FPGA ACCELERATION OF DNA ERROR CORRECTION

BY

ANAND RAMACHANDRAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Deming Chen

ABSTRACT

Correcting errors in DNA sequencing data is an important process that can improve the quality of downstream analyses using the data. Even though many error-correction methods have been proposed for Illumina reads, their throughput is not high enough to process data from large genomes. The thesis describes the first FPGA-based error-correction tool, which is designed to improve the throughput of DNA error correction for Illumina reads. The base algorithm of the FPGA implementation is BLESS which is highly accurate but slow. The hardware implemented on the FPGA consists of a Bloom filter that is the main data structure of BLESS and the error-correction subroutines in BLESS. The design is compared with the software version of BLESS, and two other leading tools. The results show significant improvements in speed for the FPGA-based implementation with comparable accuracy of error correction.

To my parents, for their love and support.

ACKNOWLEDGMENTS

This work was supported by a donation from Altera Corporation, the IBM Faculty award, the UIUC CompGen Fellowship, and the Samsung Fellowship. I would like to thank my adviser Professor Deming Chen, and colleague Dr. Yun Heo, whose valuable insights helped this project greatly.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This chapter provides an introduction to DNA error correction, and briefly considers issues related to the throughput and acceleration of the associated algorithms.

First, the nature of data available from DNA sequencing is examined, and the need for DNA error correction is explained. Next, a popular and highly successful class of DNA error-correction algorithms is introduced. The need to accelerate these algorithms is discussed, and the opportunities and challenges in accelerating them are listed. Third, Feld Programmable Gate Arrays (FPGAs) are introduced as a viable platform that may be employed to accelerate these error-correction algorithms, and a particular algorithm is picked, matching its specific features with the strengths/weaknesses of an FPGA. Finally, related work in FPGA acceleration as applicable to bioinformatics, and the novelty of the work attempted in this thesis are discussed.

1.1 DNA Sequencing Data

Each organism has material inside its cells called the genome that has information on how the organism is built and how it functions. The genome is composed of a sequence of Deoxyribonucleic acid (DNA) bases, also called nucleotide bases (in the foregoing, the terms “genome” and “DNA” will be interchangeably used). Given its fundamental nature in governing how an organism executes life processes, examining genomic data is essential to many biological and medical analyses. The information regarding the DNA is extracted through the process of DNA sequencing. The entirety of this information may be viewed as long strings of characters (running to billions in large genomes) belonging to the alphabet {A, C, G, T} representing the two strands of the DNA molecule. Each of the characters in the alphabet represents a nucleotidic base, that is, one of Adenine, Guanine, Thymine or Cytosine.

However, sequencing does not give us two contiguous strings representing the DNA strands. Instead it gives us a large number of much shorter strings, called reads. Information on which portion of the genome each read originates from is unknown. Reads overlap with each other to provide some redundant information, at the same time the extend of overlap is variable from read pair to read pair, and unknown. In addition, some of the reads sequenced from the same region of the genome may disagree because of errors in the sequencing process. Given this nature of DNA sequencing data, it is difficult to directly build a picture of the organism's genome from reads. Hence, many computational methods and algorithms have been invented to extract information about the genome from sequenced data.

DNA sequencing is becoming increasingly affordable. The cost of sequencing is estimated at approximately \$1000/genome. The trend in sequencing cost has been following a steep fall over the years, often advertised as steeper than Moore's law [1]. Coupled with the massive increase in sequencing throughput, this has led to the initiation of large-scale sequencing projects that aim at sequencing millions of genomes [2]. Such projects are ushering in a future of precision medicine, which would enable clinical decisions based on genome sequencing and analysis of the sequenced data.

Even though the throughput and cost of sequencing are improving rapidly, DNA reads remain erroneous. For example, the most popular sequencing platform available today is from Illumina. Illumina's sequencing machines provide reads of length up to 300 bases long [3]. Illumina's HiSeq X Ten system can sequence 1.8 Tbp (tera base pair) in a single run spanning less than three days; however only three-fourths of the reads from this sequencing run have sufficiently high quality [4] (quality of bases in a read is available as a parameter of the sequenced data; "high quality" here means a quality score of more than 30). These sequencing errors can cause errors in the analyses that use the data, and correcting these errors can improve the quality of these analyses [5], [6].

1.2 k-mer Spectrum-Based Algorithms for Error Correction

Fortunately, two properties of the sequenced data which were briefly introduced in the previous section can facilitate the correction of a majority of these errors:

- Redundancy: Reads overlap. Though there may be disagreements among reads coming from a particular region in the genome, regarding a base in an overlapping portion, usually the error is present only in a minority of the overlapping reads. Hence, a majority vote may be used to determine the correct base at that position.
- Quality Scores: The data available from sequencing machines comes with confidence values for each base in each read. These quality scores can give information that helps us to correct errors.

There is a highly successful class of algorithms called k-mer spectrum-based algorithms [7] that have been devised to correct errors in Illumina reads, in which the errors are usually substitutions, where a base in the genome is wrongly replaced with a different base in the read. These algorithms make the following heuristic assumptions:

1. If a k-length segment of a read, called a k-mer, matches with a k-mer from another read, the reads overlap. This is largely true in genomes that do not have repetitive segments if k is kept sufficiently large. Usually, k is kept in the range 15 to 50 depending on the type of the genome being sequenced.
2. If a k-mer does not occur with high multiplicity in the read dataset, then some base(s) in the k-mer is(are) erroneous and the k-mer needs to be corrected. This is based on the assumption that many reads are sequenced from any given part of the genome, which leads to the conclusion that a k-mer in that area of the genome will be shared among many of these reads. Hence a k-mer will have high multiplicity in the set of all the reads whose overlaps contain that k-mer, unless some of the reads are affected by a sequencing error changing the k-mer.

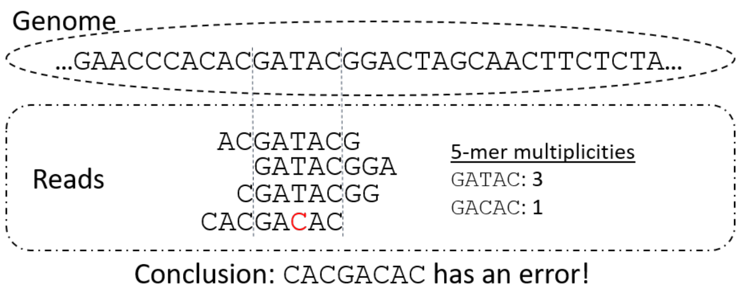

Figure 1.1: Errors in sequencing data

Figure 1.1 depicts the basic assumptions and high-level idea behind k-mer spectrum-based error correction. The example shows reads of length 8, all sampled from the same region in the genome. 5-mers from the reads are examined, and a particular 5-mer, GACAC, is found to have a low multiplicity while another, GATAC, has a high multiplicity. Error-correction tools in the k-mer spectrum category try to convert GACAC in the source read to GATAC. Different tools in the category follow different methods to determine the correct transformation.

As accurate as they are, k-mer spectrum-based algorithms require long durations to complete execution. A recent study [8] evaluated that the amount of time needed to correct errors in reads from the entire human genome can be up to four hours. In light of the fact that the amount of sequencing data is going to rapidly increase in the coming years, it becomes extremely important to correct these errors as fast as possible to enable faster DNA analysis pipelines.

There are opportunities to accelerate these algorithms on parallel architectures, but also associated challenges. For instance, the algorithms are highly data-parallel in that two reads can be corrected independently, once the k-mer multiplicities are computed. At the same time, given that the algorithms depend on the statistics of occurrences of different k-mers, and because different reads may have different numbers of errors, the particular operations followed in correcting two different reads may be vastly different, and may have to be expressed using sophisticated routines. Because of this, it is desired that the computing platform chosen to accelerate the algorithms must be able to support parallelism and efficient execution of complex serial tasks.

1.3 FPGAs for Accelerating DNA Error Correction

FPGAs are reconfigurable logic design platforms that allow the user to specify a hardware-level implementation of an algorithm. A standard FPGA consists of an array of Look-Up Tables (LUTs), registers, Arithmetic and Logic Units (ALUs), and memory bits (as Block RAMs or BRAMs), which can be configured as needed. The architecture of FPGAs allows design of sophisticated state-machines and data-paths in a highly optimized manner. It is also possible to use multiple instances of the same hardware block to do parallel processing. An FPGA can thus be used to parallelize as well as optimize serial execution of an algorithm, and hence suggests itself as a good candidate platform for use in accelerating DNA error correction.

The FPGA, however, has a limitation that it usually has access to limited off-chip memory compared to the CPU. Hence, to adapt an error-correction tool to the FPGA platform, the chosen k-mer spectrum-based algorithm has to be memory frugal. Many algorithms in this class require a lot of memory because they store the multiplicities of k-mers in the sequenced data. However, BLESS [9], one of the most accurate k-mer spectrum-based algorithms is also extremely memory efficient. BLESS achieves this memory efficiency by employing a Bloom filter to store high-multiplicity k-mers. The Bloom filter can store a large number of strings in a small amount of space, if a certain rate of false positives can be tolerated. BLESS uses additional algorithmic steps to tolerate such errors from the Bloom filter.

Considering the merits of the FPGA platform and the merits of the BLESS algorithm, they arise as good fits for each other. In this thesis, an implementation of BLESS on FPGA will be examined in detail.

However, before going into further details about BLESS and the hardware implementation, the work done in the thesis must be compared to related work in FPGA acceleration, to clearly express its novelty and value.

1.4 Related Work

Examining past work in accelerating genomic workloads on FPGAs, one can see many instances but none that accelerate error correction of Illumina data. One work, [10], corrects errors in Pacbio sequencing reads using FPGAs. The base algorithm can correct errors in PacBio sequencing reads (http://www.pacificbiosciences.com) by aligning Illumina reads that have fewer errors to the PacBio reads. Their implementation focused only on the alignment part.

On the other hand, the Bloom filter that is the main data structure in BLESS has been widely used in FPGA designs mainly for finding patterns. In [11], [12], Bloom filters were used to accelerate the seed matching stage of BLASTN [13], [14] that is the most popular homology database search algorithm. Bloom filters were used in these works to store k-mers in querying sequences.

In addition, there are plenty of successful implementations of genomic tools on FPGA. For example, [15] accelerates the assembly of reads into larger contiguous strings called contigs, a process called de-novo Assembly. In [16], the process of DNA sequence alignment, in which DNA sequence data is mapped to a reference DNA sequence by minimizing the edit distance between the reads and the reference sequence, is accelerated.

This thesis presents the first FPGA implementation of DNA error correction for Illumina reads. It obtains up to $40\times$ acceleration compared to the base version of BLESS, on which the implementation presented here is based. The presented implementation is compared against two other popular CPU-based error-correction tools as well, and it comes out to be faster in these comparisons. Its accuracy is comparable to that of BLESS while not equal to it. These points will be explored in detail in the remaining sections.

The rest of the thesis is divided into two parts. Chapter 2 examines background concepts regarding the Bloom filter, and the BLESS algorithm. Chapter 3 examines techniques that were used to map BLESS onto an FPGA [17] using customized hardware. The results of experiments with the implementation using various benchmarks are also presented.

CHAPTER 2

BACKGROUND

2.1 Bloom Filter

At the heart of BLESS is a Bloom filter [18], a space-efficient data structure that responds to membership queries. A Bloom filter can return false-positive queries, but no false-negatives. BLESS uses the Bloom filter to store a list of high multiplicity k-mers in the read set, called solid k-mers. Low multiplicity k-mers are called weak k-mers. Use of a Bloom filter suggests that BLESS may be prone to false corrections. However, BLESS tests a base correction by checking the solidity of multiple k-mers containing that base, so the effects of false positives from the Bloom filter are reduced drastically.

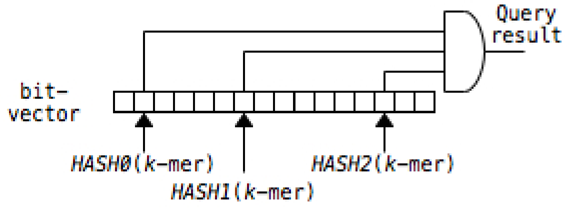

Figure 2.1: Bloom filter query process

The query operation in a Bloom filter is depicted in Figure 2.1. A given input is hashed into multiple locations in a bit-vector. If all the locations in the bit-vector are 1, the result of the query is positive, otherwise it is negative. For a store operation, all the hashed bits are set to 1. Under the assumptions of ideal random and independent hashes, the false positive rate (FPR) of a Bloom filter can be adjusted as

$$FPR = (1 - e^{-qn/m})^q \tag{2.1}$$

where q is the number of hash functions, n is the number of elements stored in the Bloom filter, and m is the size of the bit-vector. The Bloom filter size for BLESS for large genomes can run up to a few GBs. Thus the Bloom filter queries are largely expected to go to main memory.

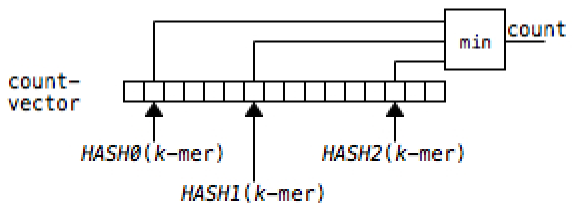

Figure 2.2: Counting Bloom filter

A variant of the Bloom filter, called the counting Bloom filter (CBF) replaces the bit-vector with a count-vector, (Figure 2.2) where each location is a count value. Originally designed to be a type of Bloom filter that could program as well as delete elements from it, this data structure can also be used to count the multiplicity of a stored element. The query input is now hashed into multiple count values in the count-vector, instead of bits in a bit-vector. The minimum value among all the indexed counts is an estimate of the multiplicity of the queried element. When an element is to be stored in the CBF, the hashed count values are all incremented.

Given that hash functions for the Bloom filter generate hash values that are uniformly distributed across positions in the bit-vector, there can be wastage of memory bandwidth during the query and program operations. A DDR3 memory fetch, for example, opens up a whole line in the memory, which can run up to a couple of Kbs, while the Bloom filter might end up using only one bit in that line. Other hash values may index locations in other DDR lines. A solution to this problem is a blocked Bloom filter [19]. As shown in Figure 2.3, the blocked Bloom filter is an array of bit-vectors. A hash function $HASH0$, say the primary hash function, hashes the input into one of the bit-vectors. This bit-vector is read from memory, and the remaining hash functions (call them subsidiary hash functions) index into locations within this bit-vector. This ensures that all operations access locations close to each other in memory thus ensuring better memory performance. The blocked Bloom filter performs with better throughput but also a higher false positive rate, depending on how well the hash functions perform.

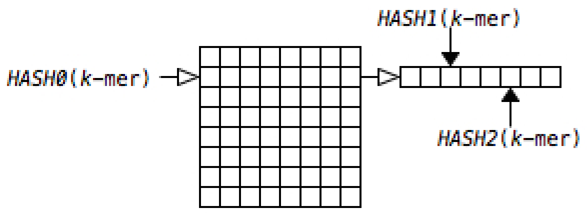

Figure 2.3: Blocked Bloom filter

2.2 BLESS

2.2.1 Counting k-mers

Being a k-mer spectrum-based algorithm, BLESS has to count the number of occurrences of each k-mer across all reads in the input dataset. A read of length l can give $(l-k+1)$ k-mers. The size of all k-mers is thus much larger than the size of the reads, and a data-structure designed for (key, value) storage of all k-mers in the read set and their associated multiplicities can be too big to fit in main memory.

BLESS does the counting by hashing k-mers into a number of files such that a particular k-mer will be written always into a particular file. This means that when finally counting the k-mers, these files can be loaded one at a time, and each file can be made small enough to fit into main memory.

To decide solid k-mers, either the user can provide a threshold for the number of occurrences above which a k-mer must be considered solid, or let the tool decide a good threshold. In the hardware implementation, the former method is used and the value may be programmed into the FPGA through an access through the PCIe port, though it is not very hard to mimic the tool's method of determining the threshold in hardware. The solid k-mers are then programmed into a Bloom filter for use during error correction.

2.2.2 Determining locations to be corrected

BLESS corrects errors in reads by replacing a weak k-mer by the correct solid k-mer. For this, first the parts of the read containing weak k-mers must be determined. A simple way to achieve this is to query whether each k-mer in a read exists in the Bloom filter's list of solid k-mers, and if it does not exist in the Bloom filter, to correct it. However, the Bloom filter can deliver false positive queries, so BLESS stands to miss correcting some of the weak k-mers by following this naive procedure. To avoid this problem, BLESS uses some heuristics to turn some of the solid k-mers into weak k-mers.

For example, if there is an isolated solid k-mer in a read, then that k-mer is considered weak. If there is an island of weak k-mers of size $k-1$ (a weak k-mer island is a set of consecutive weak k-mers surrounded by solid k-mers or the read boundary; a similar definition may be used for a solid k-mer island), BLESS extends the island by one position in both directions if possible. This is because an erroneous base will be contained in k k-mers and one should expect as many weak k-mers to be present adjacent to each other in a read containing an erroneous base.

2.2.3 Correcting errors

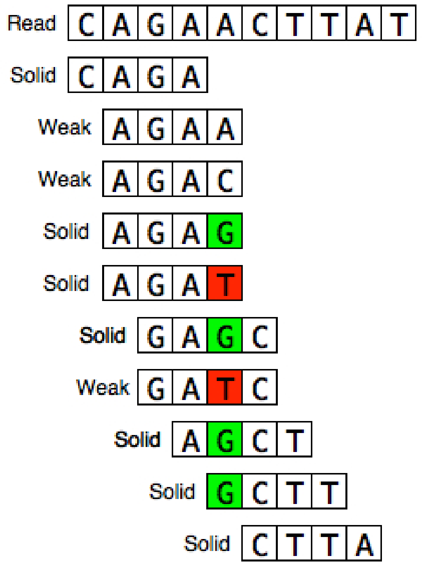

Figure 2.4: BLESS error-correction algorithm

Figure 2.4 depicts a sample execution of the core error-correction routine in BLESS that determines possible corrections to an erroneous read. Given is a read of length 10, and correction is attempted using 4-mers. The first k-mer, CAGA, in the read is a solid k-mer. This implies that all the bases

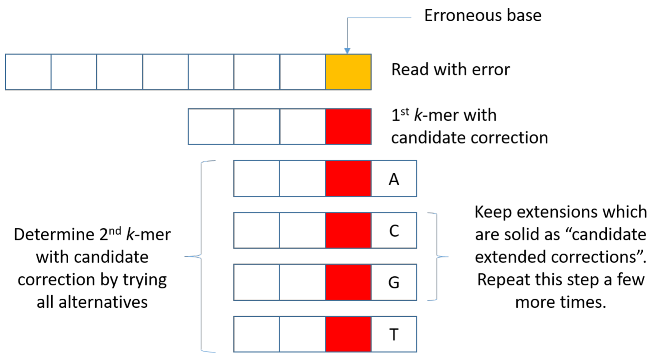
Erroneous base
Read with error
1st k-mer with candidate correction
A
C
G
T
Determine 2nd k-mer with candidate correction by trying all alternatives
Keep extensions which are solid as "candidate extended corrections". Repeat this step a few more times.

Figure 2.5: Corrections at the end of a read

in the first k-mer are correct. The next k-mer, AGAA, is weak. This means that the last base in the second k-mer has to be corrected. The algorithm tries three alternative k-mers with bases C, G, and T at the last position. Two alternatives, AGAG and AGAT are solid k-mers, but only one of the two can be the correct replacement for the second k-mer in the read. The other could arise from a similar k-mer in another part of the genome, or from a false positive from the Bloom filter. To filter out wrong candidate correction(s), adjacent k-mers from the read are checked, assuming that each of the two possible candidates is the correct one. In this case, the correction implied by AGAG holds and is accepted as a valid candidate correction. In general, when k k-mers covering a base correction all turn out to be solid, that base correction is a good candidate to retain.

If the base to be corrected is at one of the ends of the read, it may not be possible to find k k-mers within the read containing each candidate correction to the base. In this case, the algorithm looks for all possible extensions of the k-mer beyond the end of the read by removing one base at the end contained inside the read and trying out all four bases at the opposite end. The extension is done for a few steps less than k. This is depicted in Figure 2.5.

There can be cases where the algorithm needs to correct a read with no solid k-mers in it. In this case, first the leftmost k-mer (or the first k-mer) is corrected by making a small number of modifications to it. Subsequently, the algorithm presented above is employed to correct the rest of the read starting from the second k-mer. The second step has to be executed multiple times if multiple choices exist for the correct first k-mer in the read.

It is possible that after the algorithms described so far complete, they provide a list of candidate corrections rather than just one. BLESS sums up the quality score of each corrected base in a candidate, and discards all candidates except the one with the minimum total score. The hardware implementation follows BLESS, but uses a reduced 2-bit representation of quality scores rather than the 8-bit native representation.

CHAPTER 3

HARDWARE ACCELERATION OF BLESS

3.1 Design Methodology

3.1.1 Overview

The custom design for accelerating BLESS is implemented on the DE5 FPGA board that comes with Altera's Stratix V FPGA. The board supports communication with a host PC through a PCI Express Gen3 x8 bus. It has two DDR3 memory modules mounted on it, of capacity 2 GB each. The memory modules are accessed through Altera's IP blocks, which provide an interface to an accelerator with a bus width of 512 bits per access at a rate of 200 MHz.

The overall architecture of the hardware implementation is shown in Figure 3.1. First, the host PC dispatches reads to the FPGA via the PCIe interface for counting the occurrences of k-mers. The custom hardware implementation (or "accelerator", henceforth) uses a CBF to count the occurrences of the k-mers in reads. To do this, the hardware extracts all k-mers from each of the reads and programs them into the CBF counter vector housed in the DDR3 memory modules on the board. Then, reads are transferred to the FPGA a second time, and weak k-mers in each read are corrected in the error-correction unit and the corrected reads are read back by the host PC. Each operation will be explained in the following sections in detail.

3.1.2 Hardware components

This section will describe the main hardware components used in the design.

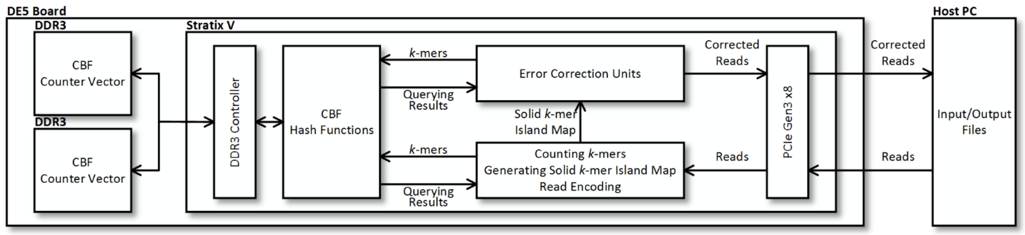

Figure 3.1: Architecture of hardware implementation

Counting Bloom filter

The CBF in the hardware implementation is fashioned after the blocked Bloom filter depicted in Figure 2.3. Instead of the primary hash function choosing a bit-vector, the primary hash function will now choose a count-vector and the subsidiary hash functions will index into this vector of count values. An example structure of the CBF for 45-mer queries is shown in Figure 3.2. Here, the number of input bits is 90 because two bits each are needed to represent a base from the alphabet {A, C, G, T}, and hence a string of 45 such bases needs 90 bits. The main hash function $HASH0$ addresses into the DDR3 memory retrieving a block of data which is 1024 bits wide. This 1024-bit wide block is an array of 2-bit counter values. The remaining 9-bit wide hash functions address into this array of counters. During the program operation, the selected counter values are incremented and the line is written back into memory (the counters saturate). For a query operation, if each selected counter exceeds the threshold value for solidity, the query (k-mer) is determined to be solid.

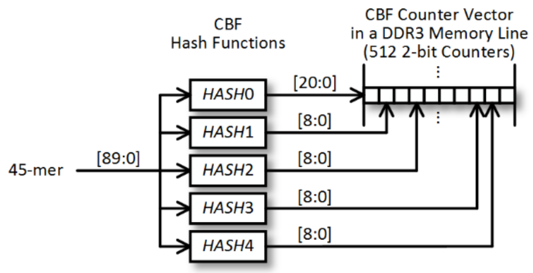

Figure 3.2: The blocked counting Bloom filter structure used to count k-mers

The hash functions in the CBF are created out of look-ahead Linear Feedback Shift Registers (la-LFSR) [20]. An n-step la-LFSR is a pseudorandom number generator, which in a single iteration, implements n iterations of a corresponding LFSR. In the accelerator, a k-mer is used to determine the initial state of the la-LFSR and after running the la-LFSR the new state is used as a hash output. Computationally simple, with a systematic method to determine the circuit of an n-step la-LFSR given n and the taps of the corresponding LFSR, there are three advantages in using this method for implementing hash-functions:

- The structure of the la-LFSR can be easily pipelined. For example, to generate a single 21-step la-LFSR, one may use three 7-step la-LFSRs cascaded one after the other giving a three-stage pipelined implemetation.
- Multiple hash-functions can be generated by simply changing the LFSR taps.
- The RTL for a hash function can be automatically generated. To help design different configurations of the accelerator, a C program was written that generates the Verilog code for a hash function of given specification using the la-LFSR theory.

As explained before, the rationale behind using a blocked memory layout for the CBF is to reduce the number of DDR3 accesses required per k-mer program and query operation. A normal Bloom filter requires as many random memory accesses as there are hash functions, whereas the blocked type Bloom filter requires access to a single contiguous block of data (bits addressed by $HASH0$). To further minimize the access times, one must limit the number of DDR3 pre-charge cycles required per k-mer query. This means that the number of bits accessed by $HASH0$ should be less than or equal to the bit-width of a DDR3 row, since only a single row is available for reads or writes after a pre-charge cycle. If a smaller number of bits than the DDR3 row-width are accessed per k-mer program or query, then a pre-charge cycle is not completely utilized. However an increase in the number of bits read by $HASH0$ would not increase the throughput of the algorithm, though it could potentially improve its accuracy by reducing the chances of two different k-mers choosing the same set of counters within a block. Hence, allocating only 512 bits for each count-vector accessed by $HASH0$ (DDR3 row width in DE5 is greater than 512 bits), will minimize the number of accesses requested from

Altera's external memory controller IP per k-mer program and query (since the bit-width of the IP's data bus is 512). At the same time, since BLESS verifies a base correction by querying more than one k-mer containing that base, the effect of the potentially higher false positive probability arising from the blocked layout of the Bloom filter is reduced greatly.

Since the la-LFSR hash functions are randomizing functions, the addresses generated for the DDR3 memory will have little spatial locality. This allows exploitation of bank-parallelism in such memories. Bank-parallelism means that a DDR memory can work on accesses to different banks in the memory at the same time. Hence, if the generated memory accesses are very well spread across the different banks within the same memory device, then the throughput of the memory accesses is maximized. In addition, there are two DDR3 memory devices on the DE5 board, and each of the devices can act independently of the other. This provides another opportunity to increase memory throughput and reduce the time taken to query for a k-mer on the average. Combining these features of DDR memories and the particular memory specifications of the DE5 board, the CBF in the accelerator is split across the two DDR3 memory slots on the DE5 board, the CBF utilizing all 4 GB of space available. The $HASH0$ function is designed to generate an address whose bit-width is one more than that of the address bus of each individual DDR3 memory module on the board. The most signficant bit of the address is examined to determine which of the two memory slots an access goes to. Since the two different DDR3 slots can have two different response times, a slot that was requested for data second can respond first. To deal with this problem, the MSB of the address is stored until the access is completed, to correctly order the responses from the slots and send them back to the CBF logic.

Even though the la-LFSR-based hash functions generate addresses that are pseudorandom, it is possible that at some point of time, many addresses are consecutively generated to access only one of the two DDR3 memory slots. This can lead to one of the DDR3 memory slots being swamped with requests while the other slot is idling. To average out the effects of such blocks of consecutive accesses occurring to a single slot, extra First In First Out (FIFO) queues are added at the input interfaces of the DDR3 controller IPs.

Figure 3.3 shows the overall memory organization for the CBF as discussed

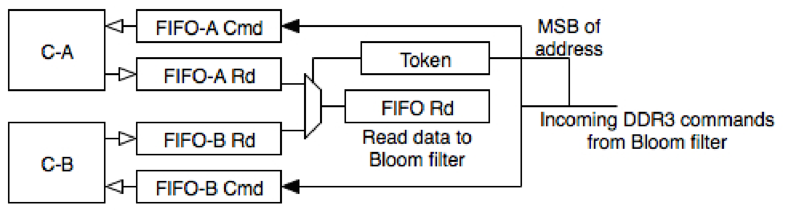

Figure 3.3: Memory organization for the counting Bloom filter

above. An example is described next to demonstrate the operation of the memory structure.

C-A and C-B are two memory controllers. The address map for the CBF starts at C-A and ends in C-B. Say, C-A and C-B have a capacity to accept eight outstanding memory accesses at any given point of time. Even though the hash functions in the Bloom filter should generate addresses uniformly across the memory map, it is possible that addresses come in chunks addressing C-A and C-B at some point in time. Say, the first sixteen requests are to C-A and the next sixteen are to C-B. In the native design, the accesses would be stalled after the eighth access to C-A because the subsequent accesses have to wait for C-A to complete processing the first request. Note that C-B is idling during this period while accesses for C-B may be ready in the pipeline. To prevent this stall, the extra accesses are collected into FIFO-A allowing the second set of sixteen accesses to proceed to access C-B. Since the seventeenth request (first request to C-B) may be serviced before the sixteenth request (last request to C-A), the read data may be available out of order. To correctly order the read data, the MSBs of the accesses, identifying which DDR3 memory an access is meant for, are stored in a FIFO (labeled "tokens"), and in conjunction with a MUX determine, the order in which the read data must be sent back.

Read profiling unit

After the CBF has been programmed with all the k-mers in the read dataset, and before correcting errors, the accelerator has to decide which bases in a read need to be corrected. There are $l-k+1$ k-mers in a read R with length l, which are generated in parallel and sent to the CBF. The results of the queries are used to build the solid k-mer island map of the read, which is a bit-vector with $l-k+1$ bits with a 1 at a position indicating that a k-mer starting at that position within the read is solid. The parallel extraction of k-mers out of a read allows the generation of a large number of queries for the CBF, which sends requests to DDR3, making good utilization of the DDR3 bandwidth. For example, when l is 101 and k is 45, 57 k-mers are generated in a single clock-cycle. In addition, the hardware cost of doing this is very little given that only rewiring is needed for this. An example is in Figure 3.4.

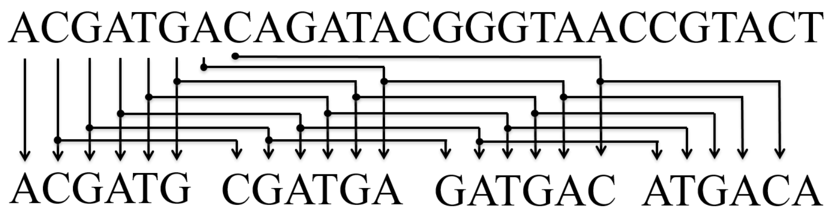

Figure 3.4: Generating k-mers from reads

Once the island map is produced it is sent through a pipeline of combinational logic masks that implement many of the heuristics BLESS uses so as to mitigate the effects of the false positives of the Bloom filter, or to filter out solid k-mers which may be produced due to a similarity between an erroneous read and another part of the genome that is sequenced correctly.

The recursive unit

The base algorithm in BLESS was summarized in Figure 2.4. This algorithm returns a list of candidate corrections that fix the weak k-mers in a read. Naturally, the core hardware unit for correction executes this functionality. The software implementation of BLESS executes this functionality using recursive function calls, hence the rest of the document will call the hardware that executes this feature, the recursive unit. The implementation of this unit is depicted in Figure 3.5. It uses two BRAMs to store intermediate candidate corrections. Referring back to Figure 2.4, for example, after looking at the first weak k-mer in the read, the intermediate candidates will be CAGA**G**CTTAT, and CAGA**T**CTTAT. In a recursion that looks at an even numbered position in the read, candidates are picked from BRAM1, k-mers produced from it, new candidates decided for correcting the current location, and the results written to BRAM0. In a recursion that looks at an odd numbered position in the read, the roles of the BRAMs are interchanged.

Comparing to the software, the recursive call stack includes the candidate corrections and various control signals and state-machine bits that indicate the stage of execution.

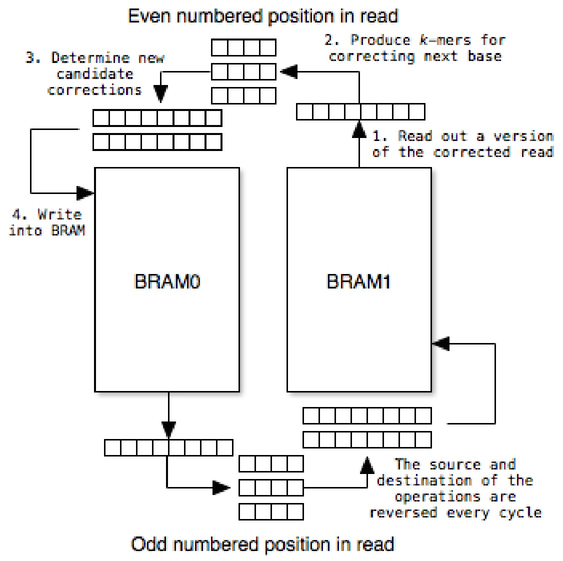
Even numbered position in read
3. Determine new candidate corrections
2. Produce k-mers for correcting next base
1. Read out a version of the corrected read
4. Write into BRAM
BRAM0
BRAM1
The source and destination of the operations are reversed every cycle
Odd numbered position in read

Figure 3.5: The recursive unit

Later representations of the recursive units will assume the inclusion of a comparator block that picks the candidate that corrects bases with the lowest total quality score as the correct candidate, along with the recursion implementation presented above.

One other point needs to be noted so that subsequent diagrams are not confusing. For the case where a read has no solid k-mers in it, the error-correction routine needs to potentially correct bases at either ends of the read. Figure 2.5 described the method used to verify a correction at the end of a read using extensions beyond the read. In the implementation of the task that handles reads with no solid islands in them, therefore, two such extensions are needed, one at the left end of the read and another at the right end of the read. The extension toward the right end of the read is handled by the recursive unit executing the correction algorithm following the determination of the candidate replacements for the first k-mer in the read. The extension toward the left needs to be handled by a different recursive unit. Hence, in later representations of the task handling the correction of a read with no solid k-mers in it, there will be two types of recursive units operating one after the other.

Other units in the design

There are a few other hardware units in the error-correction block in addition to the recursive unit. They will be referred to using different names to indicate their impact on throughput, but none of them is as critical to overall throughput as the recursive units referred to above.

- Pipelined units. These are pipeline stages that convert one representation of the island map that represents the positions of weak islands in a read into another that is convenient for the units performing the correction to use. The throughput of these units does not depend on any external memory queries, and is solely determined by the throughput at their inputs and the back-pressure from the subsequent blocks.

- Iterative units. These are slower than the pipelined units but faster than the recursive units. The reason they are slower than the pipelined units is that their throughput depends on memory latency. They are faster than the recursive units because their next access of memory does not depend on the result of the last access. In the case of the recursive units, the next set of candidate corrections, and hence the k-mers to be queried from them depend on the set of candidates written into BRAM as a result of k-mer queries from the last set of candidate corrections (numbered step 3 in Figure 3.5). There are two types of iterative units in the design. One type operates after the recursive unit completes recording all possible candidate corrections for weak islands, when the weak islands are contained completely inside a read. This type takes each candidate correction from the recursive unit and checks k-mers from the solid part of the read that contain the modified bases in this candidate. The other type operates before the recursive unit when there are no solid k-mers in a read. This type tries to make a minimal number of changes to the first k-mer in the read to see whether it can find a solid k-mer to substitute into that position. This allows the recursive algorithm to correct the rest of the read as described in Section 2.2.3.

3.1.3 Extracting task-level parallelism

Errors in disjoint portions of a single read can be corrected independently. As shown in Figure 3.6, disjoint erroneous portions are separated by at least one solid k-mer. Four independent tasks running in parallel can be employed to correct four types of errors and the results can be appropriately combined into the correction of a single corrected read (note that the ends of the read are labeled 5'-end and 3'-end; these will be used as terminology without further elaboration). This can allow error correction to proceed faster due to the extraction of task-level parallelism.

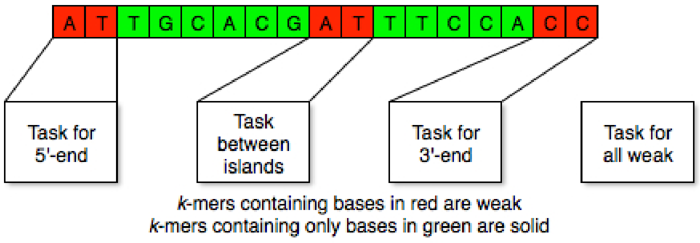
A T T G C A C G A T T T C C A C C
Task for 5'-end
Task between islands
Task for 3'-end
Task for all weak
k-mers containing bases in red are weak
k-mers containing only bases in green are solid

Figure 3.6: Task-level parallelism

Each of the tasks shown in Figure 3.6 uses one or more recursive units. In addition, each task may contain a variable number and type of pipelined and iterative units. So, overall these are very similar tasks, with customizations for dealing with a specific part of the read. However, the amount of work done by these tasks is not the same. This is because errors are not uniformly distributed in a read. It is observed that there are more errors in the 3'-end part of the read than in other parts. If a similar number of resources is allocated to each task, then there would be a load-balancing problem. Hence, resources need to be allocated to tasks in proportion to the work they do. In addition, the allocated resources need to address critical computations in each of the tasks. As mentioned before, the performance critical blocks are the recursive units. Hence more recursive units will be allocated to tasks that need to do more work. For example, the task addressing errors at the 3'-end of the read will get more recursive units. The allocation is decided based on the simulation of a sample set of reads from a complete dataset.

When the results of load-balanced tasks need to be combined there can still be some issues. This is because the load-balancing scheme discussed above may only balance the throughputs of the tasks on the average. There can still be throughput variations among the tasks from one input to another. Referring to Figure 3.7, two load-balanced tasks, Task A and Task B, operate on the same input and their outputs are combined. The tasks have equal average throughput. However, for any given input they can have different execution times. In this case, the faster executing task should hold its output and remain idle until the other task completes. The tasks will thus keep blocking each other creating idle cycles for portions of the hardware. To more efficiently use the hardware, extra memory is added at the output and at the input of each task. In this way, each task can write its output to local memory and proceed to the next input without waiting for the slower task. Since the tasks have the same average throughput in the long run, the number of times the tasks block each other can be reduced if a sufficient amount of local memory is provided.

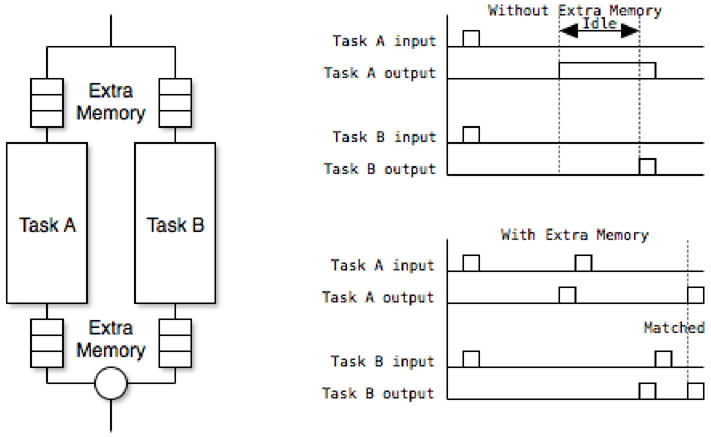

Figure 3.7: Averaging throughput variance

The nature of tasks in Figure 3.6 is identical to that of the tasks described in the preceding two paragraphs. The tasks can be balanced on the average, but not from one read to the next. This is because, for any particular read, there can be more errors in one part of the read than another, and there may be more candidates that need to be considered at intermediate steps of the recursion for that part. These depend on the statistics of the sequenced read dataset and is beyond the control of the algorithm to decide. Thus provisioning for load-balancing and throughput-averaging, the error-correction block in the hardware implementation is assembled as in Figure 3.8.

The input consisting of the read, 2-bit quality scores and the island map

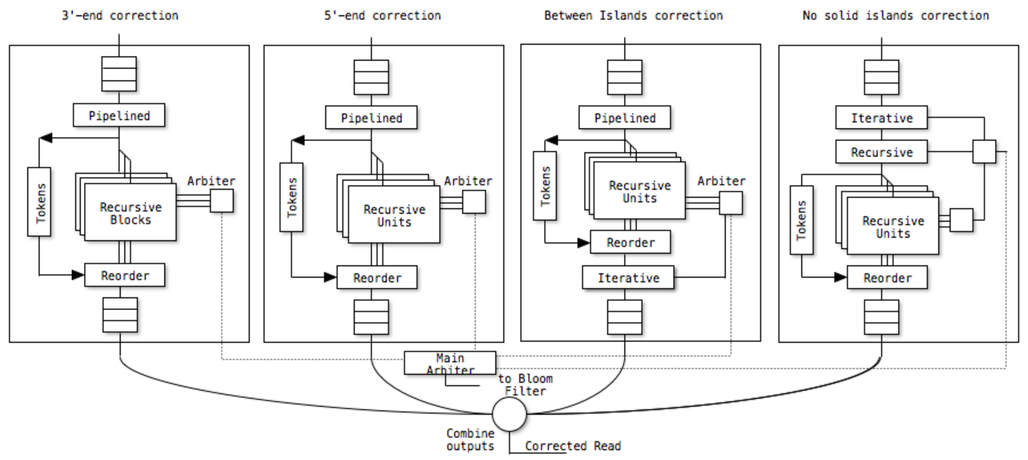

Figure 3.8: The architecture of the error-correction block

representation are sent to each of the tasks. The tasks each have a set of pipelined and iterative units. Where recursive units are used, provision is made for instanciating more than one of the units in parallel. Many recursive units operating in parallel can provide outputs out of order, because a recursive unit that accepts the second input may have a lesser amount of work to do than the recursive unit that accepted the first input. This makes it necessary to employ some form of reordering at the outputs of these units. A scheme similar to the one used to reorder memory request responses presented in Figure 3.3 is used here. In addition, on examining Figure 3.8, it becomes clear that many hardware units are operating in parallel that need to query for k-mers in the CBF. This means that some form of arbitration is required among these different units. In Figure 3.8, it may be noted that there are arbiters provided at various levels in the design that feed into a main arbiter. The arbiters are designed to assign priority to certain units that interface with them. For example, tasks related to 3'-end correction have higher priority in the main arbiter.

3.2 Experiments and Results

The methods presented in Section 3.1 were implemented in hardware and experiments were conducted using different benchmarks to examine the efficacy of the design. The results of the experiments are presented here.

3.2.1 Data preparation

To assess the performance of the accelerator, its accuracy and runtime were compared with those of BLESS, Musket [21], and Lighter [22]. Musket is a very fast and accurate tool that was the best in terms of execution speed in a previous evaluation work [23] and Lighter is a recently introduced super-fast error-correction algorithm that does not require counting the multiplicity of k-mers. The versions of BLESS and Musket used for the experiments were 0.12, 1.1, respectively. Lighter was downloaded from its repository on May 24, 2014.

Four read sets from two bacterium genomes and two human chromosomes were used for the evaluation. D1, D2, and D3 were downloaded from the National Center for Biotechnology Information (NCBI) Sequence Read Archive (SRA) and the Genome Assembly Gold-Standard Evaluations (GAGE) [24] website. D4 was generated using simNGS (http://www.ebi.ac.uk/goldman-srv/simNGS) from human chromosome 1 that is the longest chromosome in the human genome. The characteristics of the read sets are summarized in Table 3.1. The corrected reads were compared with the reference DNA sequences downloaded from the NCBI Reference Sequence Database, and their accuracy was evaluated using Error Correction Evaluation Toolkit) [25]. BLESS, Musket, and Lighter were evaluated on a server with two six-core Xeon X5650 processors and 24 gigabytes of memory.

3.2.2 FPGA resource usage

The design was fitted on an Altera Stratix V 5SGXEA7N2F45C2 FPGA. All configurations used four recursive functional units in the 3'-correction task and three recursive functional units in the no-solid-island correction task. All other units were instanciated once. The FPGA has 234,720 ALMs, 938,800 registers, and 52,428,800 memory bits. Results are summarized in Table 3.2.

3.2.3 Accuracy and running time

ECET uses sensitivity, gain, and specificity as the accuracy metrics. It counts the number of erroneous bases correctly modified (true positive, TP), the number of correct or erroneous bases erroneously changed (false positives,

Table 3.1: Datasets used for evaluation

ID	Genome	Reference	Read	Genome length (Mbase) [c]	Read length	Number of reads	Per-base Error (%) [d]
D1	*S.*aureus	NC010079, NC010063.1, NC012417.1	SRR022868	2.9	101	1,096,140	2.1
D2	*E.*coli	NC_000913	SRR001665	4.6	36	20,693,240	0.5
D3	Human Chr14	NC000014.8	N/A[a]	88.2	101	36,172,396	1.4
D4	Human Chr1	NC000001.10	N/A[b]	225	101	89,220,048	0.6

[a] Downloaded from the GAGE website

[b] Generated using simNGS

[c] Number of bases in reference sequences after all the bases that are neither A, C, G, nor T are removed

[d] How many substitution errors each read set has

Table 3.2: Summary of FPGA resource utilization

Dataset	k-mer length	ALM(%)	Register(%)	Memory(%)
D1	21	40	22	22
D2	19	25	15	9
D3	31	43	24	22
D4	45	44	25	22

FP), the number of erroneous bases unmodified (false negatives, FN), and the number of remaining bases (true negative, TN). Then, sensitivity, gain, and specificity are calculated using these quantities. Sensitivity, defined as TP/(TP + FN), shows how many errors in the input reads are corrected. Gain, defined as (TP - FP)/(TP + FN), represents the ratio of the reduction of errors to the total number of errors in the original reads. Specificity shows the fraction of error-free bases left unmodified, and it can be defined as TN/(TN + FP).

Accuracy and runtime of each method are shown in Table 3.3. The accuracy of the tools is sensitive to their key parameters. In order to obtain the best results using each software, we applied a contiguous range of numbers to the key parameters of each software and chose the combination of the parameters that makes gain of the method the highest. Lighter, and Musket support multiple cores and they were executed using all the twelve cores in the server while only one core was used for BLESS.

As summarized in Table 3.3, the implementation presented in this thesis performed faster than other methods in all cases. For D1, it was 43x faster than BLESS. For D1 and D2 it was more than 2x faster than Lighter, the fastest competitor. Accuracy of the hardware accelerator was better than that of Lighter and Musket for D1 and D3. The average speedup of the accelerator was 36.7x and in the worst case it was 28.2x faster than BLESS. The accuracy of the accelerator compares well with that of competing tools for all the cases.

Table 3.3: Comparison of accuracy and runtime of each error-correction method; suffix“(p)” indicates parallelized on 12 cores

Data	Tool	Accuracy			Runtime(s)	Speedup(x)
		Sensitivity	Gain	Specificity		
D1	Accelerator	0.883	0.879	1.000	6.35	1
	BLESS	0.895	0.895	1.000	273	43.0
	Lighter (p)	0.652	0.643	1.000	17	2.6
	Musket (p)	0.711	0.705	1.000	66	10.4
D2	Accelerator	0.927	0.923	1.000	31	1
	BLESS	0.969	0.967	1.000	1293	41.7
	Lighter (p)	0.941	0.926	1.000	88	2.8
	Musket (p)	0.936	0.928	1.000	115	3.7
D3	Accelerator	0.654	0.610	0.999	322	1
	BLESS	0.673	0.644	1.000	9099	28.2
	Lighter (p)	0.602	0.555	0.999	414	1.2
	Musket (p)	0.577	0.537	0.999	1649	5.1
D4	Accelerator	0.853	0.818	1.000	690	1
	BLESS	0.891	0.870	1.000	23728	34.0
	Lighter (p)	0.879	0.828	1.000	1047	1.5
	Musket (p)	0.888	0.866	1.000	3051	4.4

3.3 Conclusions

DNA error correction is an important process to be integrated into genomic workflows. While the quality of error correction is high in state-of-the-art tools, their throughput needs improvement. If the best error-correction tool could also be made the fastest, it would greatly benefit downstream analyses performed on DNA sequencing data. In this work, one of the most accurate DNA error-correction algorithms available today, BLESS, was successfully ported to an FPGA platform to create one of the fastest tools for correcting errors in Illumina reads.

Although the amount of speedup achieved is signficant, it must be noted that the implementation has not utilized the full capabilities of the FPGA device. However, it is designed to scale to a larger number of threads without hassles by simple adjustment of parameter values in the source code. There is also potential in using multiple FPGAs to further accelerate the algorithm as it is inherently data-parallel. It will also be interesting to see how this implementation would scale on a system such as BlueDBM [26].

Not all features in BLESS have been implemented in the accelerator, e.g., some heuristics applied to the initial island map to mitigate the effects of false positives of the Bloom filter are not included. Because some of these rules are yet to be implemented, the accuracy of the accelerator is not equal to that of BLESS, though it is comparable. In addition, a new version of BLESS, BLESS 2 [27], was released recently, which improves on the accuracy and speed of BLESS. To improve accuracy, BLESS 2 introduces new techniques to analyze quality scores better, as well as to trim reads where errors cannot be corrected satisfactorily. To improve runtime, it uses a new method to count k-mers [28], and also parallelizes the error-correction routine on multiple CPU cores. Future work will concentrate on combining the strength of the hardware implementation presented here to quickly find a list of candidate corrections with the strength of the new software techniques to more accurately pre-process reads and post-process candidate corrections by tightly integrating hardware and software routines. Some other directions worth exploring include combining functional units in different error-correction tasks to obtain a generic task that will partition resources automatically based on error statistics in the input data.

REFERENCES

[1] "The cost of sequencing a human genome." [Online]. Available: http://www.genome.gov/sequencingcosts/

[2] F. S. Collins and H. Varmus, "A new initiative on precision medicine," *New England Journal of Medicine*, vol. 372, no. 9, pp. 793–795, 2015, pMID: 25635347. [Online]. Available: http://dx.doi.org/10.1056/NEJMp1500523

[3] "Illumina website." [Online]. Available: http://www.illumina.com/systems/sequencing.html

[4] "Illumina HiSeq X system specifications." [Online]. Available: http://www.illumina.com/systems/hiseq-x-sequencing-system/performance-specifications.ilmn

[5] S. L. Salzberg, A. M. Phillippy, A. Zimin, D. Puiu, T. Magoc, S. Koren, T. J. Treangen, M. C. Schatz, A. L. Delcher, M. Roberts, M. Pop, J. A. Yorke, and G. Maraise, "GAGE: A critical evaluation of genome assemblies and assembly algorithms," *Genome Research*, vol. 22, no. 3, p. 15, 2012.

[6] M. D. MacManes and M. B. Eisen, "Improving transcriptome assembly through error correction of high-throughput sequence reads," *PeerJ*, 2013, 1:e113. [Online]. Available: https://doi.org/10.7717/peerj.113

[7] P. A. Pevzner, H. Tang, and M. S. Waterman, "An Eulerian path approach to DNA fragment assembly," *Proceedings of the National Academy of Sciences*, vol. 98, no. 17, pp. 9748–9753, 2001. [Online]. Available: http://www.pnas.org/content/98/17/9748.abstract

[8] H. Li, "Correcting Illumina sequencing errors for human data," *arXiv.org*. [Online]. Available: https://arxiv.org/abs/1502.03744

[9] Y. Heo, X.-L. Wu, D. Chen, J. Ma, and W.-M. Hwu, "BLESS: Bloom filter-based error correction solution for high-throughput sequencing reads," *Bioinformatics*, vol. 30, no. 10, pp. 1354–1362, 2014. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/30/10/1354.abstract

[10] G. Vacek, B. Hornung, J. Bolding, and S. Koren, “Accelerating error correction and assembly of single-molecule sequencing reads.” [Online]. Available: https://f1000research.com/posters/1093938

[11] Y. Chen, B. Schmidt, and D. L. Maskell, “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, April 2013.

[12] P. Krishnamurthy, J. Buhler, and R. Chamberlain, “Biosequence similarity search on the Mercury system,” *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 49, no. 1, pp. 101–121, 2007. [Online]. Available: http://dx.doi.org/10.1007/s11265-007-0087-0

[13] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, “Basic local alignment search tool,” vol. 215, no. 3, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022283605803602

[14] T. L. Madden, R. L. Tatusov, and J. Zhang, “Applications of a network BLAST server,” *Methods in Enzymology*, vol. 266, pp. 131–141, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S007668799666011X

[15] B. S. C. Varma, K. Paul, M. Balakrishnan, and D. Lavenier, “FAssem: FPGA based acceleration of de novo genome assembly,” in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, April 2013, pp. 173–176.

[16] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, “Hardware acceleration of short read mapping,” in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, April 2012, pp. 161–168.

[17] A. Ramachandran, Y. Heo, W.-m. Hwu, J. Ma, and D. Chen, “FPGA accelerated DNA error correction,” in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE ’15. San Jose, CA, USA: EDA Consortium, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2757012.2757131 pp. 1371–1376.

[18] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Commun. ACM*, vol. 13, no. 7, pp. 422–426, July 1970. [Online]. Available: http://doi.acm.org/10.1145/362686.362692

[19] F. Putze, P. Sanders, and J. Singler, “Cache-, hash-, and space-efficient Bloom filters,” *J. Exp. Algorithmics*, vol. 14, pp. 4:4.4–4:4.18, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1498698.1594230

[20] P. P. Chu and R. E. Jones, “Design techniques of FPGA based random number generator,” in *Military and Aerospace Applications of Programmable Devices and Technologies Conference*, vol. 1, no. 999, 1999.

[21] Y. Liu, J. Schrder, and B. Schmidt, “Musket: A multistage k-mer spectrum-based error corrector for Illumina sequence data,” *Bioinformatics*, vol. 29, no. 3, p. 308, 2012. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/bts690

[22] L. Song, L. Florea, and B. Langmead, “Lighter: Fast and memory-efficient sequencing error correction without counting,” *Genome Biology*, vol. 15, number 11, 2014. [Online]. Available: https://genomebiology.biomedcentral.com/articles/10.1186/s13059-014-0509-9

[23] M. Michael and I. Lucian, “Correcting Illumina data,” *Briefings in Bioinformatics*, vol. 16, no. 4, pp. 588 – 599, 2015.

[24] “Genome Assembly Gold Standard Evaluations.” [Online]. Available: http://gage.cbcb.umd.edu

[25] X. Yang, S. P. Chockalingam, and S. Aluru, “A survey of error-correction methods for next-generation sequencing,” *Briefings in Bioinformatics*, vol. 14, no. 1, p. 56, 2012. [Online]. Available: http://dx.doi.org/10.1093/bib/bbs015

[26] S.-W. Jun, M. Liu, K. E. Fleming, and Arvind, “Scalable multi-access flash store for big data analytics,” in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’14. New York, NY, USA: ACM, 2014, pp. 55–64.

[27] Y. Heo, A. Ramachandran, W.-M. Hwu, J. Ma, and D. Chen, “BLESS 2: Accurate, memory-efficient, and fast error correction method,” *Bioinformatics*, 2016.

[28] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, “KMC 2: Fast and resource-frugal k-mer counting,” *Bioinformatics*, vol. 31, no. 10, p. 1569, 2015. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btv022