*, vol. 35, no. 3, p. 421–432, 2019.

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

[111] J. K. Sehn, "Chapter 9 - insertions and deletions (indels)," in *Clinical Genomics*. Academic Press, 2015, pp. 129–150.

[112] M. R. Nelson *et al.*, "Large-scale validation of single nucleotide polymorphisms in gene regions," *Genome Research*, vol. 14, no. 8, pp. 1664–1668, 2004.

[113] M. A. Eberle *et al.*, "A reference data set of 5.4 million phased human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree," *Genome Research*, vol. 27, no. 1, pp. 157–164, 2017.

[114] I. Skliarova and A. de Brito Ferrari, "Reconfigurable Hardware SAT Solvers: A Survey of Systems," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1449–1461, Nov 2004.

[115] C. Kachris and D. Soudris, "A Survey on Reconfigurable Accelerators for Cloud Computing," in *26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–10.

[116] C. Liu *et al.*, "Automatic Nested Loop Acceleration on FPGAs Using Soft CGRA Overlay," in *2nd International Workshop on FPGAs for Software Programmer (FSP)*, 2015, pp. 13–18.

[117] ——, "QuickDough: A Rapid FPGA Loop Accelerator Design Framework Using Soft CGRA Overlay," in *2015 International Conference on Field Programmable Technology (FPT)*, 2015, pp. 56–63.

[118] D. Koch *et al.*, *FPGAs for Software Programmers*. Springer International Publishing, 2016.

[119] B. O'Sullivan, *Mercurial: The Definitive Guide*. O'Reilly Media, 2009.

[120] M. Aubury and W. Luk, "Binomial Filters," *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, vol. 12, no. 1, pp. 35–50, Jan. 1996.

[121] J. S. Targett *et al.*, "Lower Precision for Higher Accuracy: Precision and Resolution Exploration for Shallow Water Equations," in *2015 International Conference on Field Programmable Technology (FPT)*, 2015, pp. 208–211.

[122] N. Voss *et al.*, "Automated Dataflow Graph Merging," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2016, pp. 219–226.

[123] H. Bunke *et al.*, "A Comparison of Algorithms for Maximum Common Subgraph on Randomly Connected Graphs," in *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, 2002, pp. 123–132.

[124] M. Fazlali *et al.*, "Efficient Datapath Merging for the Overhead Reduction of Run-time Reconfigurable Systems," *The Journal of Supercomputing*, vol. 59, pp. 636–657, Feb. 2012.

[125] J. H. Rutgers *et al.*, "An Approximate Maximum Common Subgraph Algorithm for Large Digital Circuits," in *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, 2010, pp. 699–705.

[126] J. Bachrach *et al.*, "Chisel: Constructing Hardware in a Scala Embedded Language," in *Design Automation Conference (DAC)*, 2012, pp. 1212–1221.

[127] S. Takamaeda-Yamazaki, "Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL," in *Applied Reconfigurable Computing: 11th International Symposium (ARC)*, 2015, pp. 451–460.

[128] H. J. Hoover *et al.*, "Bounding Fan-out in Logical Networks," *Journal of the Association for Computing Machinery*, vol. 31, no. 1, pp. 13–18, Jan 1984.

[129] G. Venkatesh *et al.*, "Conservation Cores: Reducing the Energy of Mature Computations," in *International Conference on Architectural Sup-*

*port for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 205–218.

[130] M. Fazlali *et al.*, "High Speed Merged-datapath Design for Run-time Reconfigurable Systems," in *International Conference on Field-Programmable Technology (FPT)*, 2009, pp. 339–343.

[131] P. Brisk *et al.*, "Area-efficient Instruction Set Synthesis for Reconfigurable System-on-chip Designs," in *41st Annual Design Automation Conference (DAC)*, 2004, pp. 395–400.

[132] N. Moreano *et al.*, "Efficient datapath merging for partially reconfigurable architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 7, pp. 969–980, July 2005.

[133] C. C. d. Souza *et al.*, "The Datapath Merging Problem in Reconfigurable Systems: Complexity, Dual Bounds and Heuristic Evaluation," *ACM Journal of Experimental Algorithmics*, vol. 10, p. 2.2–es, Dec 2005.

[134] M. Zuluaga and N. Topham, "Resource Sharing in Custom Instruction Set Extensions," in *Symposium on Application Specific Processors (ASAP)*, 2008, pp. 7–13.

[135] N. Pothineni *et al.*, "A High-level Synthesis Flow for Custom Instruction Set Extensions for Application-specific Processors," in *15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010, pp. 707–712.

[136] J. Coole and G. Stitt, "Intermediate Fabrics: Virtual Architectures for Circuit Portability and Fast Placement and Routing," in *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010, pp. 13–22.

[137] D. Grant *et al.*, "A CAD Framework for Malibu: An FPGA with Time-multiplexed Coarse-grained Elements," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2011, pp. 123–132.

[138] A. Brant and G. G. F. Lemieux, "ZUMA: An Open FPGA Overlay Architecture," in *IEEE 20th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012, pp. 93–96.

[139] D. Koch *et al.*, "An Efficient FPGA Overlay for Portable Custom Instruction Set Extensions," in *23rd International Conference on Field Programmable Logic and Applications (FPL)*, 2013, pp. 1–8.

[140] H.-C. Ng *et al.*, "Direct Virtual Memory Access from FPGA for High-Productivity Heterogeneous Computing," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 458–461.

[141] F. Hannig *et al.*, "Invasive Tightly-Coupled Processor Arrays: A Domain-Specific Architecture/Compiler Co-Design Approach," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 4s, pp. 133:1–133:29, 2014.

[142] H.-C. Ng *et al.*, "A Soft Processor Overlay with Tightly-coupled FPGA Accelerator," in *2nd International Workshop on Overlay Architectures for FPGAs (OLAF)*, 2016, pp. 31–36.

[143] J. Gray, "GRVI Phalanx: A Massively Parallel RISC-V FPGA Accelerator Accelerator," in *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 17–20.

[144] R. Ferreira *et al.*, "An FPGA-based Heterogeneous Coarse-grained Dynamically Reconfigurable Architecture," in *14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011, pp. 195–204.

[145] D. Capalija and T. S. Abdelrahman, "A High-performance Overlay Architecture for Pipelined Execution of Data Flow Graphs," in *23rd International Conference on Field Programmable Logic and Applications (FPL)*, 2013, pp. 1–8.

[146] A. K. Jain *et al.*, "Efficient Overlay Architecture Based on DSP Blocks," in *IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2015, pp. 25–28.

[147] J. Kingyens and J. G. Steffan, "The Potential for a GPU-Like Overlay Architecture for FPGAs," *International Journal of Reconfigurable Computing*, vol. 2011, pp. 514 581:1–514 581:15, 2011.

[148] S. Korf *et al.*, "Automatic HDL-Based Generation of Homogeneous Hard Macros for FPGAs," in *IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2011, pp. 125–132.

[149] C. Lavin *et al.*, "Improving Clock-rate of Hard-macro Designs," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 246–253.

[150] M. Gort and J. Anderson, "Design Re-use for Compile Time Reduction in FPGA High-level Synthesis Flows," in *2014 International Conference on Field-Programmable Technology (FPT)*, 2014, pp. 4–11.

[151] C. Mulpuri and S. Hauck, "Runtime and Quality Tradeoffs in FPGA Placement and Routing," in *ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, 2001, pp. 29–36.

[152] J. B. Goeders *et al.*, "Deterministic Timing-Driven Parallel Placement by Simulated Annealing Using Half-Box Window Decomposition," in *2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2011, pp. 41–48.

[153] Y. O. M. Moctar and P. Brisk, "Parallel FPGA routing based on the Operator Formulation," in *51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

[154] N. Shirazi *et al.*, "Automating production of run-time reconfigurable designs," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 1998, pp. 147–156.

[155] T. Frangieh *et al.*, "PATIS: Using Partial Configuration to Improve Static FPGA Design Productivity," in *IEEE International Symposium on Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW)*, 2010, pp. 1–8.

[156] H.-C. Ng *et al.*, "Reconfigurable Acceleration of Short Read Mapping with Biological Consideration," in *2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 229–239.

[157] I. Coleman *et al.*, "GeDi: applying suffix arrays to increase the repertoire of detectable SNVs in tumour genomes," *BMC Bioinformatics*, vol. 21, no. 45, pp. 1–12, 2020.

[158] H.-C. Ng *et al.*, "Acceleration of Short Read Alignment with Runtime Reconfiguration," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 256–262.

[159] ——, "ADAM: Automated Design Analysis and Merging for Speeding up FPGA Development," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 189–198.

[160] Illumina, Inc, "Paired-End vs. Single-Read Sequencing Technology - Illumina," 2021. [Online]. Available: https://emea.illumina.com/science/technology/next-generation-sequencing/plan-experiments/paired-end-vs-single-read.html

[161] M. Holtgrewe, "Mason - A Read Simulator for Second Generation Sequencing Data," *Technical Report FU Berlin*, 2010.

[162] W. Huang *et al.*, "ART: a next-generation sequencing read simulator," *Bioinformatics*, vol. 28, no. 4, pp. 593–594, Feb 2012.

[163] AWS, "Amazon EC2 F1 Instances," 2021. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1

[164] Illumina, Inc., "Illumina DRAGEN Bio-IT Platform," 2021. [Online]. Available: https://www.illumina.com/products/by-type/informatics-products/dragen-bio-it-platform.html

[165] D. Kim *et al.*, "Graph-based genome alignment and genotyping with HISAT2 and HISAT-genotype," *Nature Biotechnology*, vol. 37, pp. 907–915, 2019.

[166] R. Zhao *et al.*, "Hardware Compilation of Deep Neural Networks: An Overview," in *IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2018, pp. 1–8.

[167] ——, "Towards Efficient Convolutional Neural Network for Domain-Specific Applications on FPGA," in *28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 147–154.

[168] H. Fan *et al.*, "Reconfigurable Acceleration of 3D-CNNs for Human Action Recognition with Block Floating-Point Representation," in *28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 287–294.

[169] B. M. F. Chung *et al.*, "High-throughput microparticle screening by 1-µm time-stretch optofluidic imaging integrated with a field-programmable gate array platform," in *2016 Conference on Lasers and Electro-Optics (CLEO)*, 2016, pp. 1–2.

[170] H.-C. Ng *et al.*, "High-throughput Cellular Imaging with High-Speed Asymmetric-Detection Time-Stretch Optical Microscopy under FPGA Platform," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2016, pp. 1–6.

[171] M. Wang *et al.*, "Real-time Object Detection and Classification for High-Speed Asymmetric-Detection Time-Stretch Optical Microscopy on FPGA," in *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 261–264.

[172] A. Chan *et al.*, "All-passive pixel super-resolution of time-stretch imaging," *Scientific Reports*, vol. 7, no. 44608, 2017.