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

Jason Cong*, Licheng Guo*, Po-Tsang Huang†, Peng Wei*, Tianhe Yu*
*University of California Los Angeles, USA
*{cong, peng.wei.prc}@cs.ucla.edu, {lcguo, theodoreyth}@ucla.edu
†National Chiao Tung University, Taiwan
†bughuang@nctu.edu.tw

***Abstract*—[1] Next-generation sequencing 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 sacrifices the performance of individual processing elements to maximize the task-level parallelism, this methodology suffers a serious resource under-utilization issue. Therefore, we propose SMEM++, a pipelined and time-multiplexed FPGA accelerator for SMEM algorithm. SMEM++ adopts the canonical non-blocking pipeline methodology and implements a fully pipelined accelerator with initiation interval equal to one. Moreover, we design a communication interface adapter to make the accelerator compatible to the target platform interface and increase its portability. 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, with 43% less logic resource usage.**

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*[2], and sequences each *read*. The computing phase reassembles these discrete *reads* by aligning them onto a reference genome whose length is 3×10^9 basepairs. Searching on such a billion-basepair string for each of billions of *reads* makes it compute-intensive.

The computing phase for a *read* consists of 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 paper 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 differences do bring new challenges to FPGA acceleration and motivate our study.

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 tremendous random off-chip accesses. Chang et al.'s design [10] remains the only FPGA acceleration effort for SMEM algorithm. Their 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 steps sequentially, which leads to the maximization of the number of PEs and extensive exploration of task-level parallelism. 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. Our experiments show that in [10] a PE spends 56.2% of the overall execution time idling for memory responses. Moreover, the replication of PEs rapidly exhausts the FPGA on-chip resources. 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 issue, we adopt 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 instead of letting them block the computation pipeline. Such non-blocking design 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 our 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, our SMEM++ design faces and resolves the following challenges:

1) Off-chip memory structural hazard. The SMEM algorithm consists of a linear-time forward phase and a quadratic-time backward phase, both of which sends two random off-chip requests in each iteration. To process one forward and/or backward iteration per cycle, one has to face the challenge of performing multiple 512-bit off-chip requests every cycle, which often leads to structural hazards,

[1] This paper will be published in FPL 2018

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

since many FPGA platforms (e.g., Intel HARPv2 [13]) supply only one off-chip memory port.

2) On-chip memory structural hazard. The SMEM 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.

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 of most FPGA platforms. The pipeline must correctly handle inevitable memory channel congestion, i.e., properly stall and recover when congestions occur. This challenge becomes more serious since SMEM++ contains a very deep pipeline with many affiliate modules.

To address these challenges, we 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, our pipeline achieves the throughput of processing one forward or backward iteration per cycle. Since the forward and backward phases have different computation complexity ($O(N)$ vs $O(N^2)$), this unified pipeline achieves a more efficient resource utilization than the one with separate stages. Also, in each cycle the unified pipeline requires at most one retrieve and update operations on the intermediate data, which resolves the structural hazard in Challenge #2. Moreover, we modified the backward phase of the original SMEM algorithm to achieve in-place intermediate data retrieval and update, which reduces the BRAM usage by $1/3$.

While the unified pipeline processes one basepair iteration and two 512-bit off-chip requests per cycle, we propose a communication interface adapter based on the time-multiplexing and asynchronous FIFO to address Challenges #1 and #3. We demonstrate a proof-of-concept implementation based on the Intel HARPv2 platform with one read port at 400 MHz, and connect our 200 MHz SMEM++ that requires two read ports to this platform. Then our accelerator can fully utilize the bandwidth and properly stall and recover during memory channel congestion.

In summary, we make the following contributions:

- A non-blocking pipeline design that achieves a maximal throughput of processing one basepair 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.

Our experiments show that the design achieves 87.5 Mbp/s (million basepairs per second, see Section 2.1) throughput, which outperforms the best existing design [10] by 6.3x and outperforms the single thread CPU execution by 24x. Also, our design uses 43% less logic resource compared to [10].

2. Background and Related Work

In this section we first briefly describe the SMEM seeding algorithm and its differences from the conventional backward searching algorithm. Next, we review existing studies on the FPGA acceleration for various seeding algorithms. Finally, we introduce our experimental platform.

2.1. Review of SMEM Seeding Algorithm

The SMEM algorithm [9] identifies all the super-maximal exact matches (*SMEMs*) of a *read* which 1) is an exact match on the reference genome, and 2) is not contained in any other longer match. It accepts a given *read* R and a start position x as input, and extends from position x rightward (forward) one basepair at a time, until no match is found. Next, for every match found in the forward phase, the algorithm extends it leftward (backward) from position $x-1$ one basepair at a time, until no match is found. After all the matches are examined, the quadratic-time backward phase is completed. Compared to the original algorithm in [9], Algorithm 1 applies our modification on the backward phase to realize in-place retrieval and update of the intermediate data (`CurrQueue`).

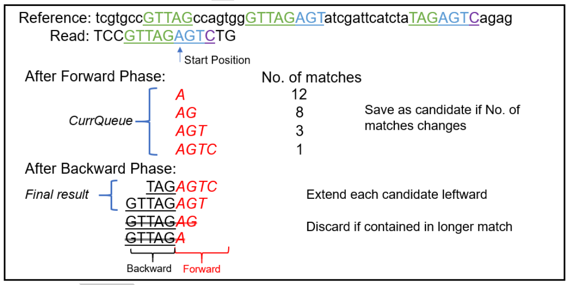
Reference: tcgtgccGTTAGccagtggGTTAGAGTatcgattcatctaTAGAGTCagag
Read: TCCGTTAGAGTCTG
Start Position
After Forward Phase:
No. of matches
CurrQueue
A
AG
AGT
AGTC
12
8
3
1
Save as candidate if No. of matches changes
After Backward Phase:
Final result
TAGAGTC
GTTAGAGT
GTTAGAG
GTTAGA
Extend each candidate leftward
Discard if contained in longer match
Backward
Forward

Figure 1: Example of SMEM algorithm

As described in [9], SMEM algorithm relies on FMD-index to realize base-pair extension in both forward and backward phases. This distinguishes the SMEM algorithm from the conventional FM-index based backward searching algorithm [8] in (1) encoding and update methods; (2) time complexity and (3) memory access patterns. These algorithmic changes bring new challenges to hardware acceleration.

Also, since the basic operation of SMEM algorithm is extending one basepair forward or backward, the performance is able to be measured by the number of basepairs processed in a unit time. Throughout this paper, we use the term *million basepairs per second (Mbp/s)* to measure the accelerator performance.

2.2. FPGA Acceleration on Seeding Algorithms

Many previous studies [14][15][12][16] are devoted to the FPGA acceleration of various seeding algorithms. [15] 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. [14] 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. [12] brings the canonical non-blocking pipeline methodology into the acceleration of the FM-index based algorithm. While both our work and [12] share the same basic idea with the non-blocking cache design in 1981 [11], the algorithmic changes lead to multiple differences and new challenges to achieve an $II = 1$ design for the SMEM acceleration, which will be discussed in Section 3.

Some studies attempt to address the same quadratic-time algorithm as ours, e.g., [16] and [10]. However, the former only optimizes the algorithm in the software level; the latter suffers the serious resource underutilization which motivates us to adopt the non-blocking pipeline methodology.

Algorithm 1 SMEM Seeding Algorithm

```
Input: Read R, Start position x
Output: All SMEM intervals containing the base pair: $R[x]$
$CurrPtr \leftarrow 0$
for $i = x + 1$ to $|R| - 1$ do                                   ▷ Forward extend
    $newIntv = BWT\_extend(lastIntv, R[i], Forward)$
    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                                        ▷ Backward extend
    $size \leftarrow 0$
    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 $OutputQueue$
```

2.3. Intel HARPv2

The second generation of Intel's Heterogeneous Architecture Research Platform (HARP) [17] represents the state-of-the-art tightly coupled CPU-FPGA platforms. To improve the communication bandwidth of FPGA, it brings a Xeon E5-26xx CPU and an Arria 10 GX1150 FPGA into a single semiconductor package. Fig. 2 illustrates the HARPv2.

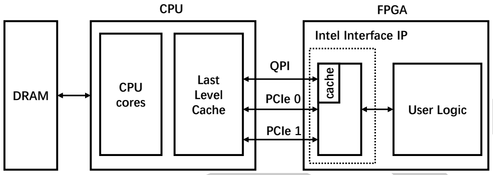

Figure 2: Intel HARPv2 System Overview

3. Non-blocking Pipeline Design

Fig. 3 illustrates the overall architecture of SMEM++ accelerator. It consists of two major components: ***Non-blocking Pipeline with II=1*** and ***Communication Interface Adapter***. We present the non-blocking pipeline design in this section, and leave the adapter design to Section 4.

The non-blocking pipeline realizes full functionality of the SMEM algorithm. It is composed of three main modules: *pipeline module*, *storage module*, and *control module*.

3.1. Pipeline Module

This module presents the datapath of our non-blocking pipeline. First we present how to make the algorithm compatible with the hardware non-blocking property, then we introduce our implementation which unifies the forward & backward phases together in parallel.

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 an FMD-index bi-interval and extend direction as input and calculates the two table reference addresses respectively to extend the interval upper/lower bounds.

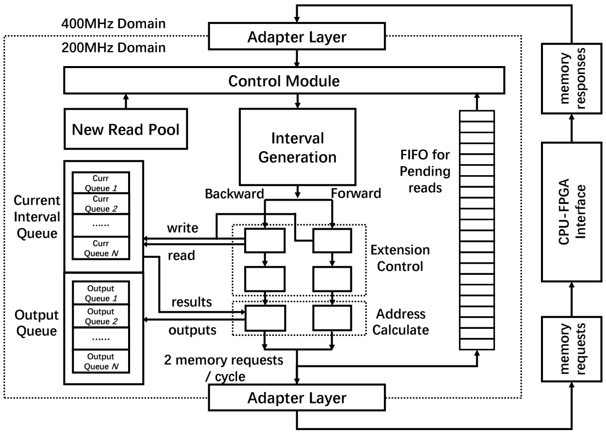

Figure 3: SMEM++ Accelerator Architecture Overview

- ***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 extension and stores the result either back in temporary queue or the final output.

Note that if directly mapped to hardware, there will be uncertain latencies between ***Step 2*** and ***Step 3***, while other steps could be tightly coupled. Thus, to make use of the non-blocking pipeline property, we shift the order of the four steps and place Step 2 at the end of the pipeline such that all fixed-latency operations are glued together. The functions are executed in pipeline in following order:

- *Interval Generation* → ***Step 3***
- *Extension Control* → ***Step 4***
- *Address Calculate* → ***Step 1***
- Issue memory requests (last stage) → ***Step 2***

After executing step 2, the computation information of the current processing *read* are stored into a FIFO just like the Miss Status Holding Registers (MSHR) in non-blocking caches, while the pipeline keeps processing other *read*s.

3.1.2. Unifying Forward/Backward Extensions.

Our unified pipeline design is based on the reordered execution flow. As previously shown, the forward and backward phases only differ in ***Step 4***. Therefore, it leads to significant resource saving if we unify the other three steps.

In addition, as challenge #2 points out, both forward and backward function will interact with on-chip memory for intermediate data, potentially resulting in structure hazard. Therefore, 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 3.3).

3.2. Control Module

The control module handles the stall and recovery of the pipeline execution when memory congestions occur.

To execute an extension operation each cycle, the pipeline requires 25.6GB/s off-chip bandwidth, which surpass the maximum bandwidth available, leading to constant memory channel congestions.

In general, the off-chip communication interface of FPGA platform will provide a `congest` signal to indicate the congestion status. When the signal is asserted, user are requested to stop sending memory requests. Therefore, when the channel is congested, we broadcasts the `congest` signal to all stages of our pipeline to stop them and preserving the current state.

However, this approach leads to a huge fanout of the `congest` signal. To resolve this, we recursively duplicate and register this signal 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 is asserted to the cycle when the pipeline execution is actually stalled. This issue is addressed by buffering the memory requests temporarily, which is discussed in detail in Section 4.

3.3. Storage Module

This module stores the input, output and the intermediate data during the processing. It consists of four components:

1) New Read Pool stores a batch of input *reads* to SMEM++.

2) Output Queue stores all generated SMEMs that are encoded as integral intervals.

3) Pending Read FIFO stores the *reads* that are waiting for their memory responses. As we reordered the memory response 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* will then be re-issued into the pipeline for the next round of extension.

4) Current Interval Queue. The current interval queue stores the intermediate data of each *read* during execution. This queue structure corresponds to the `CurrQueue` variable in Algorithm 1.

Note that we can modify the algorithm to reduce the size of *Current Interval Queue*. The original algorithm alternatively uses two queues ($CurrQueue$ and $PrevQueue$) for updating the generated intervals in the backward phase [9]. Directly using this not-in-place approach leads to a considerable on-chip RAM overhead. We reduced two queues to one by keeping the information of last and current size of the queue, retrieving the information in one iteration while process it and store it back in the following iteration. As a result, the BRAM usage is reduced by 1/3.

4. Communication Interface Adapter

This section presents how we integrate the pipeline into the target FPGA platform. We build a communication interface adapter which bridges the gap of working frequency and memory ports between the pipeline and the underlying FPGA platform interface. The adapter also improves the portability of the accelerator so that we can port it elsewhere by just reconfiguring the adapter.

4.1. Compatibility Issue

Our pipeline is designed to work at 200MHz. To achieve $II = 1$, it needs one 1024-bit data width memory port since each basic extension operation requests 2 cache line of data. However, our experimental platform, Intel HARPv2, supplies a 400 MHz off-chip communication interface with only one 512-bit read port.

Instead of changing the pipeline, we design an interface adapter between the user interface and the vendor-provided interface. To implement the accelerator onto a certain platform, we only need to modify the adapter to make the accelerator compatible, avoiding changing the complex accelerator itself.

4.2. Adapter Implementation

Fig. 4 illustrates the overall architecture of the adapter, which consists of two parts.

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. This time-multiplexing implementation successfully adapts the requirement of two reads per 5ns to one 400 MHz read port without performance degradation.

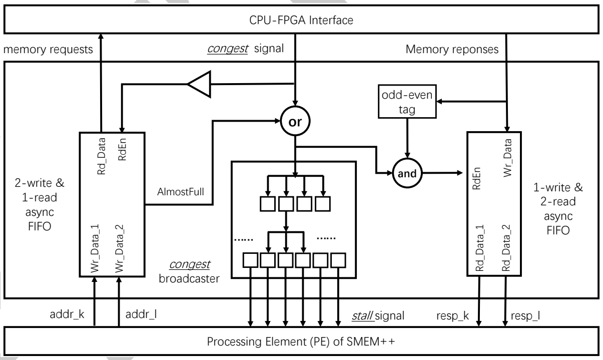

Figure 4: Communication Interface Adapter Architecture

2) Response channel. Similarly, response channel adopts an asynchronous FIFO with one write port and two read ports, and the responses are read out of the FIFO pair by pair. Another buffer follows the channel in case congestion occurs when reading out results.

Moreover, the adapter resolves the issue due to the broadcasting delay of the `congest` signal, as mentioned in Section 3.2. When the `congest` signal is set by the interface but has not reached the pipeline, the FIFO in request channel temporarily buffers the requests. After the congestion is resolved, the FIFO first sends out the buffered memory requests, then releases the stall to the pipeline.

The adapter methodology is applicable for other CPU-FPGA platforms. Specifically, Alpha Data board [18], IBM CAPI [19] and Amazon F1 instance [20] provide 512-bit memory ports, which aligns with the accelerator interface very well. The frequency gap can then be addressed by the asynchronous FIFOs in the design.

5. Experimental Evaluation

This section presents the experimental evaluation to the proposed accelerator design. We first describe our experimental setup (Section 5.1). Then we present the overall performance and resource consumption of the proposed design, and compare both with those of [10] (Section 5.2). Furthermore, we take a deeper look at the pipeline execution, with a key focus on the pipeline efficiency (Section 5.3).

5.1. Experiment Setup

We use the Intel HARPv2 platform and the input comes from a human genome sample (HCC1954 [21]), which contains 1 billion *reads*, each with 101 basepairs. Our accelerator works at 200 MHz and the interface adapter bridges it with the 400 MHz HARPv2 interface. For the software side, we adopt the batch processing methodology used in [10] to process a large number of *reads* at a time. We refer to the number of *reads* in a batch as *batch size*, a key parameter to the pipeline efficiency.

5.2. Overall Performance and Resource Utilization

Table 1 lists the resource consumptions of the accelerator for different batch sizes. The logic resource consumption is nearly constant, but the RAM consumption grows significantly as the batch size increases. This is because a *read* batch with a larger size requires more BRAM to store the input, intermediate data, and output. In comparison, the state-of-the-art SMEM accelerator [10] uses 164,273 ALMs, and SMEM++ uses only 57% as much logic resource.

Table 1: Resource Consumption of SMEM++

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%)

Fig. 5 shows the performance of SMEM++ and off-chip bandwidth usage. it achieves a throughput of 87.5 Mbp/s and uses 11.2 GB/s off-chip bandwidth, outperforming [10] by 6.3x with 43% less logic resource. Moreover, SMEM++ outperforms the single thread Xeon E5 2680 by 24x.

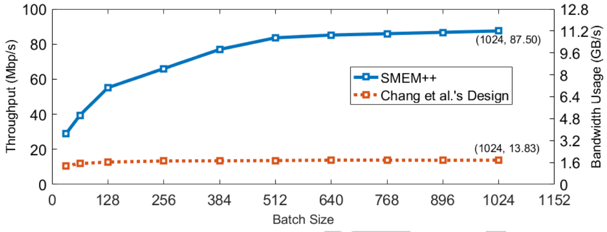

Figure 5: Overall Performance and Off-Chip Bandwidth Usage

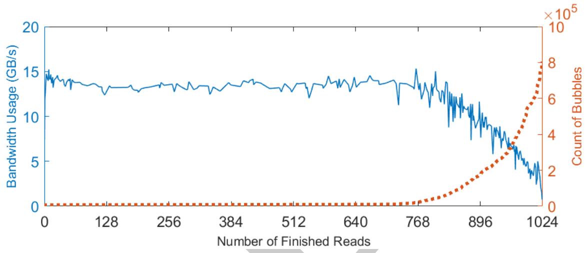

Figure 6: Real-Time Bandwidth Usage

5.3. Pipeline Efficiency Analysis

When the SMEM++ works in the optimal status, it utilizes a 25.6 GB/s off-chip bandwidth. However, it is not always perfectly utilized in real execution due to the following two factors. First, the off-chip bandwidth of the platform may not meet the accelerator requirement. Second, the pipeline will run out of jobs when it almost finishes processing a batch of *reads* but a few *reads* are still there. This is caused by the irregularity of the SMEM algorithm. The number of operations needed by a *read* vary drastically from a few tens to a few thousands. Those long-lasting *reads* become a "long tail" at the end of the execution of a batch.

To demonstrate it, we randomly select a batch of 1024 *reads*, and measures the bandwidth usage and the "bubble" cycles of the pipeline, i.e. cycles when there are no available pipeline input and a void input (bubble) will be issued instead. As Fig. 6 shows, at start the pipeline always performs valid transactions, and the off-chip bandwidth remains at about 14 GB/s, which reflects the maximal random access bandwidth of HARPv2. When most *reads* have finished execution, the count of cycles without available input increases. This also explains why SMEM++ favors a large batch size, which amortizes the long tail overhead.

6. Conclusion and Future Work

In this paper we present SMEM++ to accelerate the SMEM algorithm. SMEM++ features a non-blocking pipeline with $II = 1$ and a communication interface adapter to bridge the accelerator and underlying platform. SMEM++ outperforms the state-of-the-art SMEM accelerator by 6.3x, with 43% less resource usage. We also analyze the pipeline efficiency. The bandwidth limitation can be resolved with future improvement on off-chip bandwidth. The "long tail" is due to the irregularity of the algorithm, and can be partially alleviated by batch processing. Another solution is to dynamically send completed *reads* back to and load new *reads* from memory. This remains as the future work.

Acknowledgement

This research is partially supported by the contributions from Intel and Huawei under the Center for Domain-Specific Computing (CDSC) Industrial Partnership Program. We would like to thank Intel for donating the Heterogeneous Architecture Research Platform (HARP).

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] E. Fernandez *et al.*, "String matching in hardware using the FM-index," in *FCCM*, 2011.
[15] C. B. Olson *et al.*, "Hardware acceleration of short read mapping," in *FCCM*, 2012.
[16] N. Ahmed *et al.*, "Heterogeneous Hardware/Software Acceleration of the BWA-MEM DNA Alignment Algorithm," in *ICCAD*, 2015.
[17] "Intel HARP." [Online]. Available: https://software.intel.com/en-us/hardware-accelerator-research-program
[18] "Alpha Data - High Performance Reconfigurable Computing." [Online]. Available: https://www.alpha-data.com
[19] J. Stuecheli *et al.*, "Capi: A coherent accelerator processor interface," *IBM Journal of Research and Development*, 2015.
[20] "Amazon EC2 F1 Instances." [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/
[21] A. F. Gazdar *et al.*, "Characterization of paired tumor and non-tumor cell lines established from patients with breast cancer," *IJC*, 1998.