Title
SMEM++: A Pipelined and Time-Multiplexed SMEM Seeding Accelerator for Genome Sequencing

Author
Yu, Tianhe

Publication Date
2018

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
Los Angeles

SMEM++: A Pipelined and Time-Multiplexed SMEM Seeding Accelerator for Genome Sequencing

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Electrical Engineering

by

Tianhe Yu

2018

ABSTRACT OF THE THESIS

SMEM++: A Pipelined and Time-Multiplexed SMEM Seeding Accelerator for Genome Sequencing

by

Tianhe Yu
Master of Science in Electrical Engineering
University of California, Los Angeles, 2018
Professor Jingsheng Jason Cong, Chair

The advent of next-generation sequencing has made a great impact on many applications from precision medicine to new drug discovery. This motivates the research of FPGA acceleration for genome sequencing algorithms. The recently developed quadratic-time SMEM seeding algorithm becomes a time-consuming computation kernel in genome sequencing, but it has not been well studied. The fundamental challenge of accelerating the SMEM algorithm is to handle its large volume of random memory accesses. While the state-of-the-art SMEM accelerator attempts to solve this by sacrificing the performance of individual processing elements to maximize the task-level parallelism, this design methodology suffers a serious resource underutilization issue. To resolve this issue, I propose SMEM++, a pipelined and time-multiplexed FPGA accelerator for the SMEM algorithm. SMEM++ adopts the canonical non-blocking pipeline methodology and implements a fully pipelined processing element design with the initiation interval equal to one. Moreover, a communication interface adapter is designed to make the accelerator compatible to the designated FPGA platform interface and increase its portability. The experiments on the Intel HARPv2 platform show that SMEM++ outperforms the original software by 24x, and outperforms the state-of-the-art SMEM accelerator design by 6.3x, even with 43% less logic resource consumption.

The thesis of Tianhe Yu is approved.

Milos D Ercegovac

William J Kaiser

Jingsheng Jason Cong, Committee Chair

University of California, Los Angeles

2018

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First, I want to express my sincere gratitude to Professor Jason Cong for allowing me to conduct this research under his auspices. I am grateful for his confidence and the freedom he gave me to do this work and the guidelines given throughout the research process.

I would also like to express my gratefulness to my mother and father for the continuous support towards this degree. Also my sincere gratitude to Dr. Wei Peng on his extensive support and guidance during my MS degree, I will never go this far without the help from him. In addition, I would like to thank Licheng Guo, Dr. Wei Peng and Bug Huang for their contribution in this work. I also want to thank Weinan Song, Jessie Xu, Dr. Yuchen Hao, Zhenyuan Ruan, Jie Wang, Peipei Zhou, Cody Yu for all the help and support. It has been a joy working with you guys.

I would finally want to thank myself. Thanks for re-building confidence and making my life wonderful.

Part of this works appears in Proceedings of the 28th International Conference On Field Programmable Logic & Application. The work is supported by Intel's donation of HARPv2 system and under CDSC Industrial Partnership Program.

CHAPTER 1

Introduction

The next-generation sequencing (NGS) technology has stimulated an ever-increasing requirement for computation capabilities [1][2]. FPGA acceleration is considered a promising approach to address this requirement [3]. Genome sequencing involves both biochemical and computing phases [4]. The biochemical phase takes copies of genomes as input, fragments these genome copies into billions of small pieces, called *reads*[1], and sequences each *read*. The computing phase reassembles these discrete *reads* by aligning them onto a reference genome whose length is in the order of 10^9 basepairs. Searching on such a billion-basepair string for each of billions of *reads* makes it compute-intensive.

The computing phase for an individual *read* can be further decoupled into two steps [5]. The first step, *seeding*, finds candidate alignment locations on the reference genome by matching part of the *read* to these locations, called *seeds*; the second step, *extending*, extends the *seeds* forward and backward to align the entire *read* to each of the candidate locations. For the extending step, the Smith-Waterman algorithm [6] is predominantly adopted, and its acceleration problem has been extensively studied. However, the algorithm for performing seeding is still constantly evolving. This work is devoted to the acceleration of the newly proposed super-maximal exact match (SMEM) seeding algorithm that is adopted by the state-of-the-art BWA-MEM *read* aligner [7]. Compared to the conventional linear-time seeding algorithm that features FM-index based backward searching [8], the SMEM algorithm instead adopts *FMD-index* to perform both forward and backward, i.e., *bidirectional*, searching, resulting in a *quadratic* time complexity [9]. Although the acceleration of the backward searching algorithm is well studied [3], the aforementioned algorithmic

[1]This terminology is somewhat confused with the memory "read" access. Throughout the paper the italic *read* is used to refer to the short genome sequence, so as to distinguish from the memory read access.

differences do bring new challenges to FPGA acceleration and motivate this study.

In general, the acceleration of the SMEM algorithm faces a classic design challenge: the need of hiding the long off-chip memory access latency, since the algorithm generates a tremendous number of random off-chip accesses. Chang et al.'s design [10] remains the only FPGA acceleration effort for the quadratic-time SMEM algorithm. Their proposed accelerator features an array of parallel processing elements (PEs) that simultaneously send off-chip requests to hide the off-chip latency. The PE design is greatly simplified to process one *read* at a time and go through all the *read* processing steps sequentially, which leads to the maximization of the number of PEs and extensive exploration of the high degree of task-level parallelism inherent in genome sequencing. However, this design principle suffers a severe resource underutilization issue because when a PE is waiting for off-chip memory responses, it entirely lies idle without performing useful computation. In fact, experiments show that a PE spends 56.2% of the overall execution time idling for memory responses. This also results in the phenomenon that the replication of PEs rapidly exhausts the FPGA on-chip resources before the off-chip bandwidth is fully explored. Specifically in [10], the FPGA fabric can only hold at most 16 PEs, utilizing only 35% of the platform's off-chip bandwidth.

To resolve this resource underutilization issue, a new FPGA accelerator for the SMEM algorithm is proposed in this work, named *SMEM++*, by adopting a different design methodology—the non-blocking pipeline methodology—to hide the off-chip latency. Being initially proposed in the non-blocking cache design [11] and adopted by every modern processor, the non-blocking pipeline methodology temporarily stores outstanding memory requests in a FIFO structure (e.g., miss status holding registers (MSHRs)) instead of letting them block the computation pipeline. The computation pipeline thus remains non-blocking for other jobs, and the outstanding off-chip transactions in the FIFO overlap together to hide the off-chip latency. This canonical methodology is also adopted by the FPGA community for the acceleration for applications with extensive random off-chip accesses. For example, Arram et al. [12] has used this methodology to accelerate the aforementioned linear-time backward searching algorithm. While the SMEM++ accelerator shares with these previous studies the same big picture, the key question becomes how to achieve the optimal efficiency of the accelerator design, i.e., making the initiation interval equal to one ($II = 1$) with a highly

complex computation kernel. Specifically, the SMEM++ design faces and resolves the following challenges:

1) Off-chip memory structural hazard. The bidirectional SMEM algorithm consists of a linear-time forward phase and a quadratic-time backward phase, both of which iteratively sends random off-chip requests. In each forward or backward iteration, two 512-bit off-chip requests are sent out. If one wishes to process one forward and/or backward iteration per cycle, she or he has to face the challenge of performing multiple 512-bit off-chip requests every cycle, which often leads to structural hazard since many state-of-the-art FPGA platforms, e.g., Intel HARPv2 [13], supply only 512-bit data width for off-chip transactions.

2) On-chip memory structural hazard. The backward phase iteratively retrieves and updates the intermediate data generated by the forward phase. As a result, the on-chip memory that stores such intermediate data is going to be read/written simultaneously by both phases, leading to potential structural hazard due to the port number limitation.

3) Memory channel congestion. An $II = 1$ pipeline requires a large random off-chip access bandwidth (25.6 GB/s in our case), which surpasses the maximum bandwidth most FPGA platforms could supply. One must face the challenge of correctly handling inevitable memory channel congestion, i.e., properly stalling and recovering the entire pipeline. This challenge becomes more serious since the SMEM++ accelerator contains a very deep pipeline with many affiliate modules.

To address these challenges, I identify the similarities and differences between the forward and backward phases, and implement a unified pipeline that can process a *read* in either forward or backward way. As a result, the pipeline achieves the throughput of processing one forward or backward iteration per cycle. Since the forward and backward phases have different computation load ($O(N)$ vs $O(N^2)$), this unified pipeline achieves a more efficient resource utilization compared to the one with separate forward and backward stages. Also, the unified pipeline requires at most one retrieve and update operations on the intermediate data, which resolves the structural hazard in Challenge #2.

While the unified pipeline simply processes one basepair iteration and two 512-bit off-chip requests per cycle, I propose a communication interface adapter based on the time-multiplexing

and asynchronous FIFO techniques to address Challenges #1 and #3.

A proof-of-concept implementation based on the Intel HARPv2 platform is built, which provides one read port at 400 MHz and connect the 200 MHz SMEM++ design that requires two read ports to this platform. With the proposed adapter as a bridge between the SMEM++ design and underlying platform, the accelerator can fully utilize the supplied bandwidth and properly stall and recover during memory channel congestion.

In summary, the contributions of this work is as follows:

- A non-blocking pipeline design that achieves a maximal throughput of processing one forward or backward alignment per cycle.
- A communication interface adapter that adopts the time-multiplexing and asynchronous FIFO to make our design compatible with various communication interfaces.
- An early study that demonstrates the use of the Intel HARPv2 platform for application acceleration.

The experiments show that the proposed design achieves 87.5 Mbp/s (million basepairs per second, see Chapter 2) processing throughput, which outperforms the best existing design [10] by 6.3x and outperforms single-thread CPU execution by 24x. Also, this design consumes even 43% less logic resource compared to [10], meaning that the proposed design improves the overall resource efficiency by 11x.

CHAPTER 2

Background

2.1 BWA-MEM and SMEM algorithm

BWA-MEM is a software package for mapping genome sequences against a large reference genome, such as the human genome [7]. This work focuses on accelerating the seeding function, which finds all the super maximum exact matches (SMEMs) of a *read*. A super maximum exact match between the *read* and the reference string [7] is an longest exact match that can not be further extended and also not contained by the other exact matches . Because this seeding algorithm takes about 30% of total execution time in this software flow [10], acceleration on this would be beneficial.

The SMEM algorithm is based on Burrows Wheeler transform(BWT) [14] and FM-index [15]. BWT of a string is the last column of the lexicographically sorted all rotations of the string T. Its relation can be described with the suffix array in the following formula. The T is the original string and $SA[i]$ represent the rank of the first character (in the original string) of ith element in the sorted suffix array. Fig.2.1 shows a construction of BWT string, where the left box shows all the rotations of the string and right box shows the lexicographically sorted suffix array and the last column as BWT string c.

$$BWT[i] = \begin{cases} T[SA[i]-1] & \text{if} SA[i] > 1 \\ \$ & \text{else} \end{cases}$$

Given the string T, also define the count array $C(a)$ as the number of symbols in T that are lexicographically smaller than a, and the occurrence array $O(a, i)$ as the occurrence of the symbol a before the ith element in BWT. The FM-index querying is to match the element in BWT string

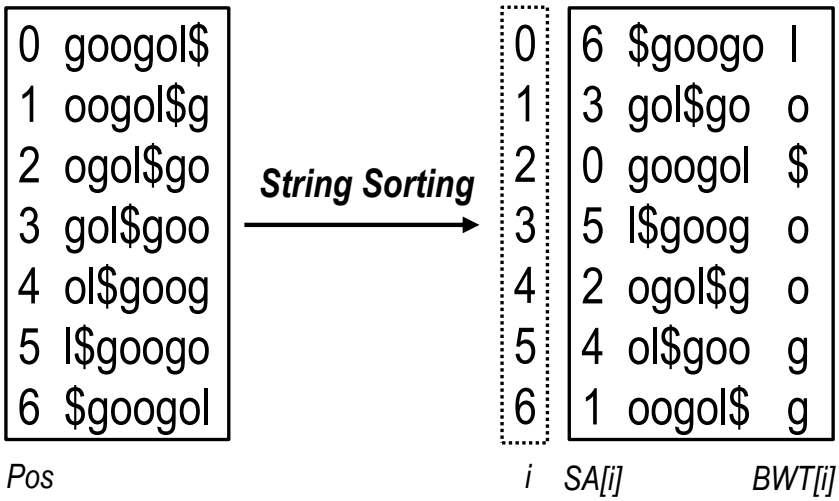

Figure 2.1: Example of BWT Construction for a String

back to its suffix array element. Specifically, it can be expressed in the following equations:

$$I^l(aP) = C(a) + O(a, I^l(P) - 1)$$

$$I^u(aP) = C(a) + O(a, I^u(P)) - 1$$

where P is a substring of the reference string and $I^l(aP)$ and $I^u(aP)$ represent string aP's first and last appearance as prefix in the suffix array of the reference string. aP is a substring of the reference string only when $I^u(aP) > I^l(aP)$. After a one-time transformation of the reference string, one will be able to find the exact match of the query string in $O(N)$ time where N is the length of the query string. Because of the pattern of querying the perpending character a on P, It only allows uni-directional search in FM-index. Fig.2.2 shows an example of querying a string of "og" with the previously constructed BWT information. The left square bracket is $I^l(aP)$ and the right round bracket is $I^u(aP)$ in each step.

The SMEM algorithm [9] however, uses the FMD-index, which is based the FM-index but enables bidirectional querying. It distinguishes from the FM-index based backward searching algorithm [8] in the following three aspects:

1) Different encoding and update methods. The FM-index encodes the match positions into an interval $(I^l(aP), I^u(aP))$, and specifies a pair of equations to update it. In the SMEM algorithm, however, the reference becomes the concatenation of the reference genome and its reverse com-

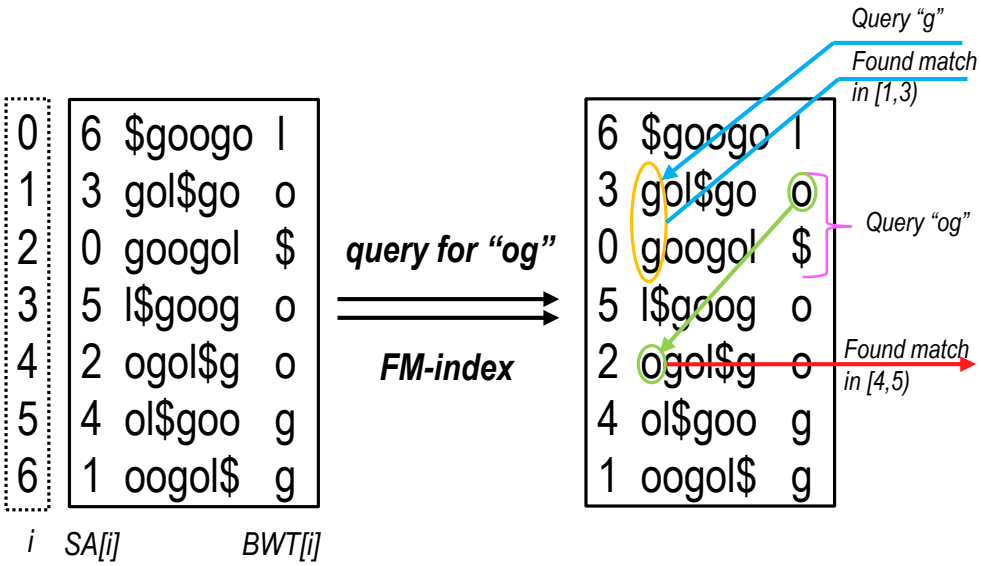

Figure 2.2: Example of FM-index Querying for String "og"

plement. The occurrences of a seed are encoded into a bi-interval $(I^l(P), I^l(\bar{P}), I^s(P))$, where $I^l(P)$ and $I^l(\bar{P})$ refer to the first occurrence of the seed and its reverse complement, respectively, and $I^s(P)$ refer to the interval size (number of matches). Subsequently, the FMD-index has more complex computation process for updating the intervals. *2) Bidirectional and quadratic-time complexity.* While the FM-index based algorithm always searches from the last string character back to the first, the FMD-index technique searches bidirectionally. Since the backward phase examines all the intervals generated in the forward phase, the algorithm complexity increases from $O(N)$ to $O(N^2)$. *3) Different memory access patterns.* In the FM-index algorithm, each update of the interval requires two 64-bit off-chip data; instead, each FMD-index bi-interval update requires two 512-bit off-chip data. In other words, the SMEM algorithm has a higher off-chip bandwidth requirement.

The detailed algorithm is presented in Algorithm 1. It takes a read R and a position index x as inputs, and starts to find SMEMs from a one-basepair substring $R[x : x]$. First, the algorithm performs the *forward* phase which extends the substring rightward (forward) one basepair at a time. For each step of extension, the algorithm locates all the positions on the reference genome that exactly match the current substring. This forward extension ends when the extended substring grows longer and finally finds no more matches on the reference genome. Next, the algorithm per-

forms the *backward* phase which iteratively examines each interval generated during the forward phase. Since each interval corresponds to a substring starting from $R[x]$, the algorithm extends this substring leftward (backward) from $R[x-1]$ until no match of the extended substring can be found on the reference genome. Assuming an interval that corresponds to the forward position j ends its backward extension at the position i, the substring $R[i:j]$ is a SMEM if it is not fully covered by any previously found SMEM. All the identified SMEMs are stored in an output queue that is returned as the output of the algorithm. Fig.2.3 shows an example of querying a read on the reference genome. From the start position, it first align to the right until "AGTC". Then it align to the left each of the four temporary result. "GTTAGA" and "GTTAGAG" is abandoned in the last iteration because it is a substring of "GTTAGAGT". The final results are "GTTAGAGT" and "TAGAGTC".

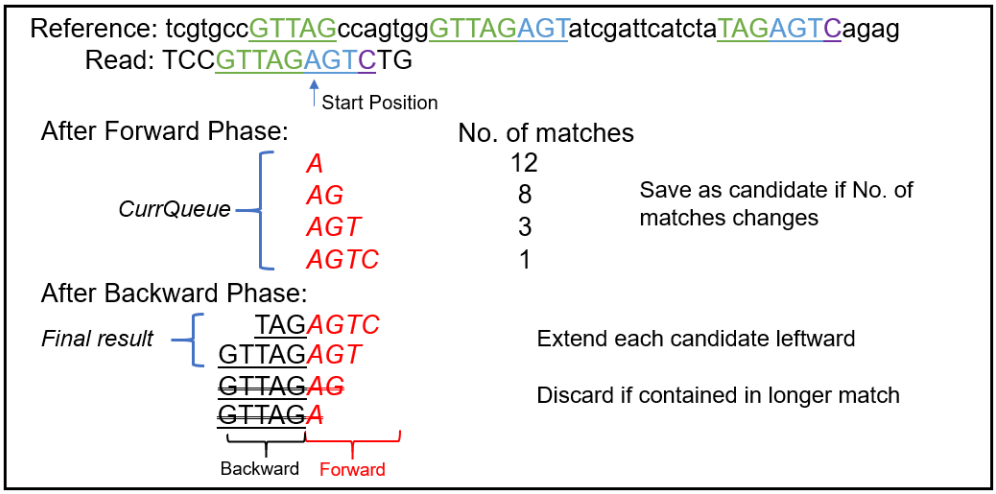

Figure 2.3: Example of SMEM algorithm

The basic operation (bwt_extend in the algorithm) is that of extending one basepair forward or backward. Inside the basic operation, two random access requests are sent to load two 512-bit data from off-chip memory to the FPGA, followed by a series of arithmetic/logic calculations on the loaded data.

The performance of the SMEM algorithm is then able to be measured by the number of base-pairs processed in a unit time. Throughout this work, the term *million basepairs per second (Mbp/s)*, i.e., the number of million basepairs processed in a second, will be used to measure

Algorithm 1 SMEM Seeding Algorithm

Input: Read R, Start position x

Output: All SMEM intervals containing the base pair: $R[x]$

Initialize $CurrQueue$ storage
$CurrPtr \leftarrow 0$
for $i = x + 1$ to $|R| - 1$ **do**
    $newIntv = bwt\_extend(lastIntv, R[i], Forward)$
    if $newIntv$!= $lastIntv$ **then**
        $CurrQueue[CurrPtr] \leftarrow lastIntv$
        $CurrPtr \leftarrow CurrPtr + 1$
        $lastIntv = newIntv$
    end if
end for
if $i = |R|$ **then**
    $CurrQueue[CurrPtr] \leftarrow newIntv$
    $CurrPtr \leftarrow CurrPtr + 1$
end if

$BackwardPtr \leftarrow 0$
$ForwardPtr \leftarrow CurrPtr - 1$
for $i = x - 1$ to -1 **do**
    $size \leftarrow 0$
    for $j = ForwardPtr$ to $BackwardPtr$ **do**
        $newIntv = bwt\_extend(CurrQueue[j], R[i], Backward)$
        if No More Match & No Longer Matches **then**
            Push $CurrQueue[j]$ to $OutputQueue$
        else $newIntv$!= $lastInterval$
            $CurrQueue[ForwardPtr - size] \leftarrow newIntv$
            $size \leftarrow size + 1$
        end if
    end for
    if $size == 0$ **then**
        break
    end if
    $BackwardPtr \leftarrow ForwardPtr - size + 1$
end for

return $OutputQueue$

the performance of the proposed SMEM accelerator and compare it with previous work.

2.2 Field Programmable Gate Array and Domain Specific Computing

As conventional CPU system start to slow down on improving its performance because of dark silicon and the end of Moore's law, people has been focusing on domain-specific customization on computation to bring orders-of-magnitude power-performance efficiency improvement [16]. Field Programmable Gate Arrays [17] are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs are able to be reprogrammed to different functionality after manufacturing. Because of its high reconfigurability, FPGA is considered one of the promising device for domain-specific acceleration.

Intel's Heterogeneous Architecture Research Platform (HARP) [18] family represents the state-of-the-art technology advances on CPU-FPGA platforms. The first generation, HARPv1, connects a Xeon CPU with a Stratix 5 FPGA via the QuickPath Interconnect (QPI) interface, and supplies a coherent, shared memory model. This improves the data communication efficiency between CPU and FPGA and achieves a higher communication bandwidth compared to traditional PCIe-based FPGA platforms [19]. The second generation, HARPv2, further improves the CPU-FPGA communication bandwidth by bringing the CPU and the FPGA into a single semiconductor package, and connecting them through three physical communication channels. Specifically, it remains the coherent QPI channel and adds two PCIe channels to offer extra bandwidth. The experiments show that HARPv2 improves the CPU-FPGA communication bandwidth by roughly 3x.

Fig. 2.4 illustrates the overall HARPv2 system. The FPGA logic is divided into two parts: the Intel-provided FPGA interface unit (FIU) and the user accelerator function unit (AFU). FIU provides platform capabilities such as unified address space, coherent FPGA cache and partial reconfiguration of user AFU, in addition to implementing interface protocols for the three communication channels. The core cache interface (CCI-P) is exposed to the accelerator designer with one read channel and one write channel, both of which are at 400 MHz and of 512-bit data width. Also, an optional memory properties factory (MPF) component in Basic Building Blocks (BBB) is supplied to provide higher-level memory services and semantics, including virtual addressing and

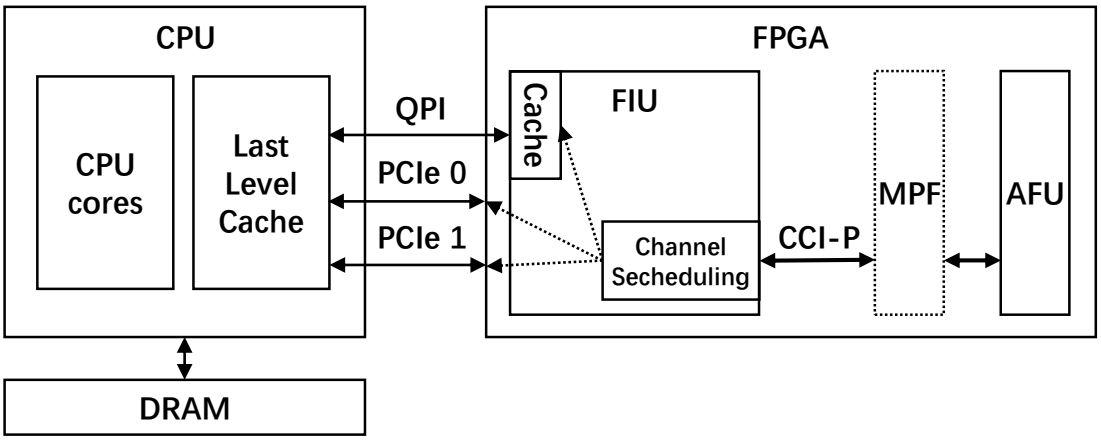

Figure 2.4: Intel HARPv2 System Overview

memory response ordering.

In this work, HARPv2 is used as the experimental platform since its high memory bandwidth is suitable for demonstrating the proposed design. However, the proposed design is not limited to any specific platform.

2.3 Existing Works

Many previous studies [20][21][12][22] have been devoted to the FPGA acceleration of various seeding algorithms. [21] accelerates the BFAST sequencing algorithm that uses a *read*'s k-mer (all of a string's substrings of length k) as seeds. The search engine to find candidate locations on the reference genome is based on hashing and filtering algorithms, as opposed to FM-index or FMD-index. [20] is the first work that accelerates the FM-index based algorithm on FPGA with the assumption that the whole reference data can be stored on FPGA. However, this is not practical for genome sequencing because the size of the reference genome are at least in size of gigabytes. [12] stores the reference genome on DRAM and brings the canonical non-blocking pipeline methodology into the acceleration of the FM-index based algorithm. While both this work and [12] share the same basic idea with the non-blocking cache design in 1981 [11], the algorithmic changes lead to multiple new challenges to achieve an $II = 1$ design for the SMEM acceleration. Specifically, the design in [12] does not need to face Challenge #1 since 1) the FM-

index algorithm has a lower off-chip bandwidth requirement (two 64-bit data per cycle), and 2) the underlying FPGA platform they use, Maxleler MAX3, supplies multiple memory ports which naturally avoids structural hazard. Also, since the FM-index algorithm performs only one-time linear pass to the input *read*, there is even no intermediate data that needs to be stored (or only two 64-bit registers for the intervals). Hence, the design does not need to bother Challenges #2 as well.

Some studies attempt to address the same quadratic-time algorithm as SMEM, e.g., [22] and [10]. However, the former only optimizes the SMEM algorithm in the software level without hardware acceleration; the latter suffers the serious resource underutilization issue which motivates us to adopt the non-blocking pipeline methodology.

CHAPTER 3

Fully Pipelined SMEM++ Accelerator

Beginning with this section, the details of SMEM++ accelerator will be presented. Fig. 3.1 illustrates the overall architecture of SMEM++. It consists of two major components:

Non-blocking Pipeline with II=1. The pipeline realizes full functionality of the SMEM algorithm. It is composed of three main modules: *pipeline module*, *storage module*, and *control module*. Specifically, the *pipeline module* performs the core computation to realize the bi-directional basepair extension of FMD-index. The *storage module* keeps the needed data during process. The *control module* handles the stall and recovery of the pipeline execution during memory channel congestion. This component mainly addresses Challenge #2 in Chapter 1.

Communication Interface Adapter. The adapter bridges the pipeline interface with the underlying FPGA platform interface. It improves the portability of the accelerator so that it can be ported elsewhere by just reconfiguring the adapter. This component mainly addresses Challenge #1 and #3 in Chapter 1. The non-blocking pipeline design is presented in this section, and leave the adapter design to Section 3.4.

3.1 Pipeline Module

The pipeline module presents a non-blocking hardware pipeline for the basepair extension operation. It consists of three components: *interval generation*, *extension control* and *address calculation*, corresponding to different steps of the basepair extension operation (see Fig. 3.1). As is mentioned in Chapter 1, the proposed pipeline supports both forward and backward extensions. The following sections present the detailed approaches to build this unified pipeline.

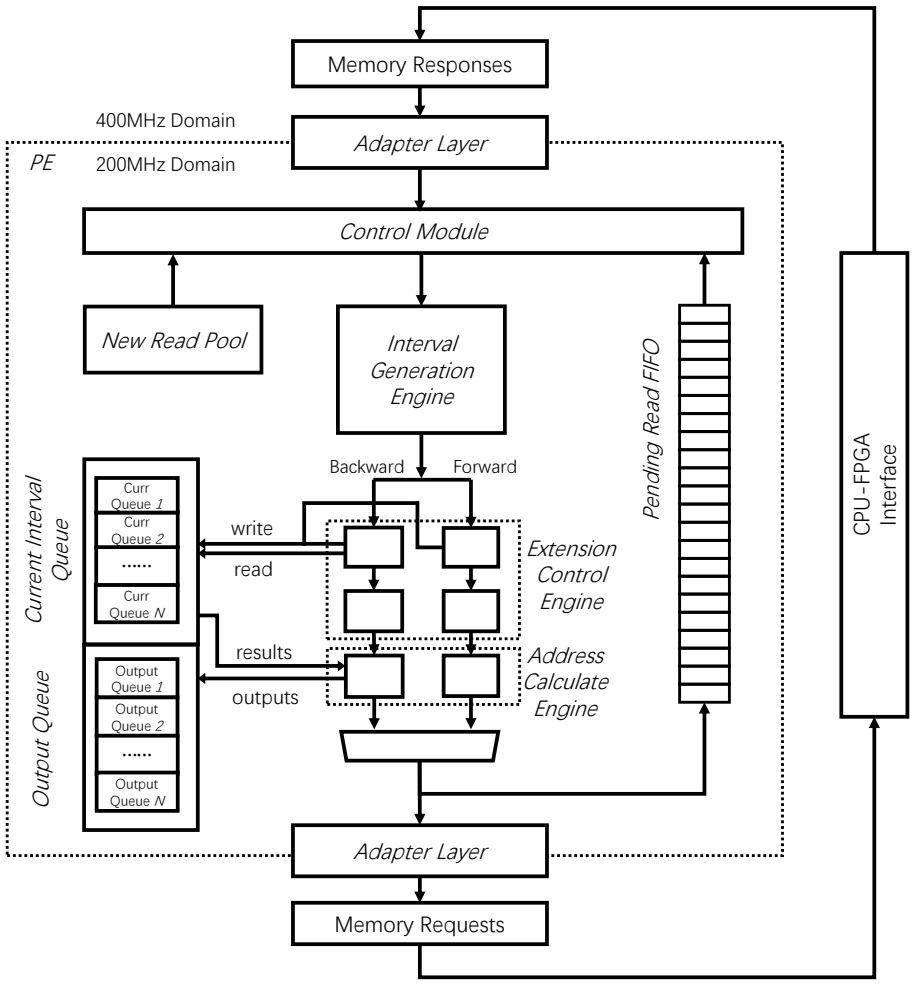

Figure 3.1: SMEM++ Accelerator Architecture Overview

3.1.1 Function Module Reordering

Originally in the SMEM software, the extension operation can be divided into the following four steps:

- ***Step 1*** takes a FMD-index bi-interval and extend direction as input and calculates the two table reference addresses respectively to extend the interval upper/lower bounds.
- ***Step 2*** performs off-chip accesses to retrieve the data.
- ***Step 3*** decodes and processed the retrieved data, then calculates the forward/backward extension candidate results.

- ***Step 4 (Forward)*** decides whether further alignment is possible and stores valid match in a temporary queue.
- ***Step 4 (Backward)*** decides whether all the matches in the temporary queue has finished one basepair extension and stores the result either back in temporary queue or the final output queue.

When mapping to hardware, the uncertain memory access latency in between of the certain-latency operations inhibits it to be fully pipelined. Thus the order of the four steps is shifted and Step 2 is moved to the end of the pipeline such that all fixed-latency operations are grouped together:

- ***Step 3*** $\rightarrow$ *Interval Generation*
- ***Step 4*** $\rightarrow$ *Extension Control*
- ***Step 1*** $\rightarrow$ *Address Calculate*
- ***Step 2*** $\rightarrow$ Last stage of the pipeline

After executing step 2, the computation information of the current processing *read* are stored into a FIFO just like the MSHR in non-blocking caches while the pipeline keeps processing other *read*s. In this way, the variable memory access latency will be covered by continuous processing of other reads. The memory response data is then feed to the pipeline for further alignment.

3.1.2 Unifying Forward/Backward Extensions

The unified pipeline design is based on the reordered execution flow. The *interval generation* component corresponds to the shared `bwt_extend` function in Algorithm 1, and is thus identical to both forward and backward phases. Meanwhile, the *extension control* and *address calculation* components handle the two phases differently.

The *interval generation* component corresponds to the core extension operation of FMD-index. It processes the memory responses through 12 pipeline stages. In detail, part of the off-chip data

are used as index to look up a pre-calculated table. The accumulation of the retrieved data, together with the other part of the off-chip data, are used to produce the lower and upper bounds of four types of nucleotides. Finally with the accumulation of other on-chip data, the bi-intervals to align all 4 types of nucleotides are generated, as shown in Fig. 3.2.

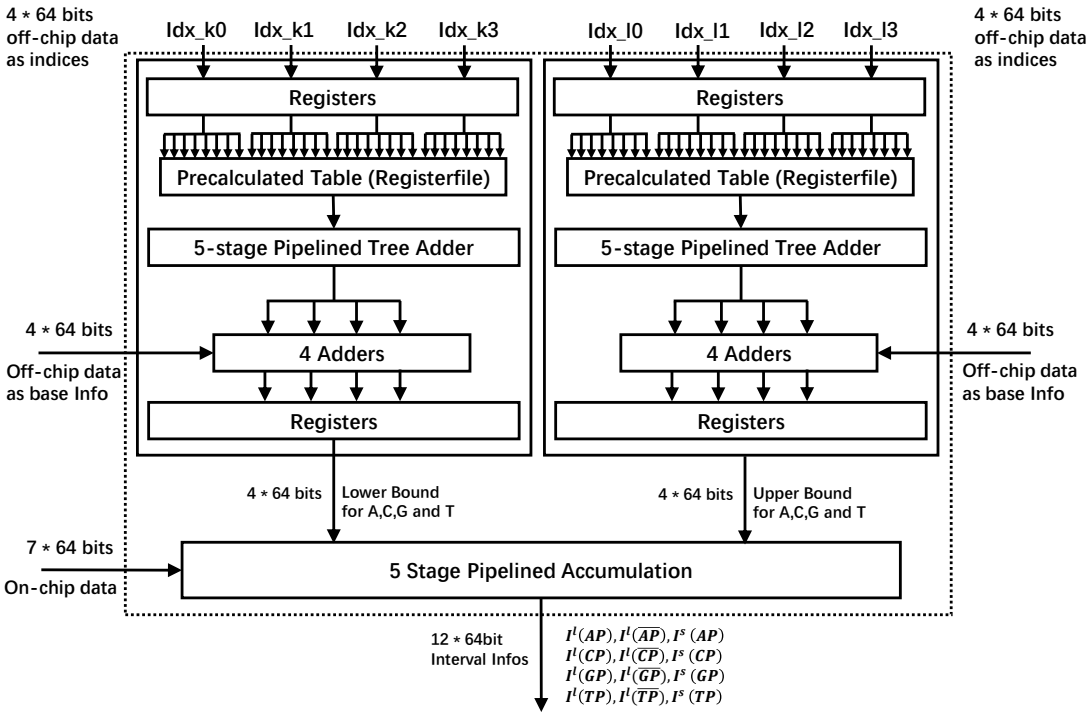

Figure 3.2: Interval Generation Architecture

The *extension control* and *address calculation* components correspond to the control flow of each loop, which are different between two phases. To avoid structure hazard due to the limited BRAM write ports, all memory access operations are scheduled to be in the same level of the two branches in pipeline. At any stage, only one of the two branches is active, which eliminates the possible port contention. Bypassing logic is used when candidates are to be stored and fetched in the same cycle. In addition, some empty stages are inserted between *extension control* and *address calculation* to meet the multi-cycle delay when accessing the *Current Interval Queue* (see section 3.2).

3.2 Storage Module

The storage module stores the input, output and the intermediate extension results during the processing. Specifically, the module consists of the following four components:

1) New Read Pool stores the input *reads* to the accelerator. A large number of *reads* are loaded and processed onto the FPGA at a time to efficiently utilize the pipeline. Compared to the execution time, the input loading time is negligible.

2) Pending Read FIFO stores the critical informations of *reads* that are waiting for their memory responses. The critical information includes read ID, input parameters for alignment, temporary generated data for further alignment and read/write addresses for temporary storage, etc. As the memory response are reordered to be in the same order as memory request, the earliest arrived memory response always belongs to the read at the head of the FIFO. Together with its memory responses, the read information will then be re-issued into the pipeline for the next round of extension.

3) Current Interval Queue. The current interval queue stores the intervals that are generated throughout the forward phase, and used and updated by the backward phase. This queue structure corresponds to the `CurrQueue` variable in Algorithm 1. The *extension control* component of the pipeline generates new entries or updates existing entries during the forward or backward phases, respectively.

The SMEM algorithm alternatively uses two large queues (approximately 10M bits each) for updating the generated intervals in the backward phase. Directly writing to this large address space without buffering will lead to setup time violations. To resolve this, multi-level bus are deployed to reduce the fanout in each stage when connecting the memory blocks. This on the other hand, results in multi-level delay when accessing data, which is then handled by adding empty stages in the pipeline.

4) Output Queue. The output queue stores all generated SMEMs that are encoded as integral intervals, which correspond to the `OutputQueue` variable in Algorithm 1. These are the final outputs of the entire SMEM accelerator.

3.3 Control Module

The control module handles the stall and recovery of the pipeline execution. When the pipeline is fully operational, it will require a bandwidth of 25.6 GB/s, which exceeds the maximum bandwidth of many FPGA platforms. Therefore, memory channel congestion constantly occurs, during which the pipeline should stop sending out memory requests.

In general, the off-chip communication interface of FPGA platform will provide a `congest` signal to indicate the congestion status. When the signal becomes active, user are requested to stop sending memory requests. this `congest` signal is broad-casted to all stages of the pipeline to stop them and preserving the current state.

However, this approach leads to a huge fanout of the `congest` signal, resulting in timing violation. To resolve this, the `congest` signal is recursively duplicate and registered until the fanout requirement is met. While this approach successfully resolves the timing violation issue, it results in a multi-cycle delay from the cycle when the `congest` signal becomes active to the cycle when the pipeline execution is actually stalled. Any memory requests sent during these cycles may be lost. This issue is addressed in the communication interface adapter design, which will be described in detail in the following section.

3.4 Communication Interface Adapter

This section presents the issues and solutions in integrating the proposed fully pipelined accelerator into various FPGA platforms. Section 3.4.1 analyzes the issue when porting an accelerator to different platforms. Section 3.4.2 describes the communication interface adapter implemented for HARPv2 as an example that demonstrates the approach for the portability issue.

3.4.1 Portability of Accelerator

When the pipeline is fully occupied and works at 200 MHz, the proposed PE design requires two memory read ports of at least 512-bit width to achieve its optimal throughput of processing one basepair per cycle, i.e., 200 Mbp/s. If more read ports of 512-bit width are available, our accelerator

can achieve even higher performance by replicating the PE design multiple times. However, not all existing FPGA platforms align with this interface requirement. For example, our experimental platform, Intel HARPv2, supplies a 400 MHz off-chip communication interface with only one 512-bit read port. Worse still, the interfaces of different platforms from different vendors vary from each other, further limiting the portability of the accelerator design.

This issue motivates the design of a communication interface adapter between the accelerator memory access interface and the vendor-provided data communication interface. To port the accelerator onto a certain platform, one only need to adjust the configuration of the adapter design to make the accelerator compatible with the target platform, which greatly improves the accelerator portability. In the following section, the Intel HARPv2 platform is used as an example that demonstrates how the adapter works.

3.4.2 Adapter Implementation

Fig. 3.3 illustrates the overall architecture of the adapter design. The adapter contains a request channel and a response channel to handle both the requests and responses.

1) Request channel. The request channel employs an asynchronous FIFO for clock-domain crossing. This FIFO writes two entries and reads one entry at a time. Recall that the pipeline sends two memory requests at each 200 MHz clock cycle. The adapter then collects one request in the odd cycle and the other in the even cycle, both at the 400 MHz clock domain. This time-multiplexing implementation successfully adapts the requirement of two reads per 5ns to one 400 MHz read port without performance degradation.

Moreover, the request channel utilizes the FIFO to resolve the issue due to the broadcasting delay of the `congest` signal, as mentioned in Section 3.3. During normal pipeline execution, this FIFO stays empty as the written-in requests will be immediately read out. When the `congest` signal is set, the FIFO temporarily buffers the requests without actually sending out off-chip requests. The maximal number of requests that need to be buffered are determined by the delay from the `congest` signal set to the pipeline is actually stalled. When the memory channel congestion is resolved, the FIFO releases the buffered requests while still stalling the pipeline execution. Once

all the buffered requests are sent out and the FIFO becomes empty again, the FIFO will release the stall to the pipeline.

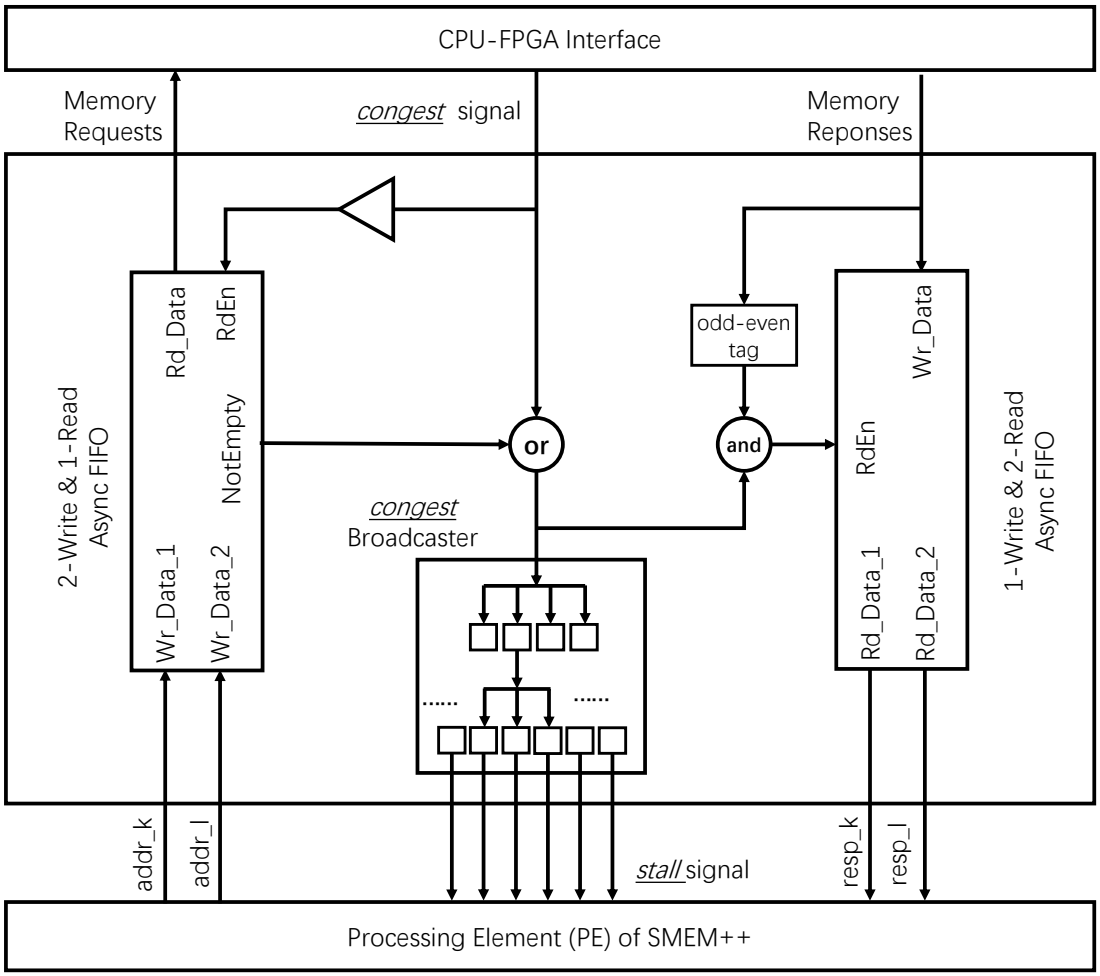

Figure 3.3: Communication Interface Adapter Architecture

2) Response channel. In contrast to the request channel, the response channel employs a "1-write, 2-read" asynchronous FIFO to transfer the response data to the processing element (PE) clock domain. Since the PE requires two responses at a time, the responses are read out of the FIFO pair by pair. To meet this requirement, the response channel maintains a one-bit odd-even flag at the 400 MHz clock domain to record whether the current response is the first or the second one in the pair. Once a pair of responses are both received, the response channel assembles the two 512-bit data together and sends them to the PE where it is then written back to the PE's pending read FIFO (see Section 3.2). In case the pipeline stalls when the responses arrive, the

response FIFO contains enough entries—equal to the maximal number of outstanding reads in the pipeline—so it will never run out of space.

In summary, how the adapter bridges the accelerator interface with that of the underlying FPGA platform is detailedly described here. While Intel HARPv2 is used for the demonstration purpose, the adapter design methodology is applicable for other state-of-the-art FPGA platforms. Specifically, platforms like the Alpha Data board [23], IBM CAPI [24] and Amazon F1 instance [25] supply accelerator designers with 512-bit memory ports, which aligns with the accelerator interface very well. The frequency difference of different interfaces can then be addressed by the asynchronous FIFOs employed in the adapter design. Therefore, the adapter design greatly eases the burden of porting our accelerator across platforms.

CHAPTER 4

Experimental Evaluation

This section presents the experimental evaluation to the proposed accelerator design. The experimental setup are first described (Section 4.1). Then the overall performance and resource consumption of the proposed design compared with those of Chang et al.'s design (Section 4.2) will be showed. Furthermore, a deeper look at the pipeline execution is given, with a key focus on the pipeline efficiency (Section 4.3).

4.1 Experimental Setup

Here demonstrates the proposed accelerator design using the Intel HARPv2 platform (see Section 2). The processing element works at 200 MHz and the communication interface adapter bridges it with the 400 MHz CCI-P interface. From the software side, the batch processing methodology used in [10] is deployed to combine and process a large number of reads at a time. In later text the number of reads in a batch will be referred as *batch size*. As will be discussed in Section 4.3, the batch size is a key parameter to the pipeline efficiency.

4.2 Overall Performance and Resource Utilization

Table 4.1 lists the on-chip logic (ALMs) and RAM resource consumption under various batch sizes. We can see that the logic resource consumption is nearly constant between different batch sizes, and the RAM consumption grows significantly as the batch size increases. This is because a read batch with a larger size requires more RAM resource to store the increased input, intermediate data, and output. On the contrary, the SMEM++'s pipeline module remains the same, leading to

negligible changes on the logic resource consumption. Looking into Table 4.2, it is clear that our pipeline module takes only 10% of the total ALM used whereas Intel HARP interface consumes most of the logic resources.

Batch Size	Total ALMs	Total RAM Bits
64	94,366 (22%)	3,616,494 (7%)
128	94,389 (22%)	4,923,833 (9%)
256	94,396 (22%)	7,538,756 (14%)
512	94,546 (22%)	12,769,103 (23%)
1024	94,874 (22%)	23,238,733 (42%)

Table 4.1: Resource Consumption of SMEM++

Modules & sub-Modules		Total ALM used
Pipeline Module		9,241
	Interval Generation	7,605
	Other Steps	1,636
Pending Read FIFO		919
Curr and Output Queue		480
Communication Interface Adapter		4,546
Intel Interface Module		79,203
Total		94,389

Table 4.2: Detailed Resource Consumption of SMEM++ with Batch Size 128

Moreover, the port of Chang et al.'s design with 16 PEs on HARPv2 platform is used as the baseline. The experiments show that this baseline design consumes 164,273 ALMs, meaning that SMEM++ consumes merely 57% as much logic resource as Chang et al.'s design.

Fig. 4.1 illustrates the performance of SMEM++ in million basepairs per second (Mbp/s), along with the off-chip bandwidth usage. The results show that SMEM++ achieves a throughput of 87.5 Mbp/s and utilizes 11.2 GB/s off-chip bandwidth, outperforming Chang et al.'s design by 6.3x with 43% less logic resource, meaning that the proposed design improves the overall resource efficiency by 11x. Moreover, SMEM++ outperforms the single thread Xeon E5 2680 by 24x.

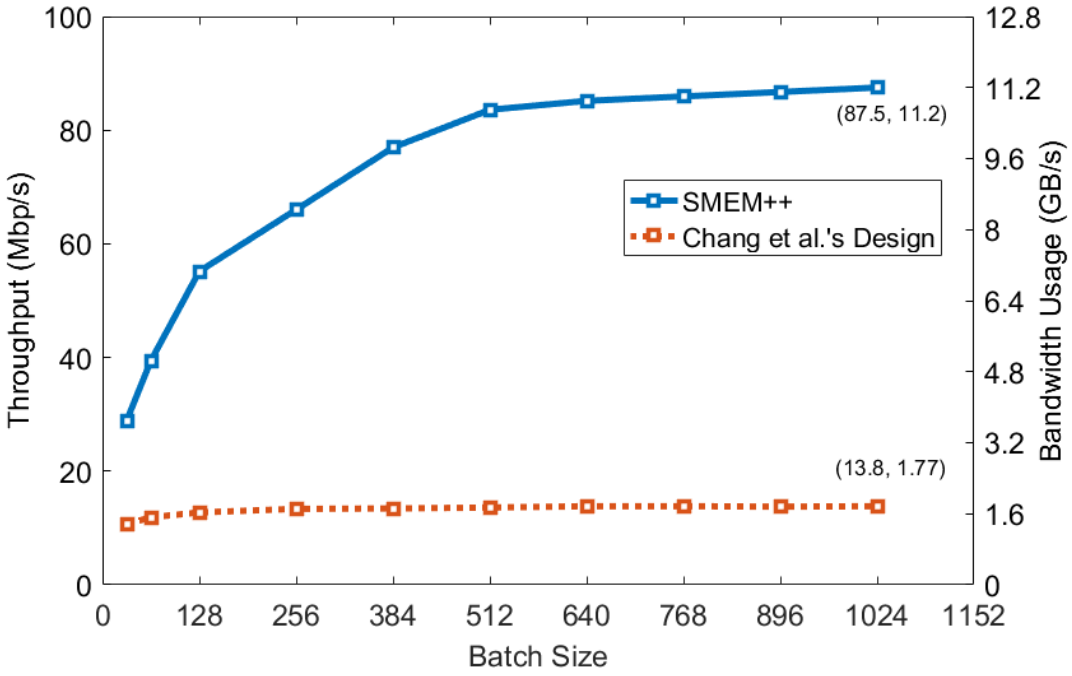

Figure 4.1: Overall Performance and Off-Chip Bandwidth Utilization Compared to Chang et al.'s Design

We can also see that the SMEM++ accelerator favors a large batch size. A batch size as large as 1024 is trivial to achieve for the *read* alignment application since the application contains billions of *reads* to be processed independently. In the following section, we go deep into the pipeline execution to analyze how a large batch size will improve the pipeline efficiency.

4.3 Pipeline Efficiency Analysis

In theory, if the SMEM++ pipeline is fully occupied and never stalls, it can achieve a 200 Mbp/s throughput and utilize a 25.6 GB/s off-chip bandwidth. However, the pipeline is not always perfectly utilized in real execution due to the following two factors:

First, the off-chip bandwidth of the underlying FPGA platform may not meet the accelerator requirement. To make the pipeline process one basepair extension operation per cycle, the platform must provide at least 25.6 GB/s random access bandwidth. This requirement, however, is beyond the capabilities of most state-of-the-art FPGA platforms, including HARPv2. When the supplied

bandwidth does not catch up with the accelerator requirement, the pipeline will continuously stall due to memory channel congestion, which in turn lowers the accelerator performance.

The other factor is that the pipeline will run out of jobs to process, especially at the end of the execution of a batch of *reads* when only a few *reads* are still not finished processing. The irregularity of the SMEM seeding algorithm makes this issue further serious. Fig. 4.2 shows the number of basepair extension iterations performed of 1000 randomly select *reads*. We can clearly see that the number of iterations performed by a *read* vary drastically from a few tens to a few thousands. If a batch of *reads* include some that performs thousands of iterations, which is shown with yellow square in the figure, its pipeline efficiency will be severely harmed because of the "long tail" at the end of the execution of the *read*.

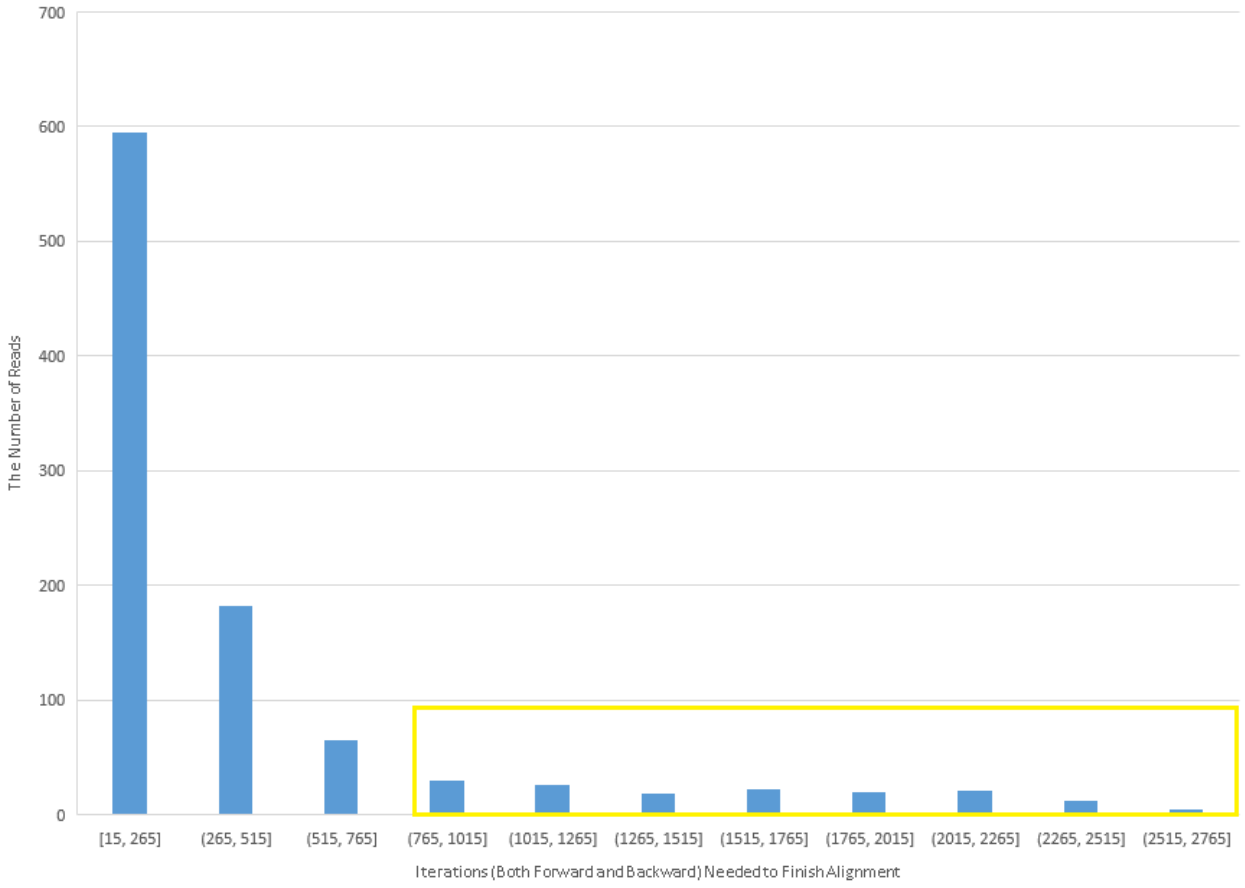

Figure 4.2: Iterations Performed of Each *read* in a Batch of 1000 *reads*

To demonstrate the impact of these two factors on the pipeline efficiency, we randomly select a

batch of 1024 *reads*, and measures the real-time memory bandwidth and the number of "bubbled" cycles of the pipeline. A bubbled cycle is referred to as the execution cycle when the pipeline is not stalled, but does not have a valid transaction to process. As is illustrated in Fig. 4.3, in the beginning of the execution, the pipeline always performs valid transactions, and the off-chip bandwidth remains at about 14 GB/s, which reflects the maximal random access bandwidth supplied by the HARPv2 platform. When most *reads* have finished their execution, the pipeline starts to suffer the long tail issue with growing bubbles and dropping bandwidth. This also explains why the SMEM++ accelerator favors a large batch size, since it makes the pipeline work efficiently in more time and amortize the long tail overhead.

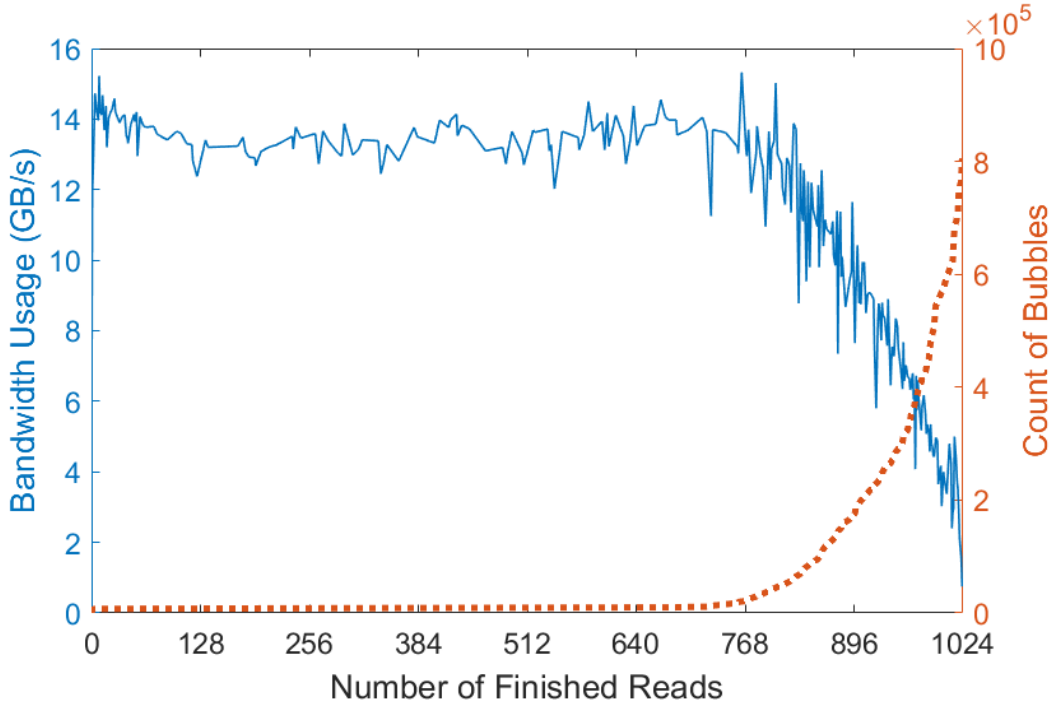

Figure 4.3: Real-Time Bandwidth Usage

CHAPTER 5

Conclusion and Future Work

In this paper the SMEM++ accelerator is proposed to accelerate the SMEM seeding algorithm. SMEM++ features a fully pipelined PE design that can achieve 200 Mbp/s when the pipeline is fully occupied. Also, an communication interface adapter is designed to bridge the PE design with the underlying FPGA platform, so as to greatly improve the portability of the accelerator. Finally, the proposed design is tested on the state-of-the-art The experiments show that SMEM++ outperforms the state-of-the-art SMEM accelerator by 6.3x, even with 43% less resource consumption. This means that given a certain resource constraint and enough memory bandwidth, the proposed accelerator could potentially reach 11x performance improvement.

Two factors are also identified that affects the pipeline efficiency. The first factor, i.e. the limited FPGA to Dram bandwidth, is the platform limitation and can be resolved with future technology advances. The second factor is due to the irregularity of the SMEM algorithm, and can be alleviated by batch processing with a large batch size. A possible solution to eliminate this issue is to dynamically send back completed reads and load new reads. This remains as future work.

REFERENCES

[1] J. Shendure and H. Ji, "Next-generation DNA sequencing," *Nature biotechnology*, 2008.

[2] J. Arram *et al.*, "Leveraging FPGAs for accelerating short read alignment," *TCBB*, 2017.

[3] H.-C. Ng *et al.*, "Reconfigurable acceleration of genetic sequence alignment: A survey of two decades of efforts," in *FPL*, 2017.

[4] "Illumina NGS." [Online]. Available: https://www.illumina.com/science/technology/next-generation-sequencing.html

[5] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Briefings in bioinformatics*, 2010.

[6] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, 1981.

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

[8] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows–Wheeler transform," *Bioinformatics*, 2009.

[9] H. Li, "Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly," *Bioinformatics*, 2012.

[10] M.-C. F. Chang *et al.*, "The SMEM Seeding Acceleration for DNA Sequence Alignment," in *FCCM*, 2016.

[11] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *ISCA*, 1981.

[12] J. Arram *et al.*, "Reconfigurable acceleration of short read mapping," in *FCCM*, 2013.

[13] P. Gupta, "Accelerating datacenter workloads," in *FPL*, 2016.

[14] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," 1994.

[15] J. T. Simpson and R. Durbin, "Efficient construction of an assembly string graph using the fm-index," *Bioinformatics*.

[16] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *ACM SIGARCH Computer Architecture News*, 2014.

[17] "Xilinx FPGA." [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

[18] "Intel HARP." [Online]. Available: https://software.intel.com/en-us/hardware-accelerator-research-program

[19] Y.-k. Choi *et al.*, "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," in *DAC*, 2016.

[20] E. Fernandez *et al.*, "String matching in hardware using the FM-index," in *FCCM*, 2011.

[21] C. B. Olson *et al.*, "Hardware acceleration of short read mapping," in *FCCM*, 2012.

[22] N. Ahmed *et al.*, "Heterogeneous Hardware/Software Acceleration of the BWA-MEM DNA Alignment Algorithm," in *ICCAD*, 2015.

[23] "Alpha Data - High Performance Reconfigurable Computing." [Online]. Available: https://www.alpha-data.com

[24] J. Stuecheli *et al.*, "Capi: A coherent accelerator processor interface," *IBM Journal of Research and Development*, 2015.

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