An FPGA Acceleration of Short Read Human Genome Mapping

Corey Bruce Olson

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2011

Program Authorized to Offer Degree:
Department of Electrical Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a master's thesis by

Corey Bruce Olson

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Carl Ebeling

_____

Scott A. Hauck

Date:_____

In presenting this thesis in partial fulfillment of the requirements for a master's degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Any other reproduction for any purposes or by any means shall not be allowed without my written permission.

Signature _____

Date _____

University of Washington

Abstract

An FPGA Acceleration of Short Read Human Genome Mapping

Corey Bruce Olson

Chairs of the Supervisory Committee:

Professor Carl Ebeling
Computer Science and Engineering

Professor Scott Hauck
Electrical Engineering

Field-programmable gate array (FPGA) based systems offer the potential for drastic improvement in the performance of data intensive applications. In this thesis, we develop an algorithm designed to efficiently map short reads to a reference genome using a reconfigurable platform. We also describe the hardware design of the algorithm running on an FPGA, and we implement a prototype to map short reads to chromosome 21 of the human genome. We analyze the results of the prototype system running on Pico Computing's M501 board, and describe necessary changes for implementing a full scale version of the algorithm to map reads to the full human genome. We then compare our results to BFAST, which is a current software program commonly used for short read mapping, running on a pair of Intel Xeon quad core processors with 24GB of available DRAM. The prototype system achieves a 14x speedup compared to BFAST for mapping reads to chromosome 21, and the hardware system implementing the full human genome mapping is projected to have a 692x improvement in system runtime and a 513x improvement in power.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Pico Computing (Seattle, WA) and the Washington Technology Center (Seattle, WA) for giving me the opportunity to perform this research by providing the funding and resources that make it possible. I would also like to express my deep thanks to Professor Carl Ebeling, Professor Scott Hauck, and Professor Larry Ruzzo for providing guidance, inspiration, and assistance to me throughout this project. I would also like to thank the other students involved in the research project, including my co-researcher Maria Kim, and research teammates Boris Kogon and Cooper Clauson, for making this research project a success. Finally, I would like to thank my girlfriend Ashley, my family, and my friends for the support I have received throughout this project; without them, this thesis would not have been possible.

1 Introduction

Genomics is an emerging field, constantly presenting many new challenges to researchers in both biological and computational aspects of applications. Genomics applications can be very computationally intensive, due to the magnitude of data sets involved, such as the three billion base pair human genome. Some genomics applications include protein folding, RNA folding, de novo assembly, and sequencing analysis using hidden Markov models.

Many of these applications involve either searching a database for a target sequence, as done with hidden Markov models, or performing computations to characterize the behavior of a particular sequence, as performed by both protein and RNA folding. Genomics applications therefore require immense quantities of computation, and they also tend to require the movement and storage of very large data sets. These data sets are either in the form of a database to be searched, as in sequence searching with hidden Markov models, or in the form of a sequence to be computed and operated upon, such as the sequence of amino acids of a protein being folded. These two challenges tend to lead to long execution times for software programs, which can result in a tool being unusable for researchers.

One interesting application in genomics is short read genome mapping. Short read mapping attempts to determine a sample DNA sequence by locating the origin of millions of short length reads in a known reference sequence. Genome mapping is performed in this manner, because machines that sequence these short reads are able to do so in a very parallel manner, producing reads at a much faster rate than traditional sequencing technology. Base-calling machines, such as Illumina's HiSeq 2000 [1] or Applied Biosystems' 5500 Genetic Analyzer [2], are currently capable of producing millions of reads per day, and their throughput is improving at an exponential rate.

This exponential improvement in sequencing also requires the improvement of short read mapping, in order to keep up with the throughput of the machines. Numerous software tools are readily available for short read mapping, including BFAST [3], Bowtie [4], BWA [5], MAQ [6], and SHRiMP [7]. However, these current software tools (which are designed to

run on traditional CPUs) are failing to keep up with the rapid throughput improvements of the base-calling technologies creating the short reads.

Field-programmable gate arrays (FPGAs) contain large amounts of programmable logic, which can be configured to perform any computation that fits within the FPGA. FPGAs can offer a dramatic increase in performance for computationally intensive applications by performing large numbers of computations in parallel. An implementation of a short read mapping algorithm on an FPGA could exploit the inherent parallelism in the problem, thereby leading to a drastic reduction in short read mapping time. Previous attempts of naïve short read mapping on an FPGA [8] have yielded small speedup versus three software tools on a standard processor, but these could be further developed to greatly improve the runtime and power consumption of short read genome mapping.

This thesis develops an algorithm targeting a reconfigurable platform for the acceleration of short read human genome mapping. It also implements a prototype of that algorithm on a system designed to map short reads to chromosome 21 of the human genome. Finally, it proposes changes to the designed algorithm and discusses how the prototype system can be modified for mapping against the full genome. The following sections of this thesis are organized as follows:

- **Chapter 2: Background** provides background information on DNA, short read mapping, and FPGAs.
- **Chapter 3: Algorithm** develops the proposed algorithm to efficiently map short reads to the human genome.
- **Chapter 4: Alternatives** discusses alternative solutions to the short read mapping problem that were considered during the project.
- **Chapter 5: System Design** provides details of the hardware system implementation as well as system performance results.
- **Chapter 6: Future Improvements** describes improvements to the current system to allow for fast and accurate full genome mapping.

2 Background

2.1 Reading DNA

DNA is a basic structure present in living cells and contains the genetic code used to construct other cell components, such as proteins and RNA molecules. For years, scientists have attempted to study DNA and its effect on various health concerns, but have struggled to accurately isolate, characterize, and describe genes or noncoding regions of the DNA sequence. Study of DNA could identify possible genetic defects and help to avoid serious health concerns for an individual.

In order to read an organism's genetic sequence, and thus compare it to a "healthy" or "normal" genome, scientists must first be able to determine a sample DNA sequence by reading the sequence of nucleotide bases. Since human DNA is a three billion base pair sequence, this is a very difficult computational and biological problem.

2.1.1 Human Genome Project

The Human Genome Project was a 13 year government-sponsored project created to determine the composition of human DNA. The project began in 1990 and was completed in 2003, at a total cost of approximately three billion dollars [9]. During that time, the project was able to identify all the genes in human DNA, determine a consensus three billion base pair sequence that makes up the reference human genome, store the genome information in databases, and improve the sequencing tools for future analysis [9].

In order to create the reference genome, the DNA of numerous individuals was sequenced, and a consensus sequence was compiled to form the reference genome. This process was only possible due to the fact that the genetic sequence of all humans is very similar. Therefore, the reference sequence is created through a process of comparing the sequence of all individuals considered and determining a consensus base at every position in the genome, as shown in Figure 1.

One possible difference between an individual's DNA and the reference sequence is a situation where one nucleotide base has been replaced by a different base from the reference sequence at a given position, which is called a single nucleotide polymorphism (SNP).

Another difference that may occur is a base from the reference genome, or a consecutive sequence of bases, that does not appear in the sample DNA, and is known as a deletion. Similarly a base, or consecutive sequence of bases, may appear in the sample DNA that does not appear in the reference genome, and is known as an insertion. Insertions and deletions are referred to as *indels*. Lastly, the human genome may contain a pattern that occurs numerous times consecutively; however, the number of times that pattern occurs at the location in the reference may be different than the individual's DNA, and this is known as a copy number variation (CNV).

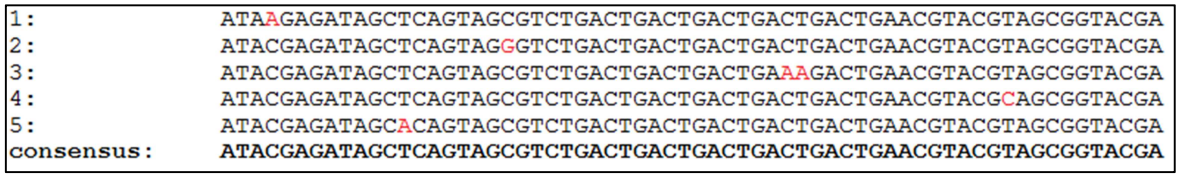

Figure 1: Construction of the consensus sequence based on samples from many individuals is possible because human DNA is 99.9% similar. Red indicates a difference in an individual's DNA sequence from the reference genome at that base.

After the creation of the reference genome, researchers can study genetic variations in individuals and can potentially identify and treat disease that may be caused by genetic defects. Other potential uses for this technology include DNA forensics, evolutionary studies, and agricultural improvements. However, a major problem faced by the human genome project was the time and cost of sequencing a single human DNA sequence. By 2002, the project was able to sequence 1.4 billion bases per year at an estimated cost of nine cents per base [9]. At this rate, a single human DNA sequence would take approximately two years to sequence and would cost $270,000. Both the cost and the throughput of the sequencing technology needed to be greatly improved in order for DNA sequencing to be beneficial.

2.1.2 Next-Generation Sequencing

DNA sequencing technology began from traditional Sanger sequencing in 1977 and has evolved into two cutting-edge technologies with very different methodologies; however, each have similar basic principles. The overall goal of both methodologies is to determine the biological DNA sequence, which can then be studied and analyzed. To determine the sequence of bases of a DNA sample in parallel, the DNA sample is first replicated to produce

approximately 30 copies of the original DNA sequence. The copies of the replicated sequence are then cut at random points throughout the entire length of the genome, as shown in Figure 2, producing short lengths of intact chains of base pairs known as *short reads*. This methodology has enabled these short reads to be read in a massively parallel fashion; however, accuracy limitations in base-calling, which is the process of determining the sequence of bases within a read, has restricted read lengths to be much shorter than previous Sanger sequencing reads.

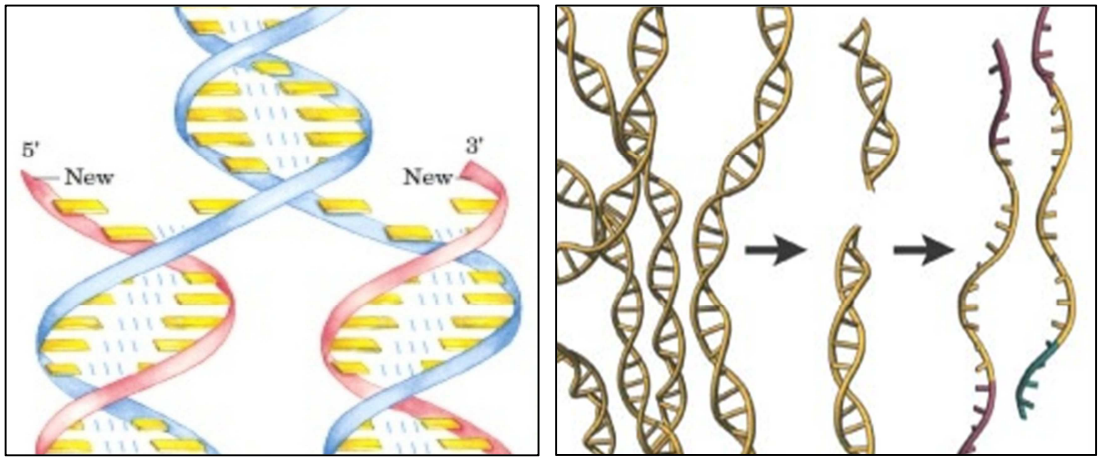

Figure 2: DNA is replicated (left) by unzipping the original double helix structure, thus allowing nucleotide bases to attach to the exposed bases [10]. Duplicate DNA strands (right) are then cut at random before base-calling [11].

The sequencing technologies differ slightly in their base-calling approach. Pyrosequencing technology, used in the 454 Life Sciences sequencer from Roche, determines the sequence of bases in a read by sequentially flowing bases over templates that are captured in tiny microscopic wells [12]. The nucleotide base sequence is then determined by measuring the intensity of fluorescence, where a brighter luminosity indicates consecutive homogenous bases. Pyrosequencing produces reads that are approximately 400 base pairs in length, and a single run generates hundreds of millions of nucleotide bases. An illustration of this technology can be seen in Figure 3.

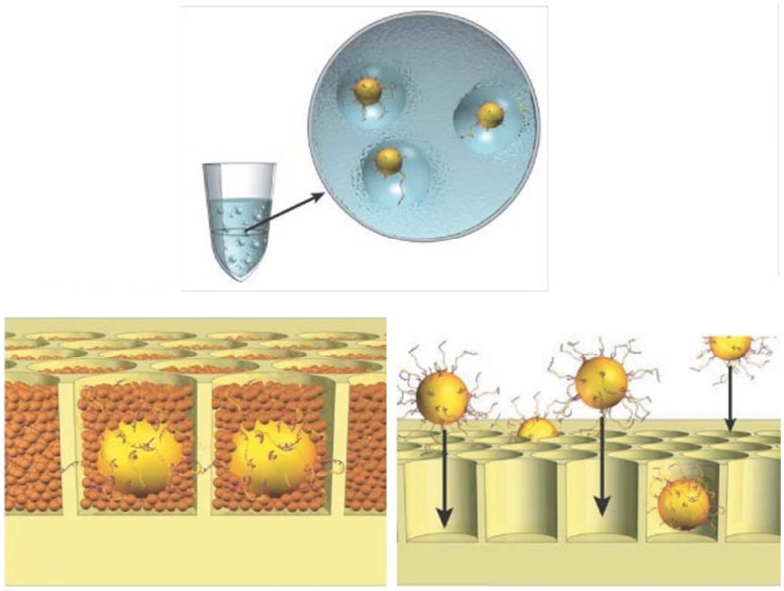

Figure 3: Pyrosequencing. Beads carrying single stranded DNA fragments are deposited into wells of a fiber-optic slide. Smaller beads with a pyrophosphate enzyme for sequencing are then deposited as well [11].

The alternative technology, used by Illumina, Applied Biosystems, and Helicos sequencing machines, targets only one base of the read per cycle but is able to complete this process in a much more parallel fashion. This technology decodes the read one base at a time by iteratively attaching the complementary base of the next base in the read, along with a fluorescent tag to identify the base and terminator to ensure multiple bases do not attach at the same time to the same read. Next, the process reads the fluorescent tag attached to the base with a laser. Finally, the tag and terminator are removed so the process can be repeated for the next base, as shown in Figure 4 [12]. This technology produces 100 base pair reads, which are much shorter than reads produced by pyrosequencing, but 20-30 billion bases can be sequenced per run, yielding a much lower cost per base in comparison to pyrosequencing.

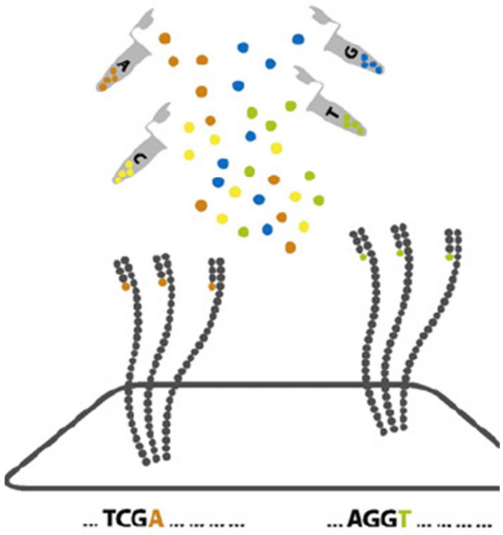

Figure 4: Sequencing technology used by Illumina attaches a nucleotide with a fluorescent tag to the next base in the read, images the read to determine the base, and removes the fluorescent tag in order to move onto the subsequent base. [13]

Neither of these systems is capable of producing perfect results, and errors arise in the base-calling steps for many reasons. In pyrosequencing, if a read contains ten consecutive adenine bases, it could be mistaken as being eleven adenine bases during the base calling step. Similar errors can occur in the alternative sequencing technology as well, and base-calling errors are a leading factor that limits the read length.

After sequencing, the short reads must be reassembled by a computer program into the full genome. This can be accomplished in two ways. The first technique, called *de novo* assembly, attempts to reconstruct the genome with no external information by looking for overlapping subsequences of multiple short reads and attempting to chain them together to form longer sequences, which are called contigs. Assuming no read errors are present and enough data is available from the entire genome, this would be analogous to putting together a 100 million piece puzzle.

The second method, known as short read mapping, uses the fact that a reference genome is known, and that the DNA of all members of a species is very similar to that reference genome. Short read mapping attempts to determine where in the reference genome a particular read came from. Due to read errors and slight differences between the sample DNA and the reference genome, the reads will not map perfectly to the reference, but their

relative position can still be used to reconstruct the individual's DNA sequence. Figure 5 shows a set of short reads mapped to a very small section of a reference genome. The red bases represent differences from the reference genome; the red 'C' shows an SNP whereas the red 'T' shows a base-calling error.

```
Reference:      ATAAGAGATAGCTCAGTAGCGTCTGACTGACTGAATAGCTCAGTGAACGTACGTAGCGGTACGA
Reads:          ATAAG     AGCTCAGTAG      GACTGACTGA       GACTGAACGT        GGTACGA
                ATA    GATAGCTCAG AGCCTCTGAC        ATAGCTCAGT  ACGTACGTAG     TCGA
                A AAGAGATAGC   AGTAGCCTCT   CTGACTGAAT    TCAGTGAACG     TAGCGGTACG
                    GAGATAGCTC    AGCCTCTGAC    CTGAATAGCT     GAACGTACGT    GGTACGA
                ATAAGAGA        GTAGCCTCTG    GACTGAATAG TCAGTGAACG  CGTAGCGGTA
```

Figure 5: Set of reads aligned to reference sequence. Base C in red shows a difference between the sample DNA and the reference. Red base T shows a read error.

Next-generation sequencing machines have been rapidly improving in throughput and cost over the last ten years, as seen in Figure 6. Creative projects and monetary prizes for advancements in this field, such as the Archon X PRIZE for Genomics of ten million dollars to be awarded to the team that can design a system capable to sequencing 100 human genomes in 10 days [13], have supported these rapid technological improvements.

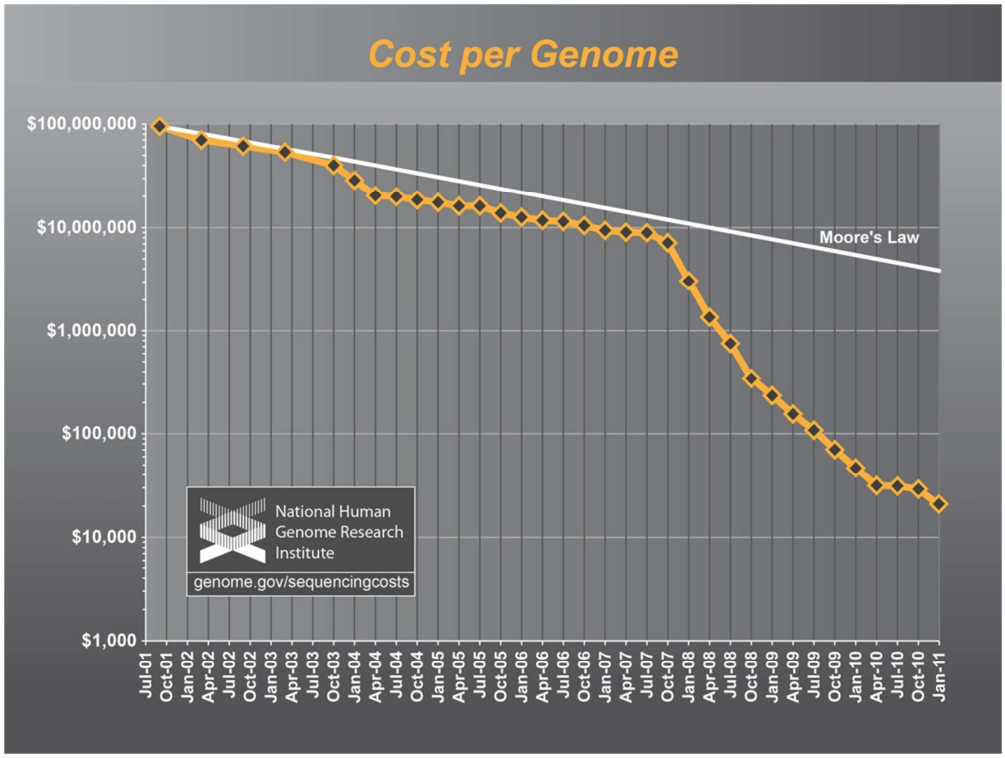

Figure 6: Recent costs to sequence a DNA sample compared to Moore's Law. [14]

Improvements in sequencing technology have not only produced the ability to rapidly sequence the DNA of numerous individuals, and even various genomes, but also resulted in a much lower cost per run. These improvements are so great that the computational analysis of the sequenced data is quickly becoming the bottleneck for human genome analysis. High-throughput machines, such as Illumina's HiSeq 2000 system can sequence 55 billion bases per day [1]. Comparatively, BFAST, which is a short read mapping software program, running on a 16-core computer would take 3.5 days to map that same amount of data.

2.2 Short Read Mapping

As described previously, short read mapping attempts to map reads to the reference genome in order to reconstruct a DNA sample. This may seem odd in that the goal is to reconstruct a DNA sample that is almost the same as a known reference genome; however, it is these small differences that may have interesting genetic mutations, which if studied could lead to a greater understanding in the cause of certain genetic diseases. Unfortunately, short read mapping requires a large amount of system memory and is extremely computationally intensive, which leads to long execution times in standard processors.

2.2.1 Algorithms

Short read mapping can be implemented with various algorithms, but it is most commonly accomplished with either an indexing-based solution or by using the Burrows-Wheeler transform [15]. The indexing based solution attempts to find subsequences of each read that match perfectly to a location in the reference genome, while the Burrows-Wheeler transform starts with a large list of possible locations to which the read aligns and iteratively reduces the list to a small set of locations. The indexing solution relies upon a final step that involves using a Smith-Waterman [16] string matching algorithm, which determines the best matching location of the read in the reference. In the next sections we discuss each of these approaches in turn.

2.2.1.1 Indexing Solution

Both solutions take advantage of the fact that all reads came from a DNA strand that is very similar to a known reference genome. If the sample DNA were an exact replica of the reference genome, and if there were no duplication or base-calling errors while generating the short reads, then each of the short reads should match perfectly to one or more locations

in the reference. If this were true, a simple solution would be to create an index of the entire reference genome that identifies the location in which each possible 76 base pair combination occurs in the reference genome, assuming 76 base pair read lengths. Figure 7 shows an example creation and use of the simple indexing solution for 10 base pair short reads.

Figure 7: Simple indexing solution shows the creation of the index table (red) by walking along the reference and recording the reads that appear at every location in the index table. Short reads are used as an address into the index table (green), and the final alignment for the read is read from the index table (blue).

Recall that the sample DNA is slightly different from the reference genome, and errors do occur while generating the short reads, thus not all reads are guaranteed to perfectly match a location in the reference genome. However, shorter subsequences of the short read, even in the presence of genetic differences and read errors, should match perfectly to at least one location in the reference, as can be seen in Figure 8.

Figure 8: In the presence of a single nucleotide polymorphism (SNP) (shown in red), the read will not map to the proper location in the reference, but shorter subsequences of the read, called seeds, will still map to the reference if they do not contain the base with the SNP.

Assuming low enough genetic variation and read error rates, some of these shorter subsequences, called *seeds*, should match perfectly to at least one location in the reference genome. Because the seeds will match perfectly to the reference, an index of the reference can be created for the desired seed length that tells at which positions in the reference a given seed occurs.

Read errors and genetic variations can occur at any position in a read. An error at a position in the read means that any seed that uses that base will not map to the correct portion of the reference genome, as can be seen in Figure 9 by the seed that causes the read to map to location 41, which is shown in red. To combat this, all possible seeds within a read are looked up in the index, and the read will have to be further compared at each of the reference locations specified by the seed lookups.

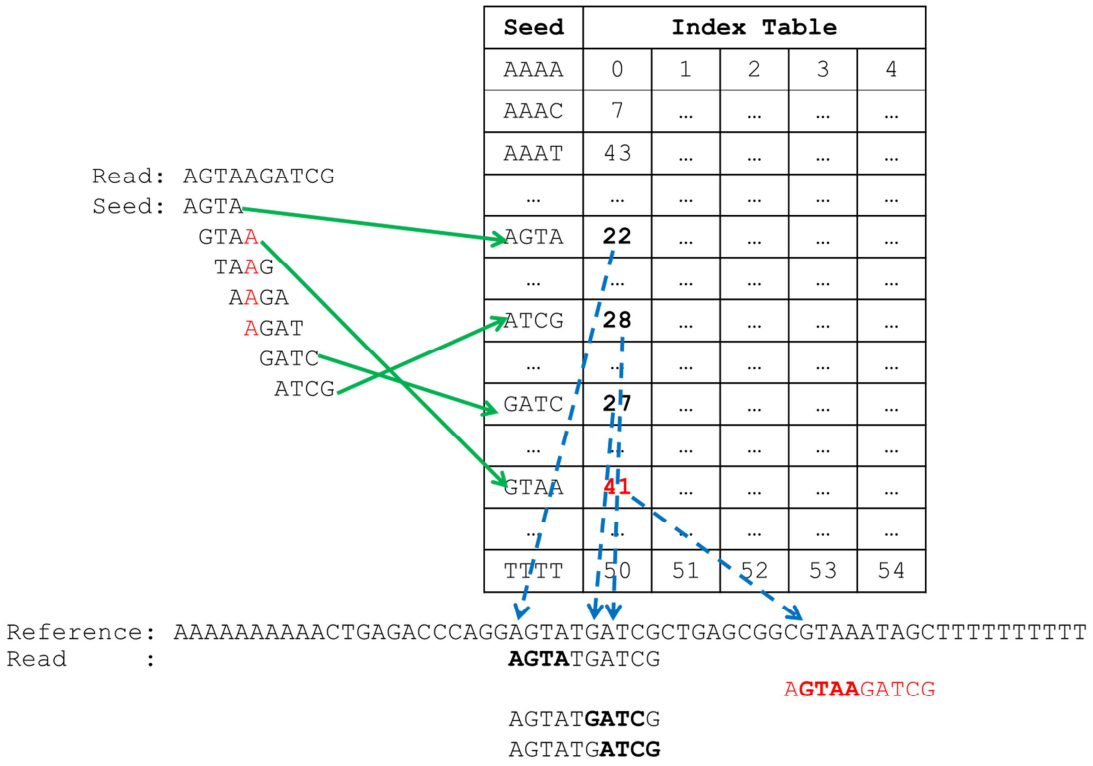

Figure 9: All seeds for a read are looked up in index table. Those seeds without the SNP produce correct alignment positions in the reference, while the seeds with the SNP may produce an incorrect alignment location.

The final comparison that is done for each specified position can vary in the algorithms that use the indexing based solutions. Some algorithms, such as MAQ [6], take the portion of the read that matches perfectly to the reference genome and extend the seed from its boundaries within the read one base at a time in an attempt to quantify how well the read matches that position in the reference. If the seed is fully extended so the entire read is matched to the reference, then the read matches perfectly to the reference at the test location. If not, this extension process attempts to quantify how well it was able to extend the seed. An example of this seed and extend method can be seen in Figure 10.

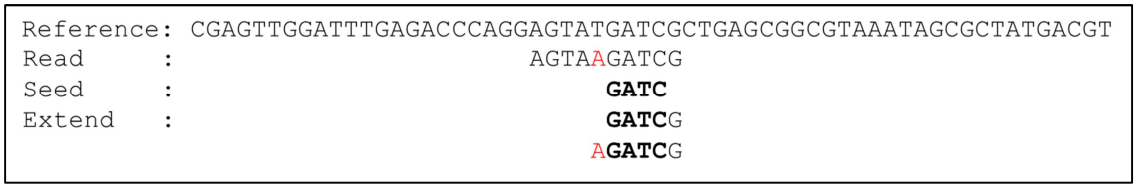

Figure 10: In seed and extend method, a possible alignment location is found for a read with the seed method, and the seed is extended to determine how well it aligns at the location.

The other manner of testing how well a read matches a location in the reference, as is the case in BFAST [3], is by using a fully gapped alignment algorithm very similar to the original Smith-Waterman [16] string matching algorithm. This is a dynamic programming algorithm that iteratively attempts to compare subsequences of the two strings in order to quantify the similarity of the two subsequences. The algorithm grows the subsequences, starting from an empty set, and adds a base to either the reference or read subsequence until the full reference and read strings are compared. At each step, a similarity score is assigned to the subsequences being compared, where the score is based only upon the similarity score of shorter subsequences of the read, the reference, and the base(s) that were just added to the current subsequence for comparison. This will be described in greater detail in section 3.1.

After each of the possible locations identified by the seed index lookup have been scored, the one with the highest similarity score, as identified by either the Smith-Waterman step or the seed extension step, is declared to be the alignment location for the read. The indexing solutions can quickly provide a list of possible alignment locations for a read, but they still require the Smith-Waterman alignment algorithm to determine which of the positions produces the best alignment. This process can be very time consuming, especially when the number of locations for each read is large. Also, the index requires a large memory footprint because approximately all locations in the reference produce a seed, with the exception of the end of chromosomes.

2.2.1.2 Burrows-Wheeler Transform Solution

The second category of algorithm that attempts to solve the short read mapping problem uses a data compression structure called the Burrows-Wheeler Transform (BWT) [15] and an indexing scheme known as the FM-index [17]. The core idea for searching the genome to find the exact alignment for a read is rooted in suffix trie theory. The BWT of a sequence of characters is constructed by creating a square matrix containing every possible rotation of the desired string, with one rotated instance of the string in each row, as shown in Figure 11. The matrix is then sorted lexicographically, and the BWT is the sequence of characters in the last column, starting from the top.

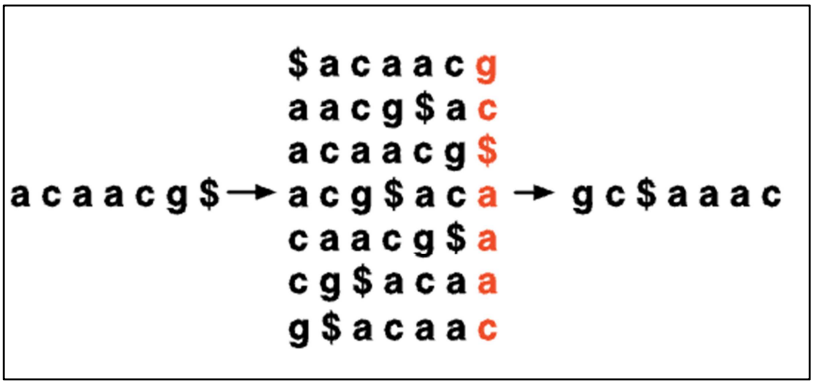

Figure 11: Creation of the Burrows-Wheeler Transform for the sequence 'acaacg' [4]. The constructed matrix is shown in the middle, and the final compressed BWT is 'gc$aaac', as shown on the right.

Once the BWT matrix has been constructed and sorted, rotations of the original string that begin with the same sequence will appear in consecutive rows of the BWT matrix. At this point, the original sequence can be searched for a target subsequence by iteratively growing the suffix of the target sequence and computing the range of indices of the BWT matrix that contain the target suffix.

During each iteration, the algorithm prepends one base of the target sequence to the current suffix and locates the first and last occurrences of the newly added base in the Burrows-Wheeler Transform that lies within the range of indices that was calculated in the previous iteration. If the base is found within the defined range of the BWT, the algorithm uses those occurrences in the BWT to compute the new range of indices into the BWT matrix for the next iteration, based on the current index range. This is continued until the entire target sequence is added to the suffix, and then the reference locations specified by the BWT within the defined range are where the target sequence occurs within the genome. Figure 12 shows an example using the BWT matrix to map 'aac' to the reference 'acaacg.'

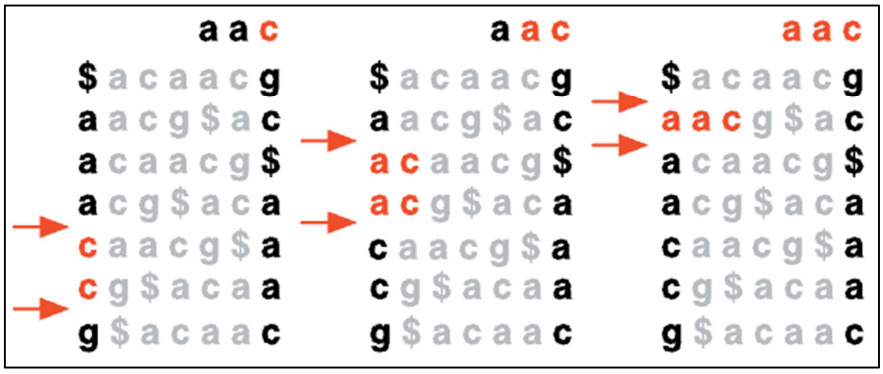

Figure 12: BWT matrix is used to compute the location of string 'aac' in the original sequence by iteratively prepending a base to the suffix of the search string, then computing the BWT table index limits based upon the limits computed in the previous iteration. [4]

For example, assume the user is trying to match the sequence 'aac' somewhere in the reference sequence 'acaacg'. The first step is to prepend the base 'c' to the originally empty suffix and locate the first and last occurrence of 'c' within the defined range of the BWT. Since the range begins as the full range of the BWT, the full BWT must be searched for the first iteration. The BWT at the first and last 'c' entry is then used to compute the index range for the next iteration, which will attempt to prepend the base 'a' to the suffix to form 'ac'. This repeats until all bases have been prepended to the suffix, and the locations of the target sequence have been found in the reference sequence.

The BWT has been shown to be a rapid way to search for a substring within a reference in the case of perfect matching, because each iteration the size of the index range for the current suffix either remains the same size or reduces in size, and therefore converges to a result. In other words, the search space for the target string begins with the entire reference and converges to only the points where the target sequence appears. However, problems arise with this algorithm in the presence of read errors or genetic differences.

For example, assume the next base to be prepended to the suffix is 'c', but no 'c' bases are found within the valid range of the BWT due to a genetic mutation from the reference genome. This situation means the 'c' base is either an SNP, an indel, a base-calling error, or one of the bases before this base was some form of an error. This leads to an exploding search space and can cause slow runtimes for large numbers of read errors or genetic variations.

2.2.2 Comparison

These two solutions to the short read mapping problem present different advantages. The indexing solution is more accurate because of the fully gapped alignment and therefore can tolerate a larger read error rate. However, it requires an extra Smith-Waterman alignment step, yielding slower short read alignments unless the Smith-Waterman step can be accelerated. Finally, even if all seeds for a read appear only in one location in the reference genome, the algorithm still must perform all the memory accesses to an index. This means the number of access to the index for the indexing solution is statically determined by the read length and seed length.

Conversely, the BWT-based solution does not have a static seed length; it can recognize if a short read occurs in only one location in the reference genome, and if so, halt the processing for this read, yielding faster processing times. The BWT-based solution is fast for reads with low error rates, both read errors and genetic differences, but it may not align all reads due to too many differences between the reference genome and the reads.

Software solutions, such as Bowtie [4] and BWA [5] can align approximately seven billion base pairs per CPU day, while advanced Illumina base calling machines, such as the HiSeq 2000, can sequence up to 55 billion base pairs per day [18]. One potential solution to this bottleneck is to accelerate the mapping phase by taking advantage of the data parallelism in the set of reads. The alignment of one read is inherently independent of the alignment of another read, and therefore ideally all reads could be aligned in parallel. This can be done by designing software to be run on expensive, large-scale computing systems composed of general purpose processors, or by designing a parallel hardware system, which does not have the restriction of sequential execution that processors have, thereby enabling the concurrent mapping of multiple reads.

2.3 FPGAs

One issue with using central processing units (CPUs) to do computationally intensive data processing is their inherent sequential nature. In other words, traditional single threaded CPUs execute a sequential program and are unable to take advantage of the parallelism that exists in some applications, either data or task level parallelism. General purpose processors attempt to take advantage of parallelism in applications by utilizing numerous processing

cores and partitioning the computation amongst those cores, as is done in multi-core processors or clusters. However, adding multiple cores for parallel processing requires the addition of more logic than the required computational unit, and that extra logic simply adds power and delay overhead to an application.

Conversely, hardware systems can be designed to exploit the parallelism that exists in a problem in order to produce a much higher computational throughput as compared to a CPU. Hardware chips that are designed and fabricated to target a specific design problem are called application specific integrated circuits (ASICs). ASICs can perform computations in parallel, resulting in a great speedup over a CPU. However, ASICs require a large amount of non-recurring engineering (NRE) costs in order to be produced, both for the design and fabrication of the chip. This NRE cost can be so large that some projects simply cannot produce enough income to cover the NRE costs, and therefore the project is ruled a failure.

Field programmable gate arrays (FPGAs) can provide a middle ground, which proves very useful in some applications. FPGAs are a sea of reconfigurable logic and programmable routing that can be connected and configured to mirror the function of any digital circuit. The combinational logic in an FPGA is implemented using static random-access memory (SRAM) based lookup tables (LUTs), and the sequential logic is implemented using registers. They also contain additional features, such as digital signal processing (DSP) blocks and large memories with low latency access times. This allows FPGAs to perform massively parallel computation, giving them a performance advantage over CPUs, but can also be reconfigured to fit multiple applications, which can reduce and almost eliminate the NRE costs of an ASIC.

The fine-grained parallelism in FPGAs gives them a performance advantage over traditional CPUs for applications requiring large amounts of computation on large data sets. The ability to be reconfigured gives them a reusability advantage over ASICs. However, both advantages are not without cost. Programming of FPGAs has traditionally been difficult because the designer has been required to write in a low-level hardware description language (HDL), such as Verilog or VHDL, instead of common and easier to debug high level software languages like C++. Also, the ability to be reprogrammed puts the FPGA at a large speed and power disadvantage when performing logical functions as compared to standard

CMOS gates that are implemented in ASICs. In spite of these costs, FPGAs can produce great improvements in both system runtime and power savings.

2.3.1 LUT

The lookup table (LUT) is the computational engine of the FPGA, because it implements the combinational logic required for all computations. Lookup tables are SRAM-based elements that take an input N-bit number, which is used as an address into an array, and returns a 1-bit number. By programming the elements of the array in the lookup table, a designer can implement any N-to-1 logical function. The SRAM based lookup table provides re-programmability to the system, but it causes the system to run at an average of about 4x slower clock speeds than standard CMOS logic [19].

2.3.2 Memory

Random access memory blocks are available on the FPGA to be used as local storage as needed by the application. This memory can be used as very fast access temporary data storage for kilobits of data during operation, but larger data sets must be stored in off-chip memory.

Since lookup tables are implemented using SRAM cells, some FPGA vendors such as Xilinx Inc., allow the lookup tables, which are distributed throughout the FPGA, to also be used as a distributed memory. Distributed memory can be great for shift registers, first-in-first-out queues (FIFOs), or can even be chained together to create larger memories, if necessary.

2.4 DRAM

Dynamic random-access memory (DRAM) is a type of memory that stores data in a very compact and efficient manner. DRAM retains data by storing charge on a capacitor, thereby creating a densely packed memory; however, the process of reading the stored data also destroys the data that was stored. Also, the capacitor loses its charge over time, and without being re-written, the stored data will be lost. To protect against this, DRAMs have a controller that continually reads and re-writes all the bits of data that are stored in the DRAM array. DRAM's density makes them perfect for storing data sets that are too large to fit into on-chip SRAM, but access into a random location of DRAM is slow.

DRAM's hierarchical organization comprises multiple banks, where each bank contains many pages, each page contains many rows, and each row contains many columns. In order to read a location in DRAM, the proper bank, page, and row must first be opened by the DRAM controller, which takes many clock cycles. Once a row is opened, large blocks of data can be read in consecutive clock cycles, until a new row must be opened. Thus, sequential access into DRAM can be fast and efficient, but random-access is very slow. Therefore, a designer using DRAM wants to read from the memory in a sequential access pattern, as opposed to random access, in order to maximize memory bandwidth.

To further improve the bandwidth to the DRAM, the controller can have multiple banks of the DRAM open at a time, which allows for a user to read from as many rows in the system as there are banks without the time penalty of opening a new row. Also, by pipelining reads and writes to the DRAM, the controller can overlap access times to different banks with the time penalty to open a new row. Double data rate (DDR), which is the transmission of data on the rising and falling clock edges, doubles the DRAM data transmission bandwidth without doubling the clock frequency.

2.5 Pico Computing

Pico Computing offers a reconfigurable computing system, which was used as the development platform in this research project. The machine comprises a computer running Linux and a set of processor daughter card (PDC) backplanes. Each backplane may hold up to six reconfigurable modules, which will be described in section 2.5.1, containing an FPGA and some memory. The machine has two quad-core Intel processors, a 1 TB hard drive for storage of large data sets, as required by genomics processing, and 24GB of main memory.

2.5.1 M501 and M503

The two Pico Computing modules of interest to this research project are the M501 and M503 modules. Both of these modules contain a single LX240T Virtex-6 FPGA, a PCI Express host interface, and DRAM. Figure 13 shows an image of the M501 board while Figure 14 shows the front and back of the M503.

Figure 13: M501 board from Pico Computing, with the Virtex-6 FPGA visible from the top view. [20]

The LX240T Virtex-6 FPGA, which is a medium-sized FPGA in the Virtex-6 family, provides approximately 240,000 logic cells [21] to the designer, which is quite large for most applications. The x8 PCI Express provides the communication interface with the host processor. One limitation of the M501 module is the relatively small amount of DRAM available, which is a maximum of 512MB. However, six M501 modules may be placed on the PDC backplane, allowing for massive computation in the system.

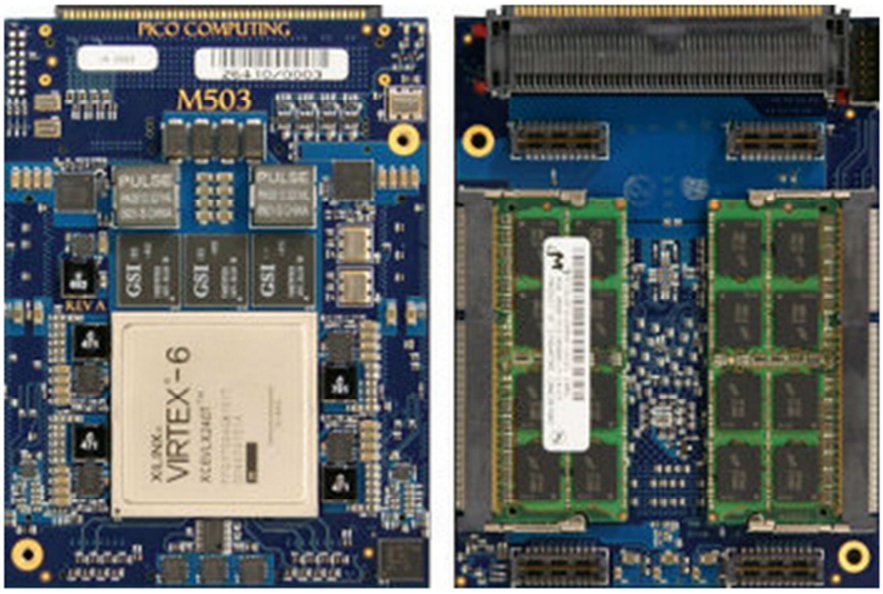

Figure 14: M503 board from Pico Computing. Virtex-6 FPGA can be seen from the top view (left) and the two 4GB DDR3 modules are visible on the underneath side of the module (right). [20]

The larger M503 card contains the same Virtex-6 FPGA as the M501 module, but contains many upgraded systems on the board. The communication to the host processor is done through x8 Gen2 PCIe. The amount of DRAM possible on the module has been improved to 8GB (two 4GB SODIMMS), and DDR3 DRAM is used as opposed to the DDR2 DRAM on the M501 module. Lastly, three 9MB QDRII SRAM chips are on the M503 module, providing fast off-chip memory that is available to the FPGA logic.

2.5.2 Streaming interface

Pico Computing's modules rely on a streaming interface for efficient communication between the reconfigurable system and the host processor. Streaming interfaces can be effective at transferring large amounts of data from the host processor to the reconfigurable logic and back to the host processor through separate PCIe communication interfaces. Each interface is capable of achieving a peak bandwidth of about 600MB per second if transferring large block sizes. Transferring data in large block sizes is much more efficient than small block transfers.

3 Algorithm

Mapping in the presence of read errors and accurate scoring are both important aspects of short read mapping, and therefore we implement a version of the indexing solution here instead of the BWT-based solution. To implement this algorithm, we find all seeds for a read, find the locations in the reference where those seeds occur, and perform Smith-Waterman alignment at those locations. In other words, the heart of our mapping algorithm is the Smith-Waterman string matching algorithm. We accelerate the alignment of each read by performing the Smith-Waterman comparison at only a few identified *candidate alignment locations* (CALs) in the reference, and then choose the one that produces the highest alignment score. We find the CALs for a read by looking them up in a large table, which we call the *index table*. An overview of the indexing solution can be seen in Figure 15.

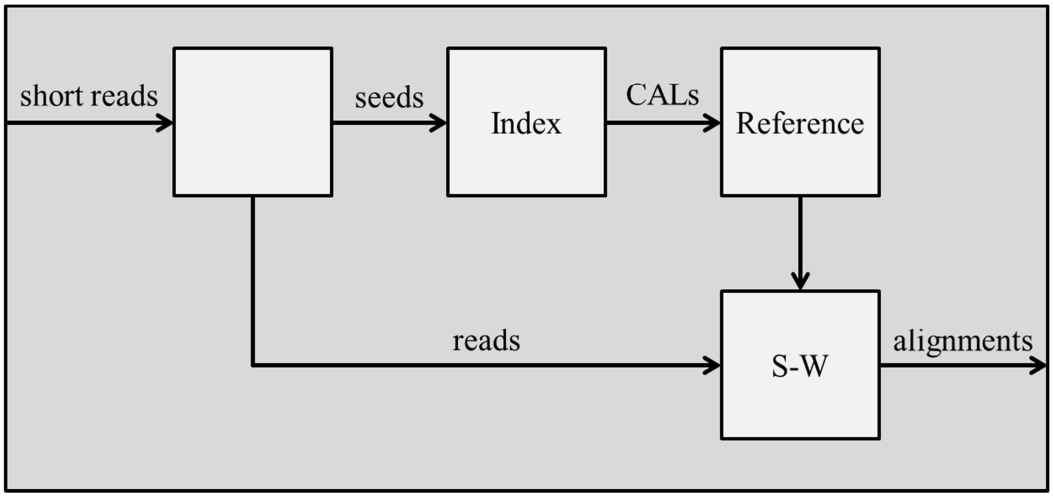

Figure 15: Overview of the indexing solution implemented in a hardware system. One module finds the seeds for a read; one finds the CALs for each seed; one looks up the reference data for the CALs; and a final module performs the Smith-Waterman alignment on the short read and the reference data.

We further accelerate this mapping process by creating a system with many Smith-Waterman units, with which we can simultaneously align many CALs for a read, or many CALs for many reads. By partitioning the reference genome across multiple FPGAs and chaining them in series with all reads flowing through all FPGAs, we are able to pipeline the alignment of reads against different portions of the reference genome. Further details of the algorithm and its expected performance are described in the following sections.

3.1 Smith-Waterman Alignment

The Smith-Waterman string-matching algorithm is commonly used to compare the similarity of two strings. The fully gapped algorithm provides accurate scoring, even in the presence of indels, and its search space remains quadratic, as opposed to the exponential search space of other algorithms such as BWT. The original Smith-Waterman algorithm was designed to perform fully gapped local alignment, which will match any subsequence of one string to any subsequence of a reference string. A slightly different algorithm, called Needleman-Wunsch, was conversely designed to globally align an entire test sequence to an entire reference sequence [22].

Both of these algorithms can be implemented using systolic arrays [23], but neither algorithm accurately performs the alignment of a short read to the reference genome. Specifically, what we want is the full sequence of read base pairs to be aligned to a subsequence of the reference genome. Another slight difference in our goal from the original algorithm is the manner in which long strings of inserted bases are handled. In the original Smith-Waterman algorithm, all insertions or deletions were scored the same. However, long subsequences of base pairs may be inserted or deleted into the reference genome, as occurs in copy number variations (CNVs), and the insertion or deletion penalty will not be identical for every base pair that is inserted or deleted from the reference. To deal with this, a model for sequence scoring, known as the affine gap model, penalizes these long insertions or deletions (indels) less. It applies a high penalty to the first base in a string of insertions or deletions [24], and a lower penalty to the subsequent indels. Figure 16 shows an example of the gap penalty difference between the affine gap model and the original Smith-Waterman algorithm.

Reference	A	C	A	–	–	–	–	C	T
Read	A	C	A	A	A	A	A	C	T
S-W Base Score	+2	+2	+2	-2	-2	-2	-2	+2	+2
Affine Gap Base Score	+2	+2	+2	-2	-1	-1	-1	+2	+2

Figure 16: Affine Gap model scoring for Smith-Waterman alignment reduces the penalty for extending gaps than for opening new gaps.

To modify the Smith-Waterman algorithm to fit our alignment goal of global read to local reference alignment, we merely have to adjust the initial setup of the similarity matrix, as shown in Figure 17. The first step of the Smith-Waterman algorithm sets the first row and column of the similarity matrix to 0. We do set the first row of the similarity matrix to 0, which allows the alignment of the read to begin at any position in the reference sequence. However, we initialize the first column of the matrix, except the cell in row 0, to very negative numbers. This effectively forces the full read to align to the reference. As required by the affine gap model, we maintain two more matrices, with the same dimensions as the similarity matrix. These matrices are responsible for maintaining scores for currently opened insertions (reference gaps) and currently open deletions (read gaps), one matrix for each. A cell in the similarity matrix is dependent upon its predecessors in the similarity matrix, as well as one neighbor in the read gap matrix and one in the reference gap matrix. Therefore, the computation can still be performed using systolic arrays. More information regarding the Smith-Waterman compute units and their hardware can be found in Maria Kim's Master's Thesis [25].

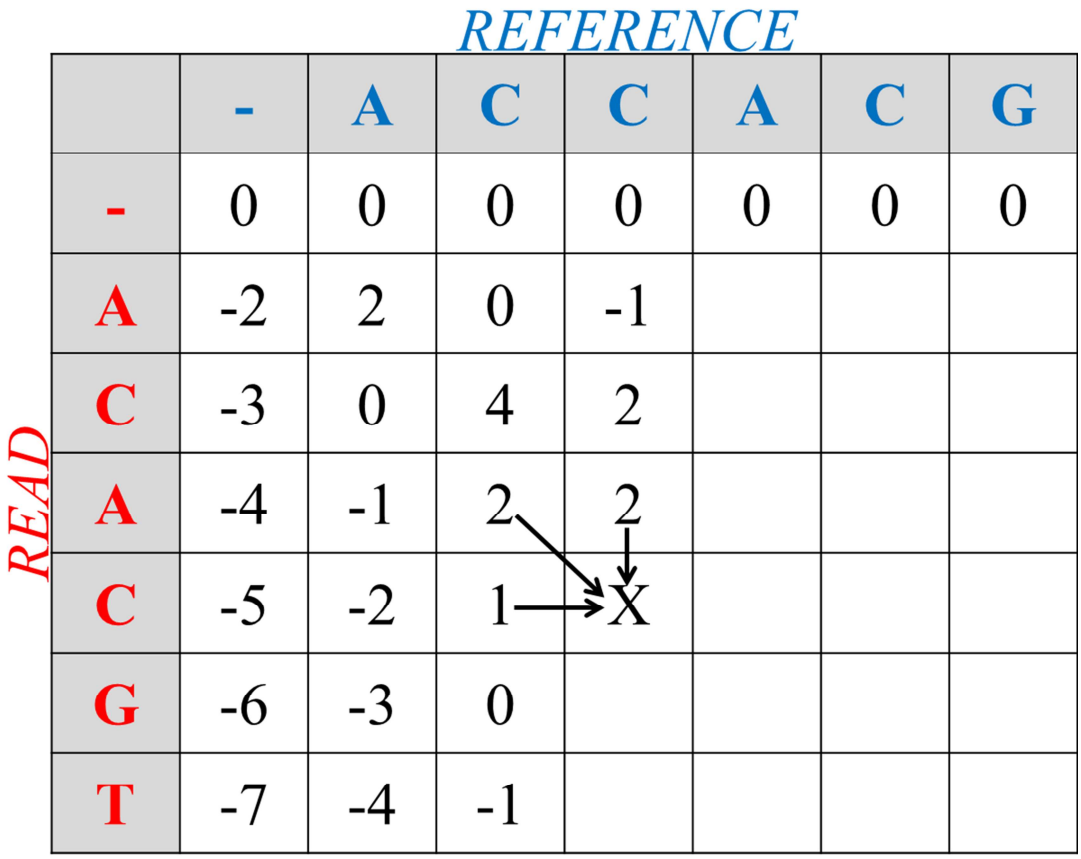

Figure 17: Modified scoring array for newly created Smith-Waterman / Needleman-Wunsch scoring matrix causes the entire read (red) to be aligned to any subsequence of the reference (blue) sequence. The score of a cell is computed as the maximum of its neighboring cells after the addition of their respective indel penalty or match bonus. The diagonal cell gets a bonus if the reference and read base match, and the other two arrows always have an associated indel penalty.

Now that we are able to align a read to a reference sequence of a given length, we can fill up an FPGA with compute units, load them with short reads, and stream the reference to the compute units. Assume we can fit 100 Smith-Waterman compute units operating at 250MHz on each FPGA, we have a system containing four FPGAs, we have a PCIe bandwidth of 600MB per second, and we are mapping 200 million 76 base pair reads. In the described system, the Smith-Waterman alignment becomes the bottleneck, because it requires 500,000 iterations of streaming the full reference and performing 400 Smith-Waterman computations, each of which takes 12 seconds. This results in a total runtime of approximately 70 days. This mapping time is much too long compared to the amount of time required to sequence the reads, which is less than one day, and therefore this is not a feasible solution [1].

If we instead only align each read to a set of candidate alignment locations, i.e. a set of locations to which the read possibly aligns, we can greatly reduce the runtime of the system. For example, assuming that each of the short reads only need to be aligned to four CALs, and assuming all other system parameters remain the same as the previous example, the time required to read the reference data from DRAM and perform the Smith-Waterman alignments for all short reads drops from 70 days to approximately 16 seconds.

3.2 Index table

In order to perform Smith-Waterman alignment on a set of CALs for each read, the CALs must be identified for that read, either in a pre-processing phase or online during the mapping phase. The identification of CALs can be done on the FPGA during the runtime of the algorithm, before the reads are aligned. CALs are found by identifying places where subsequences of a read, called *seeds*, appear in the reference. Each seed, which consists of a given number of consecutive base pairs from within a read (we use 22 base pairs), occurs at zero or more locations in the reference genome.

Thus, we need something that will accept a read as an input and efficiently produce all seeds for that read. Also, we need a structure to map a seed to the set of locations in the reference genome where that seed occurs. This structure, which identifies the CALs for a given seed, is called the *index table*, and its use is shown in Figure 18. We use the index table to identify CALs for a seed, iterate over all seeds within a read, accumulate the CALs, and align the read at each of those accumulated CALs. This appears to be a simple process, but clever organization is required to create an index table that is both space and time efficient.

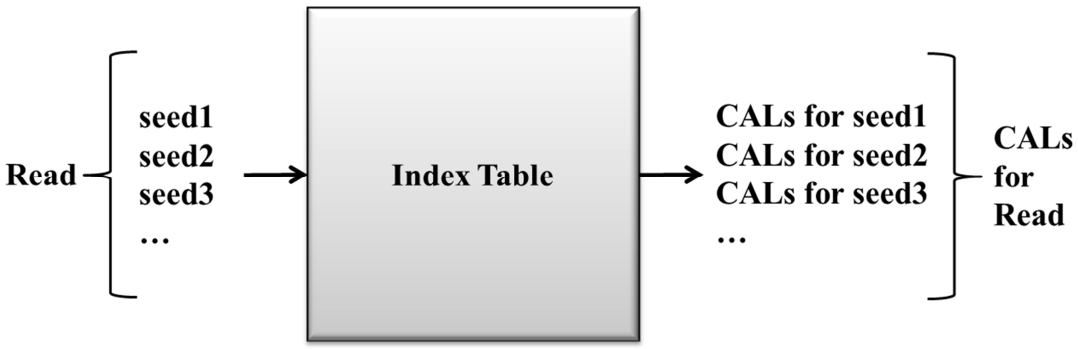

Figure 18: Index table structure should return a set of candidate alignment locations in the reference genome for a given seed.

3.2.1 Index Table Structure

The index table contains one entry for every base position in the reference genome (except for those bases that are 'N' or are near the end of a chromosome), and each entry stores the seed that occurs at that location. In order to find CALs for a seed, we must search this large array to find the complete list of CALs associated with the seed. If the seed length used in the indexing solution is very short, every possible seed will have a set of CALs associated with it, so we can use a direct mapping. Conversely, long seeds will result in many seeds not having any CALs associated with them, thereby resulting in a sparse table. For this sparse table, we use a hash table to map seeds to their set of CALs.

To store all the CALs and seeds from the reference genome, we construct an array of seed-CAL pairs, and sort the array by seed, thereby creating a structure we call the *CAL table*. Given a seed, the user can search the array until the seed is found, and all CALs associated with that seed are in consecutive positions in the array. Unfortunately, each seed will have a variable number of CALs, as seen in Figure 19, so there is no way to know exactly where a given seed lies in the array.

Seed	CAL
AAAA	0
AAAA	1
AAAG	2
AAAT	367
AACA	298
AAGA	3
…	…
TTTC	900
TTTT	899

Figure 19: CAL table contains only those seeds that occur in the reference genome. Some seeds occur multiple times (AAAA), and some do not occur at all (AAAC), so standard addressing is impossible into this array.

Given a seed, our hash table provides an exact pointer into the CAL table, which must then be searched at the index specified by this pointer for the set of CALs associated with the search seed. Since the hash table stores pointers into the CAL table, we call it the *pointer table*. When designing this hash table, we need to determine how to handle collisions and

how to probe the hash table for the target entry. If we do not allow collisions, we probe the pointer table for the target seed, possibly requiring multiple DRAM accesses into the pointer table for a single seed. Also, storing a pointer for all possible seeds occurring in the reference genome could potentially require a very large pointer table. If we instead allow collisions, we can probe for the desired set of CALs with the access into the CAL table, because we can determine how many CALs must be read from the CAL table to guarantee we retrieve all CALs for the target seed. This also guarantees we read the pointer table and the CAL table at most one time each from DRAM.

Many seeds share a single pointer, and the structure in the CAL table pointed to by one of those pointers is called a *CAL table bucket*. CAL table buckets contain a variable number of seed-CAL pairs. To find all CALs for a seed, the seed is hashed, the hashed seed is then used to obtain a pointer from the pointer table, the pointer is used to read all CALs in the specified CAL table bucket, and only those CALs paired with a seed matching the target seed are retrieved for later processing. Figure 20 shows an example pointer table and CAL table using four base seeds, where the pointer from the pointer table points to the start of a CAL table bucket and CAL table buckets may contain different numbers of CALs.

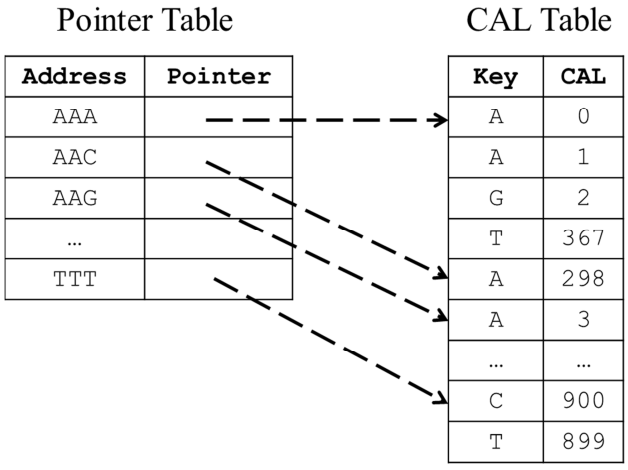

Figure 20: The pointer table uses the address portion of the seed to get the pointer into the CAL table. The key must now be matched against the key from the target seed during processing.

We hash the seed by simply taking the most significant bits of the seed as the hashed seed, which we call the *address* bits, because they are used for the address into the hash table. All seeds in the same CAL table bucket share the same address bits, but they may differ by their

least significant bits, which we call the *key* bits. An example of this breakdown using a 22-base seed, a 30-bit address, and 14-bit key is shown in Figure 21. Since the address bits are already used to obtain the pointer from the pointer table, only key bits must be stored along with CALs, which are used to match against the target seed and differentiate between the multiple seeds that share a CAL table bucket. Storing the key bits instead of the full seed bits in the CAL table reduces the size of the CAL table from 26GB to 16GB, if using 30 bits of the seed for address and 14 bits for key.

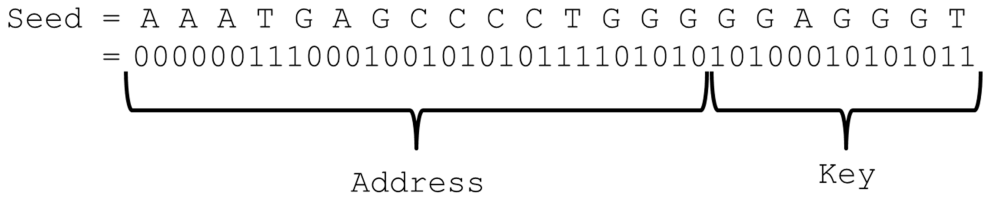

Figure 21: A seed is broken up into address and key portions. Address is used to access the pointer table and keys from the CAL table are matched against the key from the seed.

At this point, we cannot reduce the number of pointers in the pointer table without increasing the average number of CALs per CAL table access, but we can modify the amount of data required for each pointer. Instead of having all pointers in the table be 32 bits, we can store a full 32-bit pointer for every n^{th} pointer, which we call a *start pointer*, and a smaller field for each of the other n-1 pointers, which we call offsets and are relative to the full 32-bit pointer. Similar to how seeds within the same bucket only differ by the key bits, address bits of seeds that hash to the same entry of the pointer table, and therefore share a start pointer, only differ by the least significant few bits, which we now call the *tag*. The definition of address is now reduced to those bits that are common for all seeds sharing a start pointer, as shown in Figure 22.

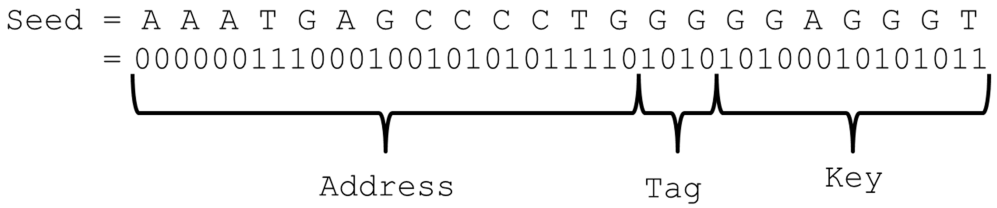

Figure 22: Least significant bits of previous address field now become a 'Tag' field.

With each read from this pointer table, we would like to get the pointer to the CAL table bucket, as well as the number of CALs in that bucket. This can easily be accomplished by slightly changing the definition of the offset fields. Instead of having each field specify the number of CALs before that bucket, each field can specify the number of CALs to the end of the bucket, relative to the start pointer. For example, offset[0] will store the number of CALs in bucket[0]; offset[1] will store the number of CALs in bucket[0] and bucket[1], and so on. An example pointer table can be seen in Figure 23, which was created using four base seeds, four bit addresses, two bit tags, and two bit keys.

Pointer Table

Address	Start Pointer	Off[0]	Off[1]	Off[2]	Off[3]
AA	0	4	5	6	6
AC	6	3	6	8	8
AG	14	2	7	9	13
…	…	0	…	…	…
TT	890	4	…	…	…

Figure 23: Pointer table where the offsets specify the offset to the end of the CAL table bucket. For example, offset[0] specifies how many CALs are in bucket[0].

The pointer into the CAL table is found by adding the start pointer with offset[tag-1], and the number of CALs is found by subtracting offset[tag-1] from offset[tag]. The only exception occurs when the tag bits are zero. When this happens, the pointer is simply the start pointer, and the number of CALs in the CAL table bucket is equal to offset[0]. An example using the pointer table and the CAL table for tag equals two can be found in Figure 24 and Figure 25.

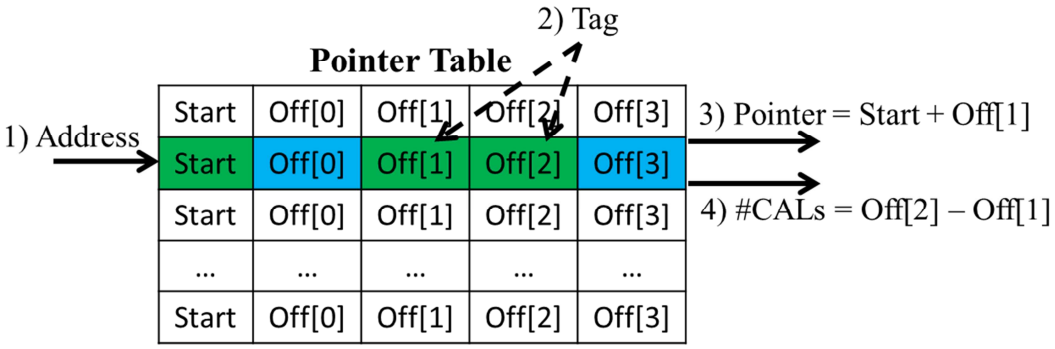

Figure 24: Example of using the pointer table for tag = 2. Pointer is the start pointer of the seed's address + offset[tag-1]. Number of CALs to read from the CAL table is the difference of offset[tag] and offset[tag-1].

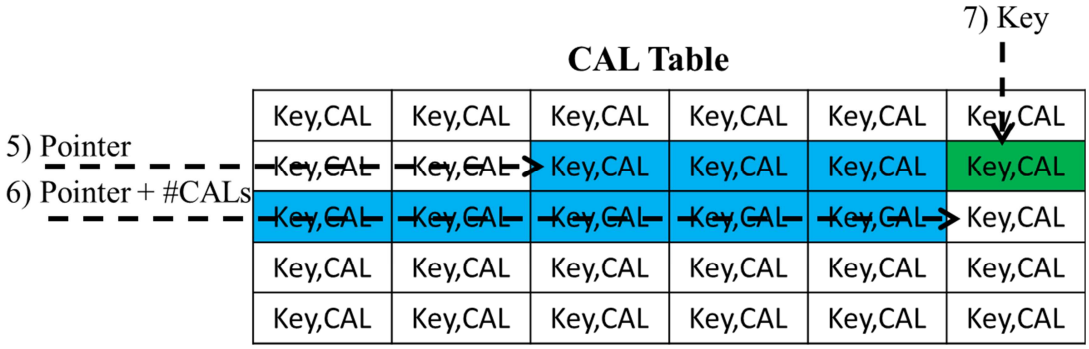

Figure 25: CALs are read from the CAL table (blue) starting at the start pointer and reading up to but not including start pointer + number of CALs. Each of the key read from the CAL table is matched against the target key, and matching keys (green) correspond to CALs that must be processed by Smith-Waterman alignment.

Problems arise when CALs are not uniformly distributed amongst CAL table buckets. As done in BFAST, seeds that appear more than eight times in the reference genome are removed from the CAL table. Also, the size of the offset field must be large enough to offset the largest set of CAL table buckets sharing a common start pointer. Thus, if the number of CALs in each bucket were uniformly distributed over all CAL table buckets, the offset fields in the pointer table, and hence the pointer table itself, would be small. However, seeds that appear in the reference genome are not uniformly distributed throughout the CAL table buckets, as shown in Figure 26.

To deal with this, a hashing function is used to redistribute the CALs amongst CAL table buckets by hashing the seed and inserting the CAL in the bucket specified by the hashed seed. When using the tables to find all CALs for a read, each seed must first be hashed

before it can be broken into the address, tag, and key fields. Our hashing function was developed by Cooper Clauson, and it permutes all bits of the seed, XORs the upper half of bits into the lower half of bits while the upper half of bits remains. These two steps are repeated seven times to achieve a 'good' distribution of CALs into buckets.

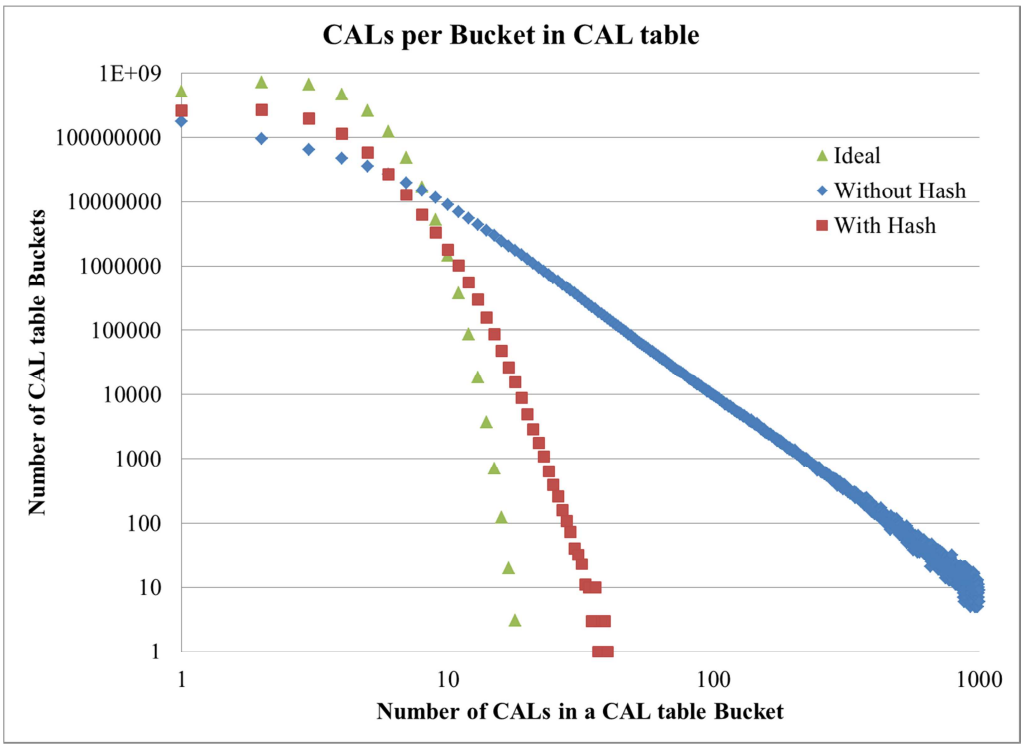

Figure 26: Distribution of CALs per CAL table bucket for the full human genome before hashing (blue), using an ideal hashing function (green), and after hashing the real data (red) plotted against a log scale. The ideal hashing function would distribute the CALs according to a Poisson process, and the real hashing function distributes the original seeds very well over the entire set of CAL table buckets with respect to the ideal values.

The structure described in this section, known as the CALFinder, is responsible for the following: finding all seeds for a read; hashing each seed; looking up the hashed seed in the pointer table; looking up the resulting pointer in the CAL table; and comparing the resulting CALs to the target seed.

3.2.2 Index Table Analysis

Given a few assumptions, we can easily analyze the size and performance of the resulting pointer and CAL tables. Assume for the purpose of analysis that the read length used for short read mapping is 76 base pairs, and we use 22 base pair seeds as found in BFAST.

Each read into the pointer table and the CAL table will return a block of data, but subsequent accesses have no relation to each other, and therefore we will have very random memory access patterns. Each seed of the read must be looked up in the pointer table and the CAL table, and therefore this algorithm requires up to 110 random accesses into memory for each read. After that, the read must be aligned via Smith-Waterman alignment at each CAL found by these accesses into the CAL table, and the reference sequence must be read from memory, which is also random in nature. This brings the number of random memory accesses per read to 110+CALs per read, or more generally, this number is:

Equation 1: Number of random memory accesses per read.

$$accesses\ per\ read = 2 * (read\ length - seed\ length + 1) + CALs\ per\ read$$

The CAL table contains an entry for almost every nucleotide base in the reference genome, which is about three billion. The only bases in the genome that do not have an entry in the CAL table are either near an unknown base in the reference, which is marked as 'N', or are within one seed length from the end of a chromosome. Each of these entries in the CAL table must store the CAL, which is a 32-bit number that specifies the position of the base from the start of the reference genome, and the key, which is used to match against target seeds during the processing of reads. Zero padding is used to align the CAL table entries to a 32-bit boundary, and the resulting 64-bit CAL table entries, assuming keys are less than 32 bits, produce a final CAL table that is approximately 16GB.

The pointer table contains one entry for each possible combination of address bits, and each of these entries contains a 32-bit start pointer along with 2^tag offset fields. Assuming we use 26 bits for address, 4 bits for tag, and 14 bits for each offset, each entry in the pointer table is 256 bits, resulting in a 2GB pointer table. The full reference genome, which is approximately 1GB (assuming a 2-bit per base encoding) must also be stored in memory for this algorithm to function properly. The final aggregate memory footprint for this algorithm with the specified assumptions is about 19GB, which is quite large for most reconfigurable systems.

A concern with the stated algorithm is the number of random memory accesses that must be done for every read. Since we have to access both the pointer table and the CAL table for

every seed, we must randomly access the index about 110 times per read. Thus, we should attempt to reduce the number of memory accesses as much as possible. We can reduce the average number of CALs per CAL table bucket by increasing the number of address and tag bits, as shown in Figure 27. Using 26 bits for address and 4 bits for tag yields an average of 2.8 CALs per CAL table bucket.

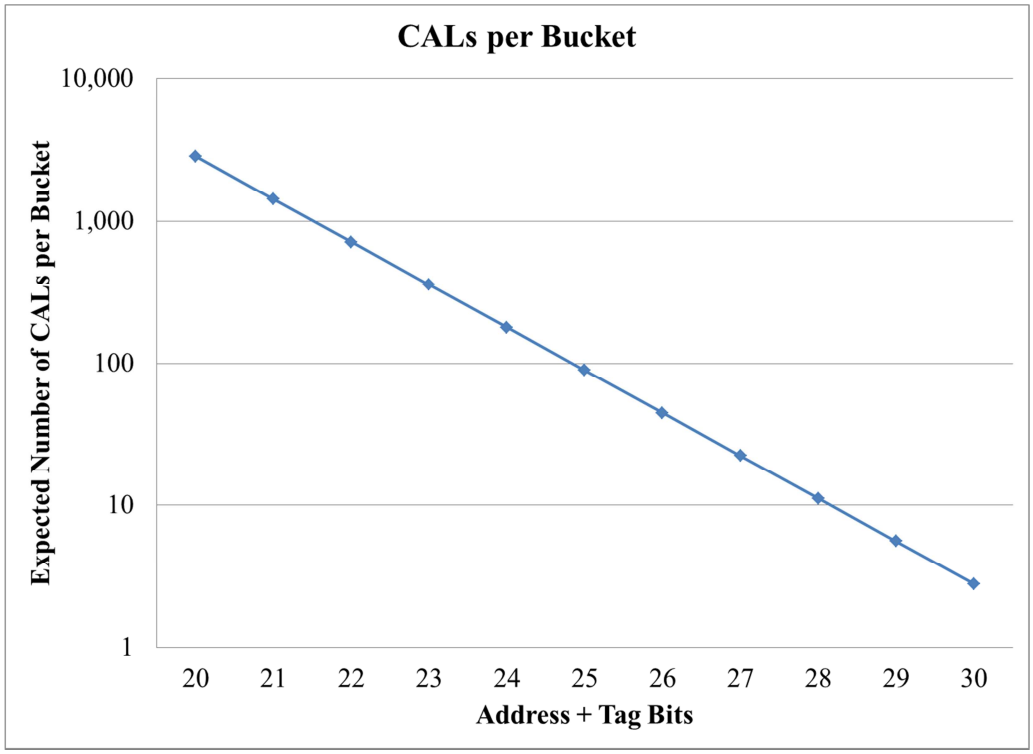

Figure 27: Expected number of CALs per CAL table bucket. Increasing the number of address + tag bits used to access the pointer table by one reduces the number of CALs that must be read for every CAL table bucket by a factor of two. Using 26 bits of address and 4 bits of tag means we use 30 bits of the seed to specify a CAL table bucket, resulting in an average of 2.8 CALs per CAL table bucket.

The total number of address and tag bits used to specify a CAL table bucket sets the size of the CAL table bucket, but the size of the pointer table is determined by the ratio of address bits versus tag bits. Figure 28 shows the minimum sized pointer table versus number of address bits where each series has a constant number of address + tag bits. This graph shows the pointer table sizing if CALs are uniformly distributed through all CAL table buckets. However, they are not uniformly distributed, so we use more bits than the ideal case for each offset in the pointer table, resulting in a 2GB pointer table instead of the ideal 1GB pointer table when using 26 bits for address and 4 bits for tag.

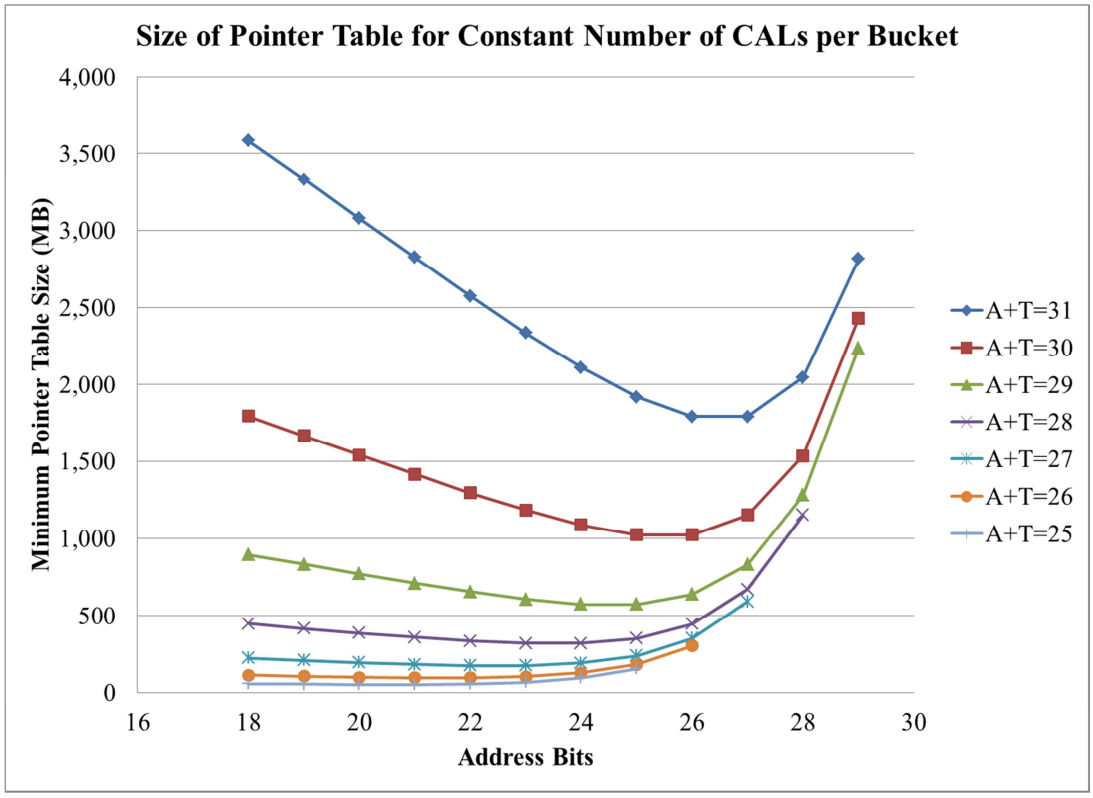

Figure 28: Minimum sizing of the pointer table across various number of address and tag bits. The 'A+T=30' series with A<25 shows that having too many tag bits to obtain a desired number of CALs per access results in an oversized table. Conversely, not enough tag bits also results in a larger than necessary pointer table, as can be seen in the 'A+T=30' series for A>26. To have 2.8 CALs per bucket on average, the pointer table size is in the 'A+T=30' series and results in a minimum pointer table size of about 1GB.

We have specified that we use 26 bits for address and 4 bits for tag, and now Figure 27 and Figure 28 enable us to analyze that decision. We decided to use 30 bits for address and tag so each CAL table bucket would contain a small number of reads, which in this case is slightly less than three. Figure 28 shows that we could have used 29 or even 28 bits for address and tag instead, which would decrease the size of the pointer table but would increase the number of CALs per CAL table bucket. These alternatives produce a pointer table that still must be stored in DRAM, so the decreased pointer table size may not be worth the increased number of CALs per bucket. If we analyze the sizing of address versus tag bits by examining the 'A+T=30' series, we can see that having either 25 address bits with 5 tag bits or 26 address bits with 4 tag bits produces the smallest pointer table for that series.

3.2.3 Index Table Performance

The CALFinder is analogous to an assembly line, with many stages that accomplish a very specific task. With the exception of the memory and bus interfaces, each stage can produce a new result every clock cycle. The stage that finds all seeds for a read must load the read in multiple clock cycles if the read and read ID are greater than the width of the bus interface to the PCIe, but once the read is loaded, it can simply shift the read and select the appropriate bits to produce a new seed every clock cycle. Similarly, the hashing function can produce a new hashed seed every cycle, either by a single stage hashing function or by pipelining the hashing function through multiple stages, if required to meet the system timing requirements. Finally, the module that matches the keys from the CAL table to the target seed uses one clock cycle per CAL being compared. Therefore, if the average CAL table access produces a significant number of CALs, then this module can be the bottleneck of the CALFinder performance. However, we can replicate this matching module as many times as required to match the throughput of the memory system, and it will no longer be the bottleneck of the system.

The remaining stages of the CALFinder are the memory accesses, which are the reads into the pointer table and the CAL table. Both the pointer table and the CAL table must be read for every seed, and each of these accesses are to random portions of the pointer and CAL tables. Each random access into DRAM requires a time penalty in order to open the desired row in memory, which is many clock cycles long. Also, without a memory controller that can pipeline read requests to the memory or distribute read requests to different memory chips, only one module can access the memory at a time. In other words, when the pointer table is being read, the CAL table must remain idle and vice-versa.

Because of this, the total throughput of the CALFinder will likely be equal to the joint throughput of the interface to the pointer table and the CAL table. To improve this bottleneck, the system memory controller should: pipeline read requests to the DRAM, allowing for many read requests to be in flight at a time; permit multiple banks of DRAM to be open at the same time, enabling different portions of the memory to be quickly read; and respond to read requests from a user system in a ‘smart’ fashion by checking if any of the current requests can quickly be handled based on the currently open DRAM bank and row.

These modifications should collectively result in a possible factor of two or three improvement in DRAM random access bandwidth.

3.3 Filter

As described thus far, the CALFinder will find all CALs for all seeds within a read. However, if a read aligns well to a location in the reference genome, many of the seeds should return the same CAL. Also, if a given read aligns perfectly to the reference genome at a location, with the exception of a small number of insertions or deletions, then many seeds will return a CAL pointing to bases in the reference genome that are close to one another. It is possible that deletions will cause the 'real' start of a read, which is the base in the reference that aligns to the first base of the read, to occur earlier in the reference genome than that CAL states. A similar argument can be made for both insertions and also the end of a read. This means we must allow for a given number of indels, on either end of the read, and must at least align the read against a section of the reference genome defined by [CAL – INDELS, CAL + READ_LENGTH + INDELS]. Finally, reference data read from DRAM must be read at a minimum granularity, and the same block of reference must be read for any CAL that lies within that block of data. For these reasons, a set of CALs that point to the same *reference bucket*, or section of the reference that consists of a given number of consecutive bases, should only cause the read to be aligned once for that reference bucket.

3.3.1 Filter Structure

To remove redundant CALs, we can filter the CALs from the CALFinder before they are sent for Smith-Waterman alignment with a module called the *CALFilter*. To filter the CALs, we simply reduce the set of CALs from the CALFinder to a set of CALs that describes unique locales in the genome. To accomplish this, we first subtract an offset to allow for a given number of insertions when aligning the short read to the reference genome, and then we bucketize the CALs so they point to a reference bucket instead of a single base in the genome, as shown in Figure 29. The CALs should be bucketized to the width of a single access to memory, since each read from memory will produce that many bases regardless where the actual desired base falls within that set of bases.

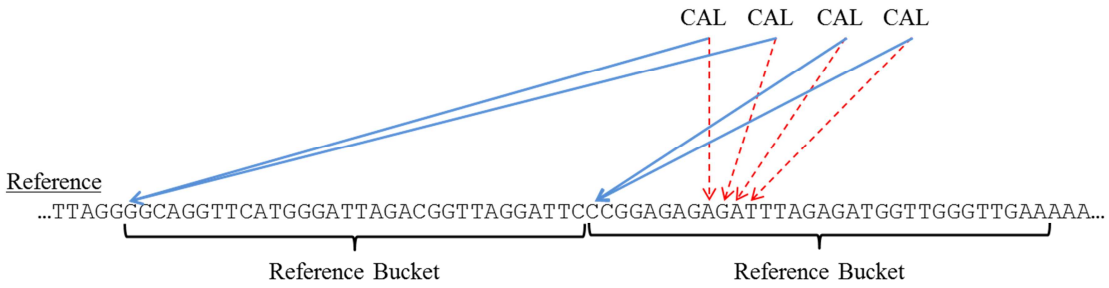

Figure 29: CALs are modified so they no longer point to individual bases in the reference genome (red) but instead point to the first base of their reference bucket (blue). Before bucketizing, we subtract a 10 base offset to allow for insertions in the short read. Because of the indel offset, these four CALs end up in two different reference buckets.

This takes care of the CALs that are within the same locale by changing them all to point to the same reference bucket, thereby making them identical with respect to one another, but we still have to reduce the identical CALs to a unique set. This is accomplished with a simple hash table, which attempts to store the list of CALs that have been sent to the Smith-Waterman alignment phase. We cannot store an entry for every CAL though, because a table with a single bit entry for every possible 64-base reference bucket is 5MB, which is too large to hold in an FPGA memory.

Instead, we map many different CALs to a single entry in the hash table by using the least significant non-zero CAL bits (bucketization causes the least significant bits of all CALs to be zero) as the address into the table. A version number is used as a quick way to clear the hash table. Instead of clearing the hash table for each new read, which takes many clock cycles, we can simply increment the version number. Now each entry in the hash table must store the version number along with the CAL, or at least the portion of the CAL not used as the hash table address.

Figure 30 shows the steps taken in order to filter CALs. An indel offset is first subtracted from the input CAL into this system, which we use 10 bases for our allowed offset, and the CAL is bucketized to a 64-base reference bucket by setting the least significant six CAL bits to zero. For each CAL for a short read, we read one hash table entry, specified by the hashed CAL, and if both the version number and the CAL stored there matches the current version number and CAL respectively, then this CAL is redundant. The version number and CAL are written into the hash table during the same clock cycle as the read, regardless of whether

this CAL is redundant or not. Redundant CALs do not require any further processing and can be discarded, but non-redundant CALs must be sent to a Smith-Waterman compute unit.

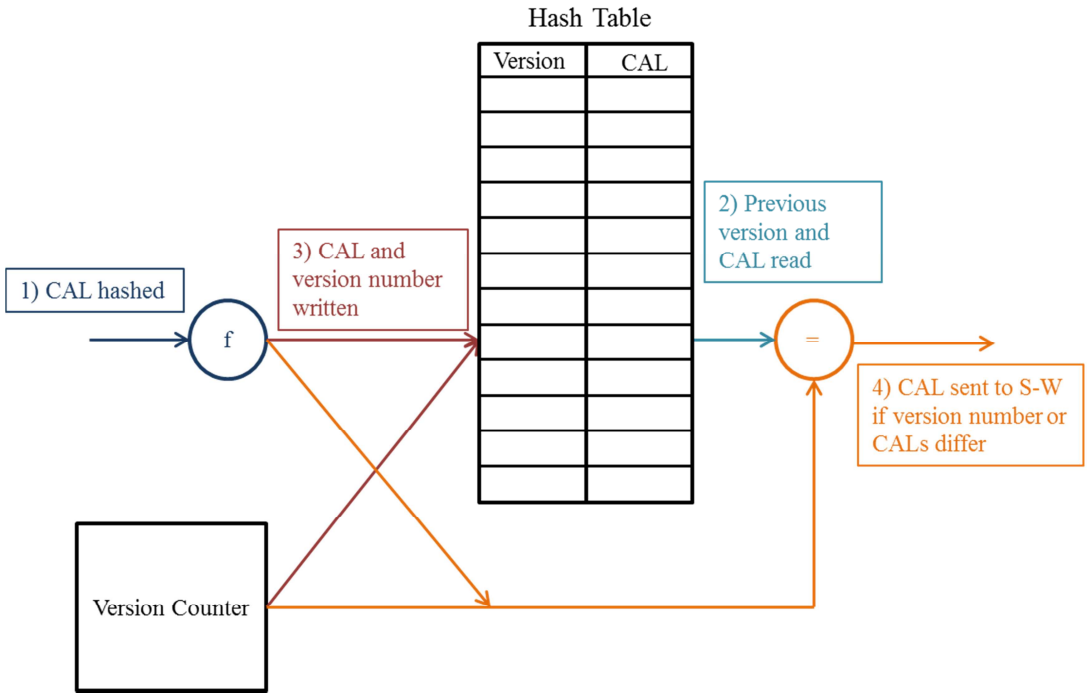

Figure 30: This figure shows the operation of the CALFilter. The version counter is incremented for every new read. Each incoming CAL is first hashed, and the location in the hash table corresponding to the current CAL is read. The version number and hashed CAL are also written into the hash table. If the new CAL-version pair is different than the data that was previously in the table, this CAL is sent for Smith-Waterman alignment.

3.3.2 Filter Analysis

In order to remove the redundant CALs from each read, the CALFilter requires some simple control logic and a block of memory on the FPGA. The control logic consumes very few resources, and the memory on the FPGA is essentially free since it is not needed by other portions of the system.

We have stated that the CALFilter reduces the set of CALs from the CALFinder to a unique set of CALs that are sent for Smith-Waterman alignment, but we must further analyze how this happens to understand what problems can occur. The two types of errors that can occur in this system are false negatives and false positives. *False positives*, which occur when a CAL for a read should be filtered by the CALFilter but instead is sent onto the Aligner, can occur if two CALs for a read hash to the same entry and appear in an alternating sequence,

such as ‘1, 2, 1, 2.’ These CALs will continually overwrite each other in the hash table, and the CALs will be sent for Smith-Waterman alignment more times than is necessary. This is rare since our hash table has 1024 entries, and even in the false positive case, we safely re-check the redundant location with Smith-Waterman alignment, and therefore will not lose any alignment sensitivity.

The more dangerous type of error is the *false negative*, which occurs when a version-CAL pair appears to already exist in the hash table for the current read, but is actually leftover from a previous read. This will result in the read not being aligned at the CAL and leads to inaccurate alignments. False negatives can occur if: the version number rolls over; two reads have the same version number; these two reads share the same CAL; and the location in the hash table has not been overwritten between the matching CALs, as shown in Figure 31. To avoid this situation, we can clear the hash table whenever the version number rolls over; the hash table takes a long time to clear, but it only must be cleared when the version number rolls over. Alternatively, we use a 32-bit version number, which is guaranteed to never roll over unless aligning more than 2^32 reads.

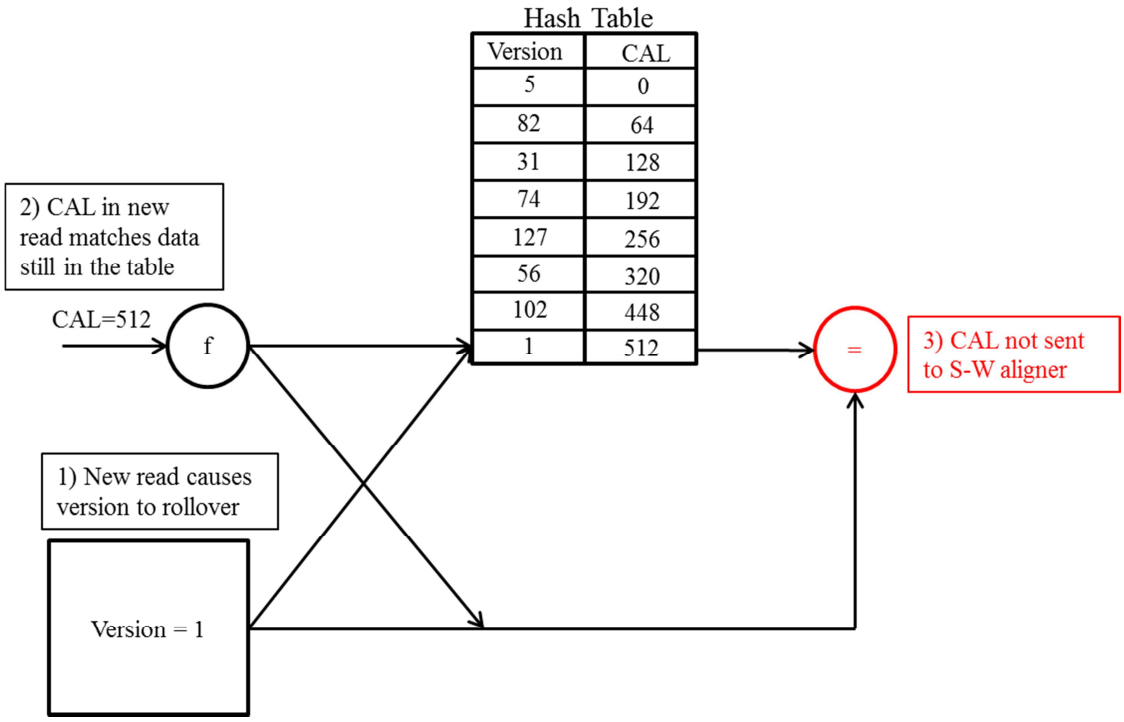

Figure 31: A false negative will occur if the version numbers match for two reads, they contain the same CAL, and the location in the hash table is not overwritten in between the processing of the two CALs.

To reduce the number of false positives for all reads, we can increase the size of the hash table, which halves the number of false positives for every new bit used in the hash. If we instead use another structure, such as a Bloom filter or a small CAM as a victim cache, we can potentially reduce the number of false positives at a faster rate than the factor of two achieved by increasing the size of the hash table. However, if the amount of extra work required by the false positives is negligible or if the Smith-Waterman step is not the bottleneck of the system, the performance gain from the decrease in false positives may not be worth the extra logic and design effort.

3.3.3 Filter Performance

The CALFilter is a simple change to the system, because it can easily be inserted between the CALFinder and the Smith-Waterman alignment unit, but it greatly speeds up the Smith-Waterman alignment step by reducing the total number of CALs to align. Without the CALFilter, each CAL from the CALFinder must be aligned. Therefore, if a read aligns perfectly to a location in the reference genome, the CALFilter will eliminate 54 unnecessary

alignments for 76 base-pair reads and 22 base-pair seeds. The CALFilter is able to read and write the hash table for the same CAL in one cycle, which enables a CAL to be processed every cycle. Since it is able to process one CAL every cycle, the CALFilter will not be the bottleneck of the system, and the only cost of the filter is the hardware resources and added latency for each read. A software prototype mapping 1 million short reads (reads were sequenced from the full human genome) to chromosome 21 reduced the number of aligned CALs from 1.47 million to 481 thousand, which is a 3x reduction in the number of aligned CALs if using the CALFilter.

3.4 Parallelization

Thus far, this algorithm has been described as a sequential process of steps for each read that is being mapped to the reference genome. However, the way to improve throughput of an algorithm on an FPGA is through parallel computation. We want to speed up this algorithm by parallelizing as much of the computation as possible. We can achieve this by partitioning the data set across many compute nodes, as well as replicating the data required for all compute nodes.

This algorithm relies heavily on the ability to rapidly perform numerous Smith-Waterman computations. Multiple Smith-Waterman computations can be performed simultaneously by placing many Smith-Waterman units on an FPGA. A controller then dispatches short reads and reference data to each of the units. The controller is responsible for: knowing when compute units are ready to accept more data; looking up the reference genome data in the memory; sending the read and the reference data to a Smith-Waterman unit; and collecting all results for a read before reporting the score and alignment to the system. By packing as many Smith-Waterman units onto an FPGA as possible, and by having a system with multiple FPGAs, we can significantly improve the runtime of a system that is limited by its computation throughput.

Our system is limited by the memory bandwidth due to the large number of random accesses into the pointer table and CAL table for each read. To speed up accesses to the memory, we can include multiple memories in the system, each with its own memory controller and memory bandwidth. We can then partition the required data across the memories, and assuming we have the same overall number of memory references, but they are now

uniformly distributed to all memories, we can gain an increase in bandwidth proportional to the number of memories. If we need to split the memories across multiple FPGAs, bandwidth to non-local memories may be reduced due to the limited communication bandwidth between the FPGAs. Therefore, if splitting across multiple FPGAs, it may be necessary to replicate some portion of the data in all memories.

Each M503 board in the target system contains approximately 8GB of DRAM, which means that we need at least three boards to contain the 16GB CAL table, the 2GB pointer table, and the 1GB reference data. If we partition the index by cutting the reference genome into four contiguous sections, we can create a CAL table and hash table for each section. The hash table remains the same size, which is 2GB, because it must still contain all the pointers from the original table. This can be modified later by reducing the number of bits used for address and tag; however this is at the cost of an increased number of CALs from the CAL table for every seed. The CAL table for each partition shrinks to one quarter the size of the original CAL table, and the same situation happens to the reference data. Each FPGA is responsible for aligning each read to one quarter of the reference genome; in doing so, we attempt to increase the memory bandwidth by a factor of four.

However, if we partition the index across the boards by location in the reference genome, we still have to perform 110 memory accesses per seed on each board (55 into the pointer table and 55 into the CAL table), thereby negating much of the improvement in memory bandwidth. If we instead partition the index across the boards by one base of the seed, we only need to access the pointer table if this base of the current seed matches the partition's base. This reduces the number of index accesses per seed by a factor of four. However, each board may find the same CAL for a read as a previous partition, and therefore be forced to perform Smith-Waterman alignment at this CAL, even if the previous board already aligned the short read at this CAL. These extra Smith-Waterman alignments cause unnecessary DRAM accesses to read the reference data, but redundant CALs from previous boards in the system can be avoided by pre-seeding the filter with the CALs found for a read by previous partitions. This may be a significant tradeoff in order to receive a factor of four reduction in the number of memory accesses per seed. A system such as the one shown in Figure 32

could map reads to the full human genome, with each board mapping to one of the four partitions, and the results from one board being sent to the following board.

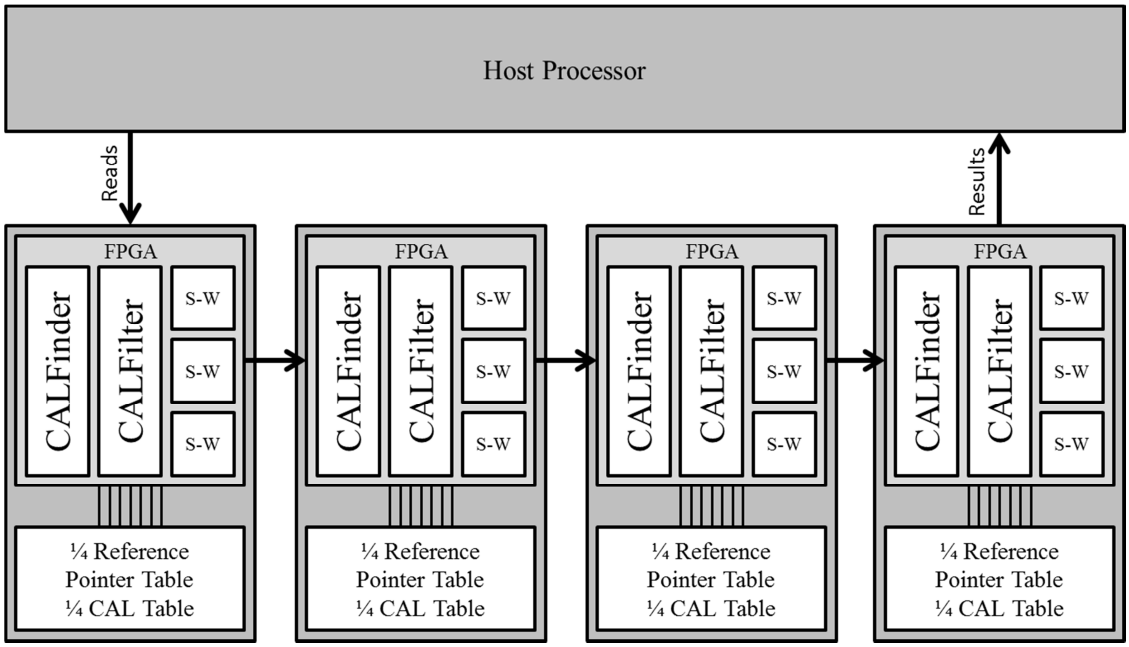

Figure 32: The host dispatches reads to one of the processing FPGAs, and as the reads are passed along, they are aligned to a different partition of the reference genome. The memory connected to each FPGA contains the CAL table for that FPGA's partition of the reference genome, the reference data for the partition of the genome, and a full pointer table, with each pointer table only pointing into the CAL table for its associated partition.

More FPGAs can be added to the system, preferably in multiples of four with each set of four mapping short reads to the full human genome, and the set of short reads to be mapped can be partitioned amongst the sets of FPGAs. In other words, if we have two sets of four FPGAs that each can align to the full genome, we can dispatch half of the reads to the first set for alignment, and the other half of the reads can be aligned by the second set of FPGAs, thereby increasing the throughput of the system by a factor of two. This throughput improvement could continue in this manner until the PCIe communication link can no longer send reads fast enough to all sets of FPGAs.

4 Alternatives

Many algorithms are sequential in nature and do not contain enough parallelism to be successfully accelerated with dedicated hardware. Before implementing this algorithm in

hardware, we first considered many alternative implementations; however, it was necessary to evaluate their parallelism and potential speedup if executed on a hardware system. Some of the alternative alignment methods that we considered for implementation were the Burrows-Wheeler based solution, as described in previous sections, a seed and extend method, which is similar to the BLAST alignment software [26], and some subtle changes to our proposed algorithm, such as storing read data on the FPGA and iteratively streaming reference data to the system.

4.1 Proposed Algorithm

The algorithm discussed in section 3 was specifically developed with the concepts of a hardware system in mind. In this algorithm, we recognized the data parallelism implicit in the alignment of a read to a CAL, the alignment of reads to different CALs, and the alignment of reads to different partitions of the genome. These elements each lend themselves to parallel computation, and therefore a large performance increase compared to sequential execution.

Cells on the anti-diagonals of the Smith-Waterman alignment matrix are independent, and therefore can be computed in parallel by using multiple copies of a cell connected together to form a systolic array. Reads are assumed to be taken from one location in the reference genome, and therefore only one alignment should be optimal; however, the scoring of the alignment for a read at each CAL is independent, and therefore the read can be aligned at all CALs in parallel by using many Smith-Waterman compute units. Lastly, reads can be aligned against multiple partitions of the genome in parallel, but the scores should be aggregated at the end, and the location with the highest score is declared the final alignment. This parallelism allows numerous compute units to be executing simultaneously, which is what hardware is able to do very efficiently, thereby reducing the alignment time compared to a processor executing sequential code.

4.2 Burrows-Wheeler Transform

The Burrows-Wheeler transform-based solutions enable the system to very rapidly search an index to find all locations of a substring. However, mismatches in the read cause the search space for the BWT based solutions to rapidly expand, because a single error could be located at any of the bases of the current suffix. This means that the algorithm must search all

possible errors at all bases of the suffix, which leads to an unbounded search space. To reduce this search space, software implementing the BWT based solution limits the number of errors allowed per short read. This results in not all short reads being aligned, which can affect the quality of the overall mapping.

This algorithm can potentially rely on the independence of read alignments in order to simultaneously align a large number of reads. However, each of these compute units must be able to perform the full Burrows-Wheeler algorithm, which requires accessing an index stored in a memory that must be shared by all compute units. Effective sharing of that memory can be difficult, because each unit will require continuous accesses to random portions of the memory, dependent upon the read it is aligning. More importantly, backtracking caused by read errors will require a complex state machine to search the entire read for possible alignments. These two problems make the Burrows-Wheeler transform-based solution difficult and costly to achieve a large speedup in a hardware system.

4.3 Seed and Extend

The alignment technique used by BLAST, known as seed and extend, begins with a process very similar to our suggested algorithm. This technique uses short subsequences of a read to identify possible locations within the genome for further processing, which is called the seeding step. At each of those possible locations, the algorithm attempts to grow the subsequence of the short read that matches the reference genome by iteratively adding bases from the short read to the initial seed. The seed is extended in a direction until the score has fallen a specified distance below the prior maximal alignment score for a shorter sequence.

This provides for very rapid identification of possible matching locations for a read, and it also aligns the read quickly at each of those locations. However, a single indel in a read can cause the alignment score to be much worse than if the same read were aligned at the same location using a fully gapped alignment, such as Smith-Waterman. Therefore, the seed and extend method could produce extremely rapid results by implementing several compute units on a single FPGA; however, they would all need to effectively share the index, which maps seeds to locations in the reference genome, as well as the reference data. Also, inaccuracies in the scoring system require a fully gapped alignment to be performed.

4.4 Reference Streaming

A simple change to the proposed algorithm from section 3 would be to first compute locations that each short read must be checked against the reference genome. This can be accomplished by using a table to map seeds to locations in the genome and then inverting that information into a new table that maps locations in the reference genome to reads that should be aligned at that location, which we call a *Read Position Table* (RPT). Step two of this algorithm would be to stream the reference genome to the system, continually reading the RPT to find all reads to be aligned at each position along the streaming reference. Each of the reads from the RPT would be loaded into a Smith-Waterman compute unit and aligned during the appropriate section of the reference as the reference is streamed through an entire pool of compute units. More information about this algorithm can be found in Ebeling's technical report [27].

Due to a finite amount of Smith-Waterman units fitting on an FPGA, only a limited number of reads can possibly be aligned during one pass of the reference streaming. Therefore, the total set of reads will have to be aligned by streaming the reference multiple times through the system. This reference streaming algorithm will require very fast access to both the RPT, in order to find which reads must be loaded into the Smith-Waterman units for a section of the reference, and to the reads data, which must be transferred from a large memory to the individual Smith-Waterman units. Also, this algorithm will require a much larger memory footprint, because the set of reads, which can be up to 30 times larger than the reference, must now be stored in DRAM instead of the reference genome.

5 System Design

The full genome mapping solution will be implemented on a system consisting of M503 boards, but a prototype system able to align 76 base pair reads to chromosome 21 was implemented on a single M501 board. Chromosome 21, being one of the smallest chromosomes in the human genome, was chosen for the prototype system because its index, which consists of the reference, pointer table, and CAL table, fits in the 512MB of available DRAM on the M501 board.

5.1 M501 Board

Pico Computing has provided a framework system for instantiating user designs and interfacing with both the memory and host processor. The M501 framework is a 128-bit bus system, which instantiates a single memory controller responsible for all accesses to the DRAM. The memory interface has multiple ports for access to the DRAM; each port contains its own read and write ports, which allows for multiple user modules to access the memory in one design. The memory controller interfaces with the *Memory Interface Generator* (MIG), which is a Xilinx specific memory controller created by the Xilinx *Integrated Software Environment* (ISE).

The M501 uses a streaming model for data processing and communication with the host processor. The framework instantiates two streaming *channels* for streaming data to and from the DRAM by simply instantiating two large FIFOs, one for each channel. The user can also create their own streaming channels by instantiating streaming FIFOs in the logic. Each channel is simply identified by a range of addresses that all correspond to the channel. Output channels dump data from the FIFO back to the host when a read is asserted on that channel. Input channels accept data from the input stream when the data is written to that channel. Two PCIe communication interfaces between the host and the FPGA system, one for transferring data to the FPGA and one for reading data from the FPGA, are each able to support a continuous 600MB/s data transfer rate. This logic is all instantiated on the Xilinx LX240T, which is a medium sized FPGA containing about 240,000 logic cells.

5.2 Host Software

The overall algorithm, which is shown in Figure 33, executes as a stream that begins and ends in the host's main memory. The host software first loads the FPGA's DRAM with the precompiled tables from hard disk, which is done by streaming a binary file to a FIFO attached to the write port of one of the ports on the memory controller. Once the tables have been loaded into memory, the reads can be aligned with a single stream through the FPGA system. At the input of the stream, a thread on the host processor load reads from a file on hard disk and sends them to the FPGA system via PCIe transfer. On the output end of the stream, a different thread on the host processor reads alignments from the FPGA system via PCIe, and alignment results are written into main memory. To efficiently use the PCIe

communication, data must be transferred in large blocks, which must be approximately 16KB per packet or greater.

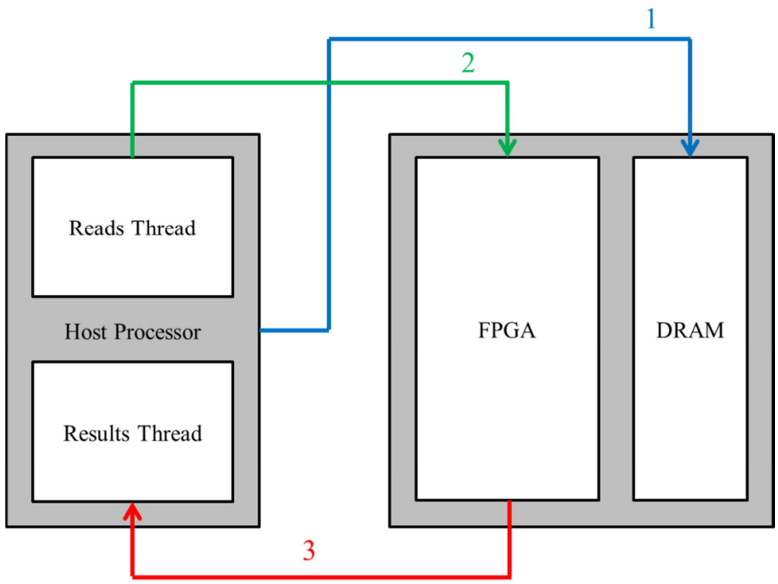

Figure 33: Steps for host software for the short read mapping system. The host loads the DRAM with the pointer table, CAL table, and reference data. After the tables are loaded, one thread on the host processor continuously streams reads from a file to the FPGA, and another thread streams results from the FPGA to a file.

5.2.1 Index

A precompiled index table must be loaded into the FPGA's DRAM before the short read mapping algorithm can be run. This index table consists of the reference data for the current portion of the reference genome to which reads are being mapped, the CAL table, and the pointer table. Of the three pieces of data, the CAL table is largest, because it must store more than 32 bits for almost all positions in the reference. The reference data is the smallest, because each base in the reference can be encoded using two bits, assuming the 'N' is treated as an 'A' in the reference. The bit encodings used in the system are shown in Figure 34.

Base	Encoding while Generating Seeds	Encoding while Processing Reads
A	00	00
C	01	01
G	10	10
T	11	11
N	invalid seed	00

Figure 34: Bases are encoded using two bits per base. Unknown bases, encoded as 'N' in the reference and the reads files, are interpreted as an 'A' during reads processing, but cause seeds containing one or more 'N' to not be added to the CAL table.

The CAL and pointer tables are precompiled by walking along the reference genome. At every position in the reference, the 22-base seed is noted, along with the current position in the reference, to create a large array of seed-CAL pairs. Any seed containing an 'N' base is not added to the array. The array is then sorted by seed, and to improve system performance, seeds appearing more than 8 times in the reference are removed, as done in the BFAST software short read mapping tool [3]. The start pointer for each entry of the pointer table is found by noting the index in the CAL array of the first seed for a set of address bits. The offsets associated with one start pointer are listed in the pointer table for every combination of tag bits. As the CAL table is traversed, the offsets should list the number of CALs from the index of the start pointer to the end of a CAL table bucket, where each of the CALs in a bucket share the same set of address and tag bits; however, they may differ by their key bits. In other words, the index of a seed that is the first to contain a given set of address bits should be noted as a full pointer into the CAL table, and the index of a seed that is the first to contain the next set of tag bits should be noted as an offset from that start pointer. Once all the pointers have been found, the address and tag bits of the seeds in the large seed-CAL array can be removed, leaving behind key-CAL pairs that make up the CAL table.

The CAL table, pointer table, and reference data must all be packed into a memory designed for 256-bit aligned accesses. While mapping reads to chromosome 21, we decided to break the seeds into 22 bits for address, 4 bits for tag, and 18 bits for key. This created a pointer table with 4M entries, where each entry contains a 32-bit start pointer and sixteen 14-bit offsets, resulting in a 128MB pointer table. Each entry is therefore 256 bits, and can be read with a single aligned read from DRAM, which is performed with a 2x128-bit transfer from

the multi-ported memory controller. Each entry of the CAL table stores an 18-bit key and a 32-bit CAL. For simplicity, four key-CAL pairs were packed into 256-bit boundaries, where each key-CAL pair is aligned to a 64-bit boundary, leaving 14 bits of padding for every entry in the CAL table. This packing scheme results in a 256MB CAL table, with 22% wasted in padding. After packing the reference two bits per base, the resulting reference consumes 12MB of memory. The resulting index was packed into the 512MB of available DRAM as shown in Figure 35.

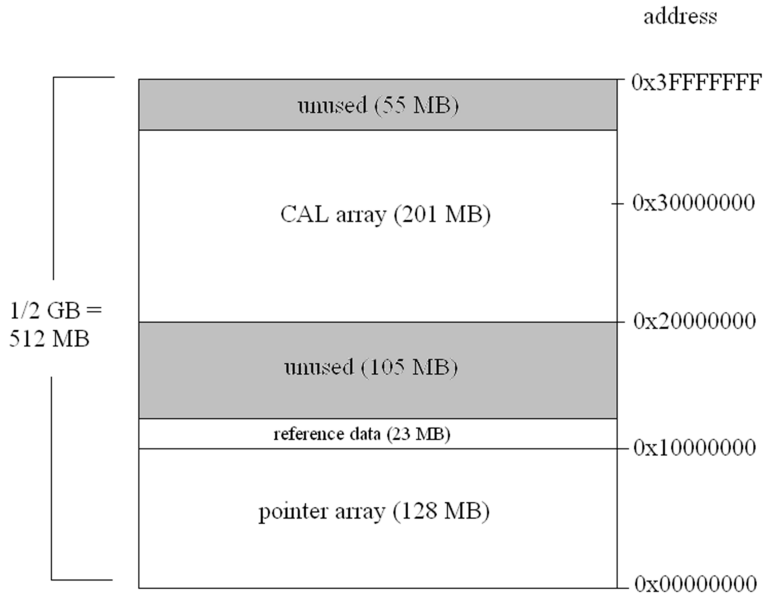

Figure 35: M501 512MB memory organization for chromosome 21 index. [28]

5.2.2 Sending Reads

One thread on the host processor is responsible for sending reads to the FPGA system. The thread reads blocks of data from a precompiled binary file from the hard disk into memory, and drivers provided by the Pico infrastructure transmit these blocks of data to the input stream FIFO via PCIe. The original FASTQ file, which is a file format containing the reads to be mapped, is pre-processed so each read and a unique ID for each read are stored in 256 bits of data. The data for 76 base pair read is stored in the least significant 152 bits, and the ID is stored in the most significant 32 bits of read data, as seen in Figure 36.

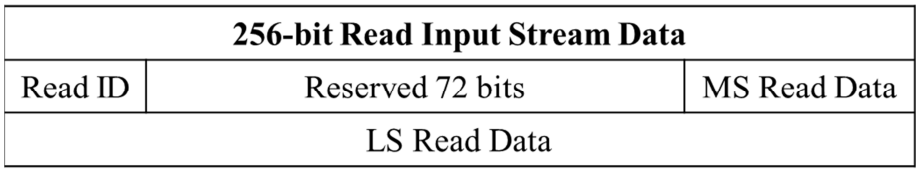

Figure 36: Reads are transmitted to the FPGA in two 128-bit transfers from the streaming FIFO, where the least significant bits of read data are the first data transferred to the system, followed by the read ID and the most significant read bits.

5.2.3 Reading Alignments

Another thread on the host processor is responsible for reading alignments from the FPGA system, once again via the PCIe link. This thread reads blocks of results, where each result is 128 bits of data consisting of the unique ID that was sent in with the read, the final alignment location, and the score of the alignment. The ID is once again packed into the most significant 32 bits, the alignment in the least significant 32 bits, and the two's complement score in the next 9 bits, as shown in Figure 37.

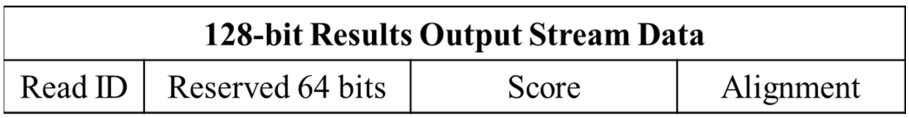

Figure 37: Result data streamed from the FPGA to the host can be packed into this 128-bit structure seen here.

5.3 Hardware

The hardware system is responsible for taking reads from the input stream and producing alignments with scores at the output stream. The modules that make up the hardware system for the mapping algorithm can be broken up into three distinct phases. The first phase finds all CALs for each read, and the second phase filters the CALs to a set of unique CALs. The final phase aligns the read at each of the CALs, passing the alignment with the highest score to the output. The *CALFinder*, which is the phase that finds all CALs for the reads, and the *Aligner*, which is the phase that actually aligns the reads to the CALs, both need to access the DRAM, which is done through the RamMultiHead module. The CALFinder uses two of the read ports from the RamMultiHead module, one to read from the pointer table and one to read from the CAL table. The Aligner uses one other read port to read reference data, and one other port is available to the host processor for streaming data to or from the DRAM.

Figure 38 shows this view of the system implementation along with the memory interfaces on the M501 board.

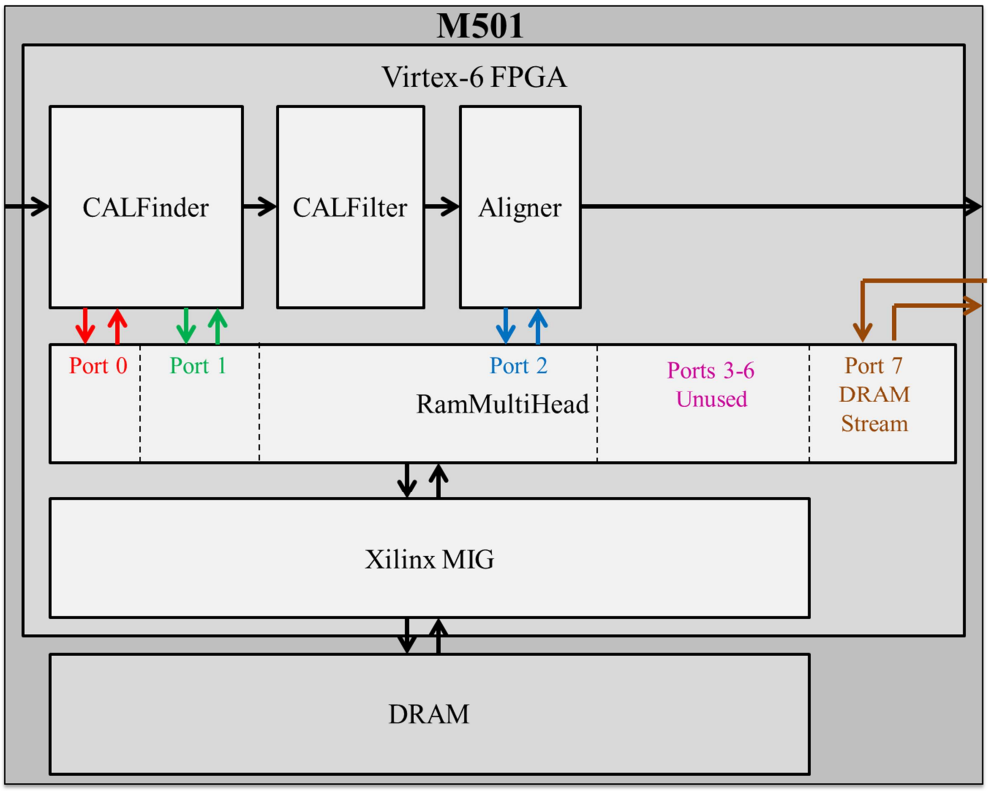

Figure 38: Three of the eight read ports of the RamMultiHead module are used by the system to access the pointer table, CAL table, and reference data. One read and one write port of the RamMultiHead module is used to stream data to and from the DRAM.

5.3.1 CALFinder

The CALFinder module is further divided into three sub-modules that collectively perform the required tasks, as shown in Figure 39. The first module, known as the *seedFinder*, produces all seeds for a given read. The *hashLookup* module, which is second in the CALFinder pipeline, hashes each seed and looks up the hashed seed in the pointer table. The last module, known as the *indexLookup* module, uses a pointer from the hashLookup module as an index into the CAL table and outputs the CALs associated with the matching key bits from the seed. FIFOs are used as the data passing mechanism between the modules.

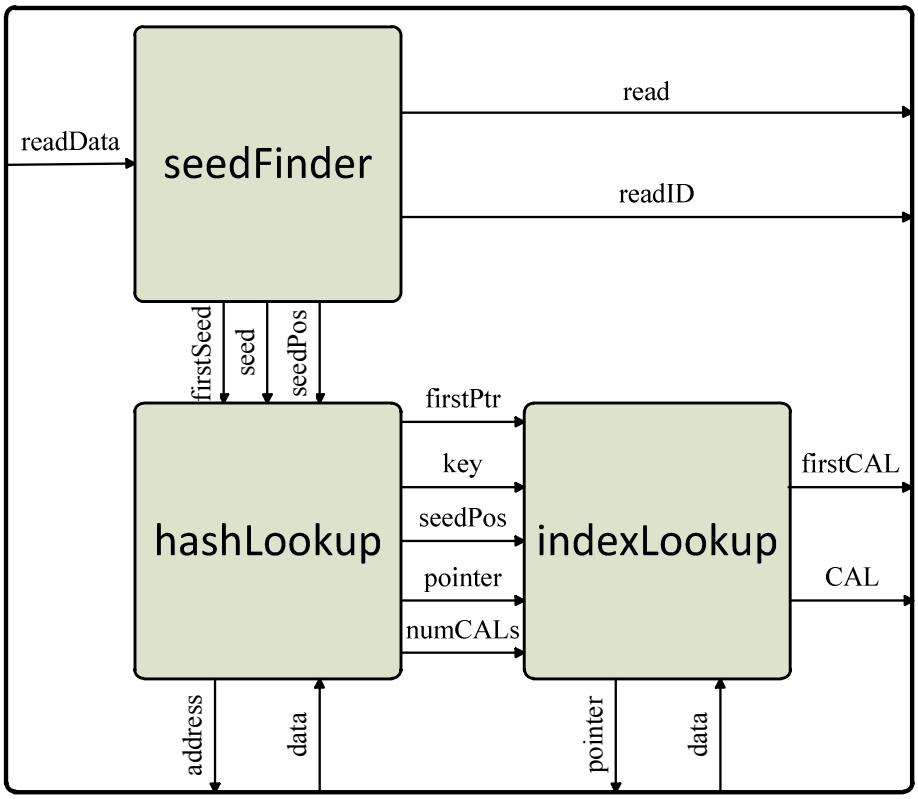

Figure 39: Block diagram of the CALFinder showing the seedFinder producing seeds to the hashLookup module and reads to the output. The hashLookup module reads from DRAM, passes a pointer to the indexLookup module, which uses the pointer to access the CALs and pass them to the output.

5.3.1.1 SeedFinder

The seedFinder module consists of a register to store the read ID, a parallel load shift register, a counter, and some control logic, which can all be seen in Figure 40. Since the read length is 76 bases, or 152 bits, the read must be transferred from the input FIFO to the seedFinder's shift register in two 128-bit transactions. The first transfer loads the least significant 128 bits of the read into the shift register. The second transfer loads the remaining most significant bits of the read into the shift register, and it also loads the 32 bits of the read ID into the read ID register. The counter, which is used to track the position of the current seed from the start of the read, gets reset when the read is loaded into the shift register.

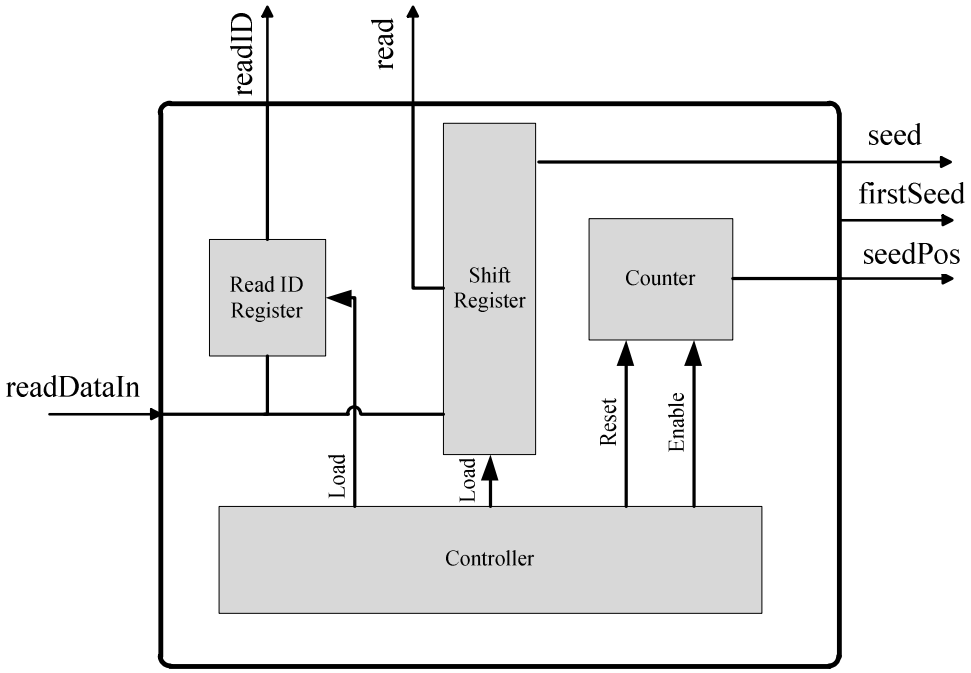

Figure 40: Block diagram of seedFinder module showing the read ID register, shift register, seed position counter, and control logic.

After the read and read ID are loaded from the input stream, the read and ID are written to the read FIFO. The current seed is simply the most significant 44 bits of the shift register, and the seed is updated by shifting the read two bits to the left and incrementing the seed position counter. A firstSeed signal is asserted for the first seed of every read and will be used later to update the read and read ID from the read FIFO.

5.3.1.2 HashLookup

The hashLookup module, as shown in Figure 41, accepts seeds from the seedFinder module and outputs to the indexLookup module a pointer into the CAL table, as well as the number of CALs to read from the CAL table. Before using the seed as an address into the pointer table, it is first hashed by the *hashFunction* module to randomly spread the seeds over all possible CAL table buckets. The hashing function used here consists of seven stages, where each stage contains a random permutation of all bits, followed by an XOR of the upper half and the lower half of bits. Details of the hashing function can be found in the Appendix. An asynchronous FIFO is used to cross a clock boundary into the DRAM's clock domain, and the hashed seed is then used by the *hashTableInterface* module to read one entry from the pointer table, as described in section 3.2.1.

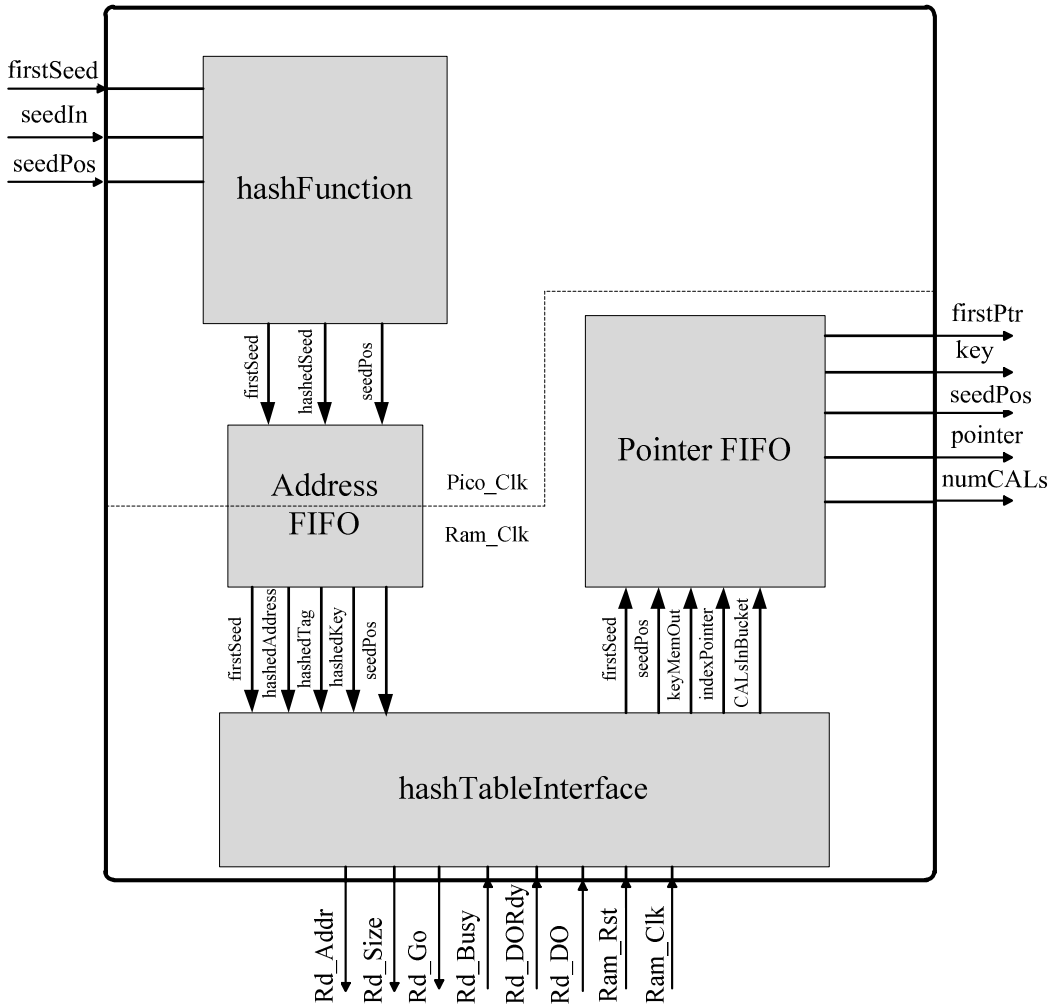

Figure 41: Diagram of the hashLookup module shows the hashFunction, which hashes incoming seeds to randomly distribute seeds among all CAL table buckets, the asynchronous FIFO used to cross into the DRAM's clock domain, the hashTableInterface module that accesses the pointer table, and the pointerFIFO that stores pointers before they are used to access the CAL table.

5.3.1.3 IndexLookup

The indexLookup module, as shown in Figure 42, comprises a memory interface to read from the CAL table, an asynchronous FIFO to cross back into the user logic clock domain, and a module to filter out those CALs associated with keys that do not match the target key. The first module, called the *CALtableInterface*, accepts a pointer and a number of CALs to read from the hashLookup module. The DRAM must be read aligned to a 256-bit boundary, so this module first computes the starting address of the read in DRAM, which denotes which 256-bit block of memory must be read first, and the size of the read, which tells how many 256-bit blocks of memory must be read to get all CALs for this pointer. Recall that each

key-CAL pair is packed into 64 bits of DRAM, so 4 key-CAL pairs are retrieved with each block that is read from memory. The CALtableInterface does not begin the read until there is enough space in the asynchronous FIFO at the output, so a data count is passed back from the FIFO to notify how many entries the FIFO currently holds. This interface outputs a dummy firstCAL for the first CAL of every read, which is used later on as a signal to update the read FIFO.

In the user clock domain, a module called the *keyFinder* parses through the key-CAL pairs that were read from DRAM. The key from the current seed is passed along through the asynchronous FIFO, and it is utilized as the target seed by the keyFinder. CALs associated with keys that match this target seed are outputted to the CALFilter, and they will eventually be aligned by the Aligner. Since we must read the DRAM aligned to 256-bit boundaries, accesses into the CAL table may return key-CAL pairs that are not in the CAL table bucket of the target seed. These key-CAL pairs that get read from DRAM but belong to other CAL table buckets are called *non-bucket CALs*. This will cause extra work for the keyFinder, which takes one clock cycle to process each key-CAL pair from the CAL table, resulting in a worst case of six extra cycles for the keyFinder to process these non-bucket CALs per CAL table access. No more than six non-bucket CALs will ever be read from the CAL table when reading at a four CAL granularity (because DRAM needs to read in 256-bit granularity and CAL table entries are 64 bits), because if so, a full step of non-bucket CALs must have been read from DRAM. Reads to the CAL table in DRAM are random access and should take many more than six cycles per access. Therefore, these non-bucket CALs create unnecessary work for the keyFinder, but they should not increase system runtime.

However, if the seeds associated with these non-bucket CALs match the target key, they will be interpreted as CALs that should be aligned for the current read. The probability of any two given keys matching is the inverse of the total possible keys, or 0.0061%. Assuming the number of non-bucket CALs returned during a given access to the CAL table is uniformly distributed between zero and six, the expected number of non-bucket CALs that are aligned while mapping 200 million reads is approximately two million CALs, which is 0.25% of the full alignment time if we average four CALs per read. Therefore, we simplify the reading from the CAL table at a very small system runtime cost.

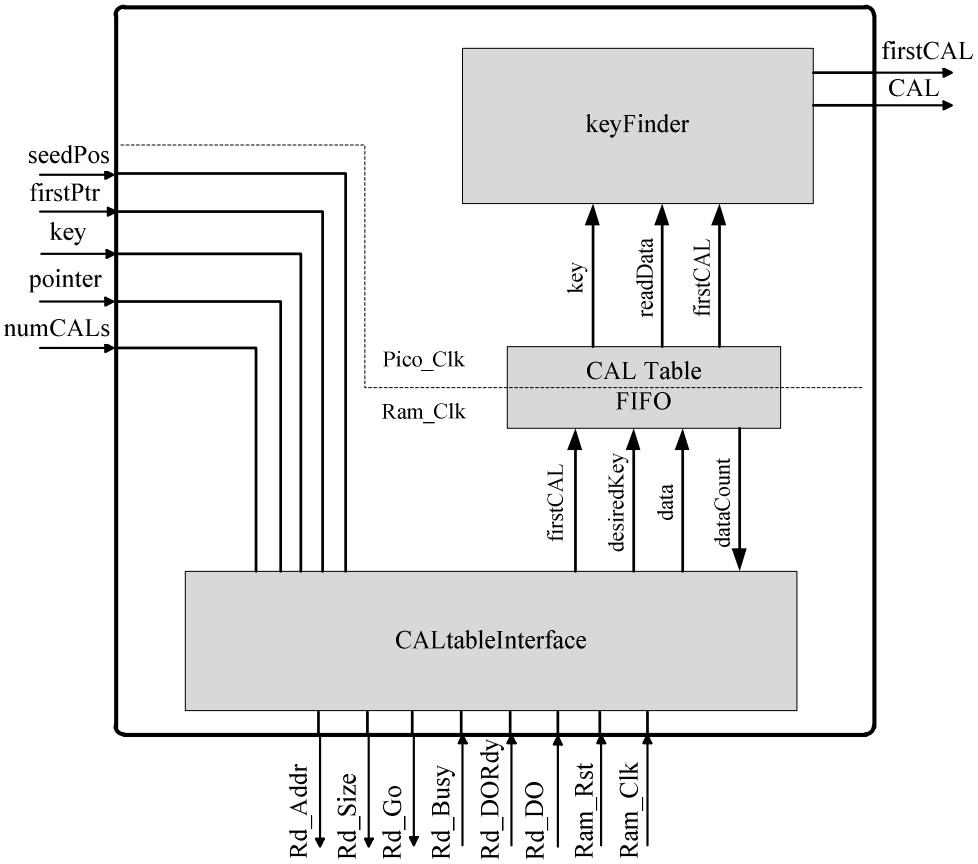

Figure 42: Diagram of the indexLookup module containing the CALtableInterface, for reading from the CAL table, and the keyFinder, for parsing the CALs from the CAL table.

5.3.2 CALFilter

The CALFilter module is responsible for reducing the set of CALs from the CALFinder module to a set of unique CALs for each read. In other words, CALs that have already been passed to the Aligner for a read should be ignored if they are seen again for the same read. This filtering process, which is described in section 3.3, is performed using a hash table and a version counter, and the block diagram is shown in Figure 43. The hash table is implemented using the block RAMs available on the FPGA. Each entry of the hash table stores a 32-bit CAL and a 32-bit version number. We hash the CAL to a 10-bit value, and the resulting the hash table contains 1024 total entries and consumes 64kb of block RAM space. Two 32kb block RAMs on the LX240T are used for this single memory.

The version counter in the CALFilter is used to rapidly erase the hash table by simply incrementing the version number, versus having to erase all entries of the hash table. The

version number is chosen to be 32 bits so it will not roll over unless mapping more than four billion reads, because multiple reads sharing a version number could potentially result in a CAL being filtered when it should actually be sent to the Aligner. This version counter does not require any correlation to a specific short read during the system runtime, but it must simply be unique for four billion reads. This permits us to use a linear feedback shift register instead of a standard counter, allowing the version counter to operate at a much faster clock frequency, due to the elimination of an adder's carry chain. When firstCALin is asserted, the version counter is updated, the input CAL is ignored and not written into the hash table, and the filter outputs a dummy firstCAL, which will later be used to update the FIFO containing the short reads.

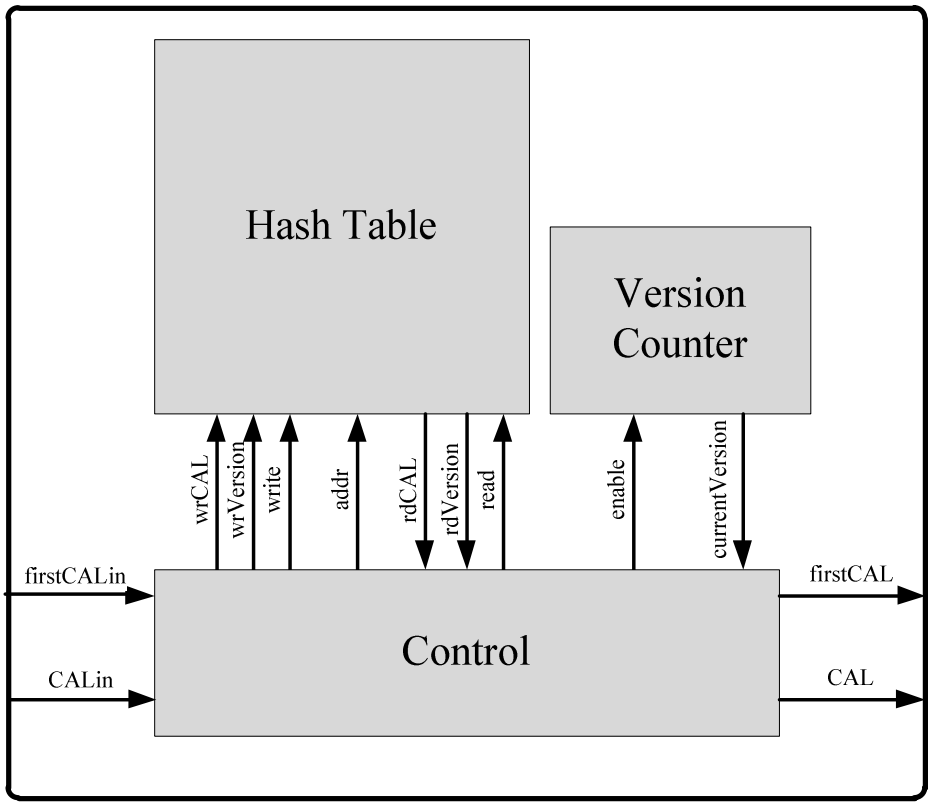

Figure 43: Block diagram of the CALFilter hardware logic containing a hash table, which is stored in block RAM, a version counter, which is implemented using an LFSR and control logic.

5.3.3 Aligner

The Aligner module accepts CALs from the CALFilter and reads with read IDs from the CALFinder. On the first CAL of a new short read, which is indicated by a firstCAL signal

from the CALFilter, the Aligner grabs the next short read and read ID from the read FIFO. For every CAL of a short read, the Aligner reads the DRAM through the RamMultiHead module to find the reference data for the CAL. It then loads the short read and the reference data into a Smith-Waterman compute engine. The Aligner contains a pool of Smith-Waterman engines, so it distributes the work by loading the read data into the next available Smith-Waterman unit in the pool. This is done in a round-robin fashion, because the Smith-Waterman comparison phase takes the same number of clock cycles for every CAL, which is equal to the number of bases of the reference against which the short read is being aligned. Results from all Smith-Waterman compute engines are reported to a single score tracking module, which then outputs the alignment and score of the CAL with the highest score for each short read. More details of the Aligner's implementation can be found in Maria Kim's Master's thesis [25].

5.4 Results

5.4.1 Resources

The system described in the previous sections was implemented with an index constructed for mapping short reads of length 75 base pairs to chromosome 21. Chromosome 21 was chosen for the prototype system because its index can fit in the 512MB of DRAM on the M501 board. The system operates in three clock domains. The main system clock runs at 250MHz, the memory interface clock at 266MHz, and the Smith-Waterman compute units at 125MHz, which is derived from the 250MHz main system clock. Table 1 and Figure 44 show the resources on the Virtex-6 FPGA used by each portion of the hardware system. The Stream Loopback is the infrastructure required for the M501 system to allow the host processor to communicate with the design on the FPGA and write the DRAM.

Table 1: Table of the resources used in each stage of the system. Infrastructure is the streaming loopback system provided by Pico Computing. Each entry represents the resources used only for that component of the system. The full system was also synthesized with one and four Smith-Waterman units to compare and identify the size of a single Smith-Waterman compute engine.

	Stream Loopback	CALFinder	CALFilter	1 S-W Unit	4 S-W Units	Available
Slices	3,459	744	10	6,759	17,239	37,680
LUTs	9,244	1,476	89	20,273	62,447	150,720
Registers	9,722	1,958	234	9,388	25,228	301,440
RAMB36E1/ FIFO18E1	24	0	2	0	0	416

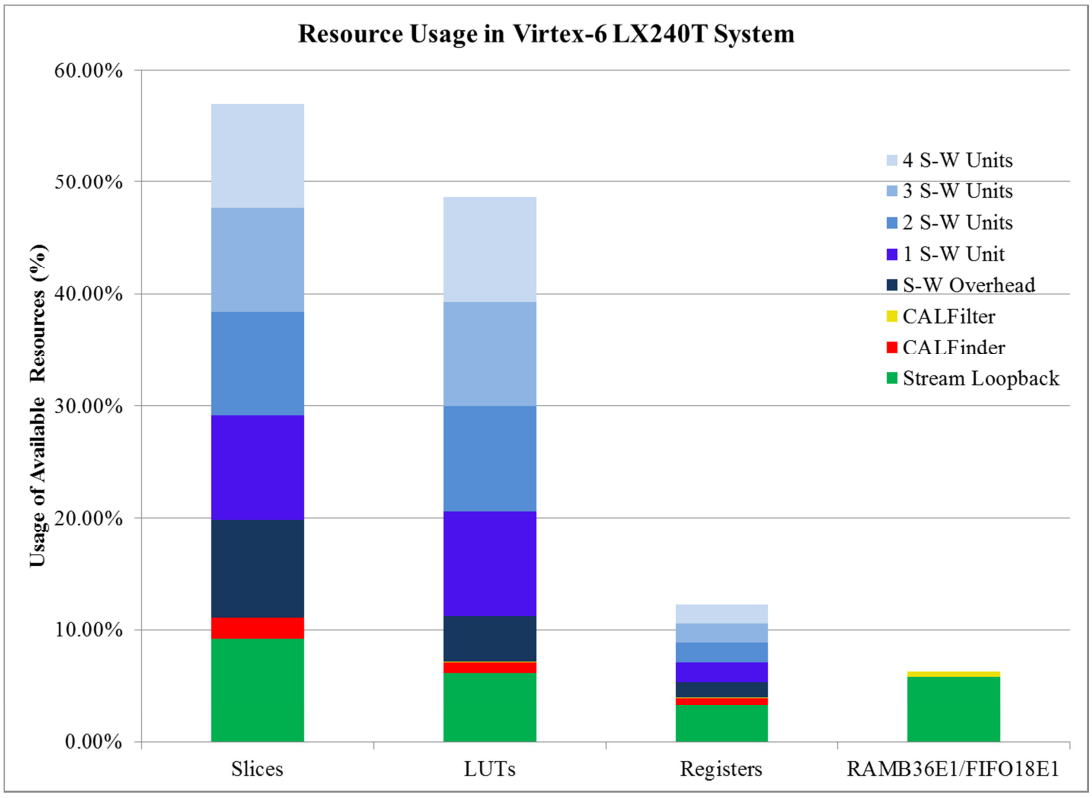

Figure 44: Resource usage of different stages of the system as a percentage of the available resources on the Virtex-6 LX240T FPGA.

Figure 44 shows that approximately 20% of the FPGA's LUTs and 30% of its slices are used once a single Smith-Waterman compute engine has been implemented, and those numbers jump to approximately 50% and 55% respectively for four Smith-Waterman engines. This means a single Smith-Waterman compute engine should consume approximately 10% of the FPGAs LUTs, allowing a single FPGA to hold up to eight of the currently designed compute

engines. Extending this design to a four-board system indicates the resulting system can contain up to 36 Smith-Waterman units.

5.4.2 System Runtime

The prototype system was run mapping short reads to chromosome 21, and the results are shown in Figure 45. For a comparison, the BFAST software was also run on a pair of Intel Xeon quad-core processors mapping the same sets of reads to chromosome 21. The runtime is shown in the graph for both the prototype and the BFAST runs. The runtime includes the time to load the index file into memory and to process all reads, for both the hardware and the software versions. Once the index file is loaded into memory in the hardware version, many sets of reads can be mapped without having to reload the index. However, BFAST loads the index every run, so it was included in both systems' runtime here for the sake of a fair comparison.

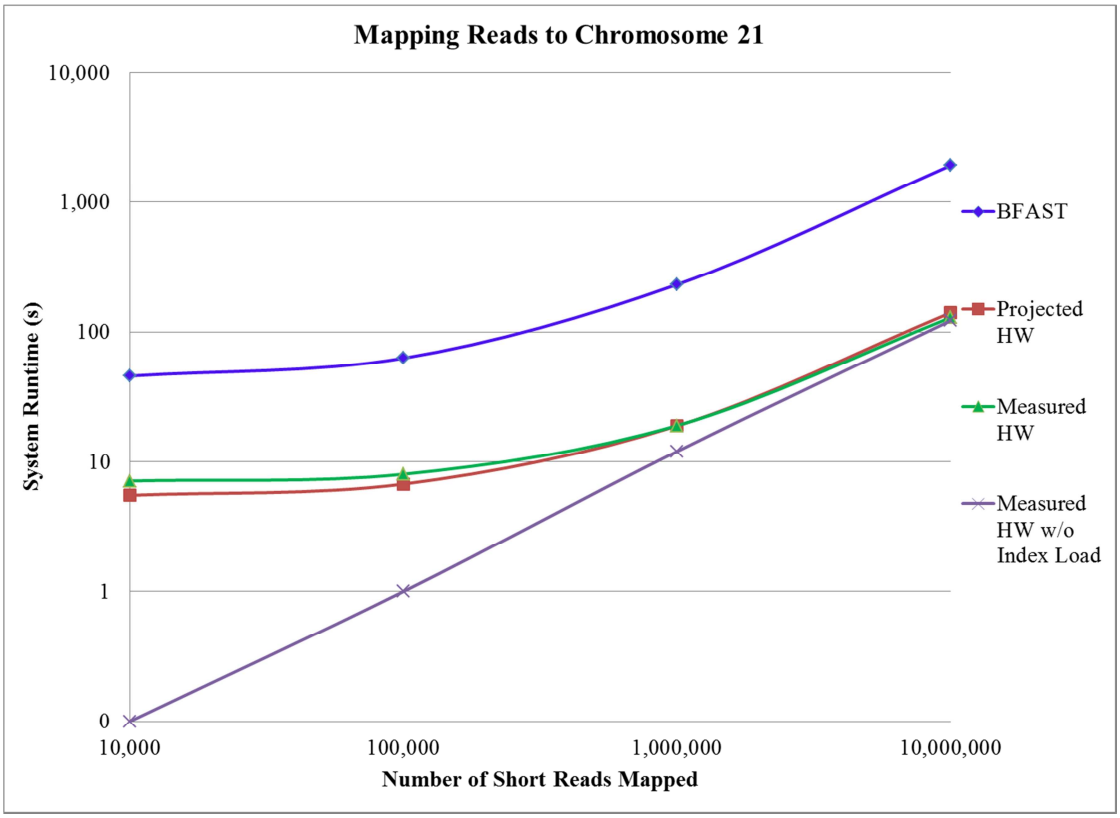

Figure 45: Projected and measured runtimes of the hardware system compared to measured runtime of BFAST mapping reads to chromosome 21. Runtime includes the time to load the index into memory, read short reads from disk, stream reads to the FPGA, align the reads, stream the results back to the host, and write results to disk.

The hardware system performs well compared to the projected throughput for aligning reads to chromosome 21. Over the full sample set of reads to map, the average runtime for the hardware system is 10.33 times faster than BFAST, with a maximum speedup of 14.86 when mapping 10 million reads to chromosome 21. The speedup is masked by the amount of time to load the index file when mapping a low number of reads, but the speedup gradually improves as the number of reads increases. The time to load the DRAM has a large effect on the performance of the system for a small number of reads, but this would be negligible in the hardware system, because the index can simply be loaded once and left in the DRAM.

Boris Kogon mapped short reads to the full human genome using BFAST on the same processors as used when mapping short reads to chromosome 21. BFAST maps short reads to the full genome in numerous steps, but only the match phase and align phases are used for comparison here. The match phase took 5 hours, 18 minutes, and 3 seconds, and the align phase completed in 57 minutes and 47 seconds, for a total runtime of 6 hours, 15 minutes, and 50 seconds (or 22,550 seconds).

For a comparison to the full genome read mapping running in hardware, which is described in sections 6.2, the hardware system is projected to map 54 million reads in approximately 85 seconds. This runtime improvement equals a 262x speedup compared to BFAST. The large speedup improvement when the dataset size increases is primarily because BFAST's runtime scales super-linearly with the size of the reference genome, due to the increased size of the index that must be searched and the increased number of CALs per read that must be aligned. Conversely, the runtime of the hardware system scales linearly with the size of the reference genome, resulting in a great improvement in throughput compared to BFAST. As shown in section 6.2, this speedup in system runtime also results in a great improvement in the power consumption for short read mapping, which is 194x less for the FPGA system compared to BFAST.

5.4.3 System Analysis

To determine where design effort should be focused in order to improve the short read mapping runtime, we now analyze the throughput of the various phases of the system. The phases that must be analyzed are: streaming reads to and from the FPGA via PCIe; aligning the CALs for a read; and performing all of the required DRAM accesses. The 600MB per

second PCIe bandwidth for streaming to and from the board enables the streaming of up to 18.7 million reads per second. The CALFilter is able to process one CAL per clock cycle, so it should never be the bottleneck of the system. However, aligning all the CALs for a read takes many clock cycles per CAL, and therefore may become the system bottleneck. Aligning one CAL with one Smith-Waterman compute unit (assuming the reference data has already been read from DRAM) takes a number of S-W clock cycles equal to the sum of the length of the short read and the length of the reference used in the comparison, which in our case is 267 clock cycles. This means a single Smith-Waterman compute unit can align at least 468 thousand CALs per second. If we assume an average of eight CALs per read, this results in aligning 58.5 thousand reads per second per Smith-Waterman compute unit. Finally, the memory system must access the pointer table 55 times per read, the CAL table 55 times per read, and the reference data eight times per read. These memory accesses result in a read processing rate for the memory system of 79 thousand reads per second per memory controller.

Based on these throughput estimates, the memory system is the bottleneck of the entire mapping algorithm (assuming we place at least two Smith-Waterman units per memory controller on an FPGA). Adding more than two Smith-Waterman compute engines per memory controller to the FPGA should have no effect on the current system runtime, because the memory cannot read reference data from DRAM fast enough to support more than two Smith-Waterman units.

6 Future Improvements

The system described in the previous sections was implemented for the purpose of producing a prototype system able to align short reads to chromosome 21 only. However, the goal of this project is to align reads to the entire human genome, which is about 66 times larger than the prototype chromosome. This section describes the necessary changes, both in the algorithm and the hardware system, to be able to align to the full human genome. This section also proposes some modifications to improve the system runtime.

6.1 Reverse Complement Strand

Short reads generated during the base calling step, which occurs before the short read mapping phase, may originate from either the forward or the reverse complement strand of the sample DNA sequence. Unfortunately, there is no way to identify if a read comes from the forward or reverse strand, and therefore every short read must be aligned against both the forward and reverse complement reference sequences. The prototype system neglects this alignment, but it cannot be overlooked in the full genome system. The forward and reverse complement of a read is shown in Figure 46.

```
          Reference: CGAGTTGGATTTGAGACCCAGGAGTATGATCGCTGAGCGGCGTAAATAGCGCTATGACGT
          Position : 000000000011111111112222222222333333333344444444455555555551
                     012345678901234567890123456789012345678901234567890123456789

Reverse Complement: ACGTCATAGCGCTATTTACGCCGCTCAGCGATCATACTCCTGGGTCTCAAATCCAACTCG
          Position : 555555555544444444443333333333322222222221111111111000000000
                     987654321098765432109876543210987654321098765432109876543210
```

Figure 46: Reads may originate from the forward strand or the reverse complement strand, so both must be indexed. When generating seeds from the reverse complement strand, CALs associated with the seed are the position of the last base in the reverse complement seed.

A simple way to map against the forward and reverse references would be to first align the forward version of all reads, find the reverse complement of all reads, and then align the reverse complements. This effectively doubles the number of short reads and causes the system runtime to double as well. If we instead only use the forward version of all short reads, we must index the forward and reverse complement reference genomes. Looking up the forward read in the index will return the CALs for both the forward and reverse complement reference sequences, but at the cost of a CAL table that has doubled in size. Once the index has been created for both directions, each valid location in the reference genome will appear twice in the CAL table, once for the forward seed and again for the reverse complement seed at that location. If we can eliminate one of those CALs, we can reduce the CAL table back to the original size.

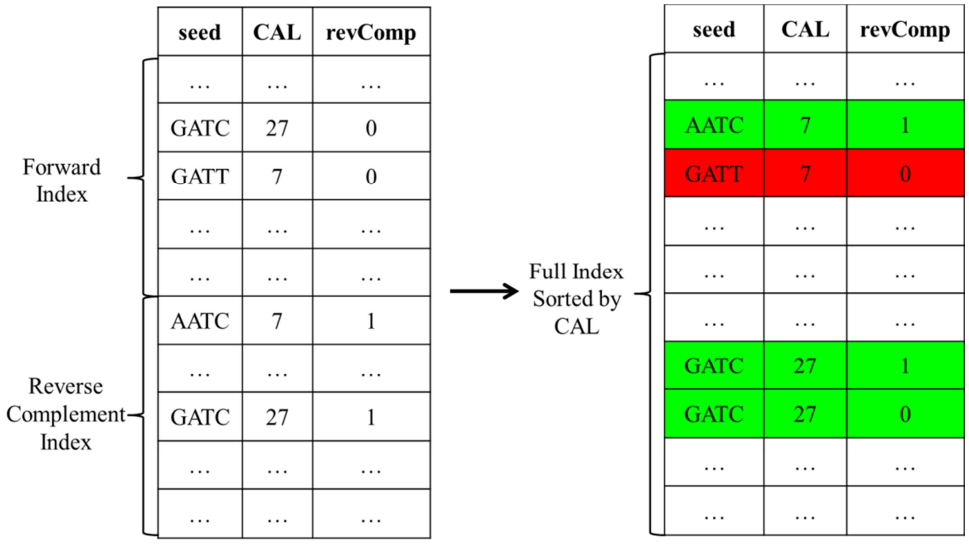

Figure 47: Forward and reverse strand indexes are combined into a single table (left) and then sorted by CAL (right). From this index, remove the lexicographically larger of the two seeds for every CAL in the index (red). Seeds that are their own reverse complement will have two entries in the table for every CAL.

Before accessing the pointer table, we can compute the reverse complement of a seed and use the lexicographically smaller of the two for the lookup in the pointer table. All CALs associated with the original seed and the reverse complement seed will be found in the CAL table entry specified by the lexicographically smaller seed. We only use the lexicographically smaller seed or reverse complement seed to access the index, so the entries in the CAL table associated with all lexicographically larger seeds can be deleted, resulting in a CAL table with a size equal to the original CAL table that was created for the forward reference only. Figure 47 shows the creation of the CAL table for the forward and reverse complement references, and the seeds in red get deleted from the CAL table before use. The reverse complement of the reference is aligned against the forward read during Smith-Waterman alignment if:

- the forward seed is used during the index lookup and the revComp bit is set for a CAL;
- the reverse complement seed is used in the index lookup and the revComp bit is not set for a CAL.

Otherwise, the forward read is aligned against the forward reference genome. The method of accessing the newly created CAL table is shown in Figure 48.

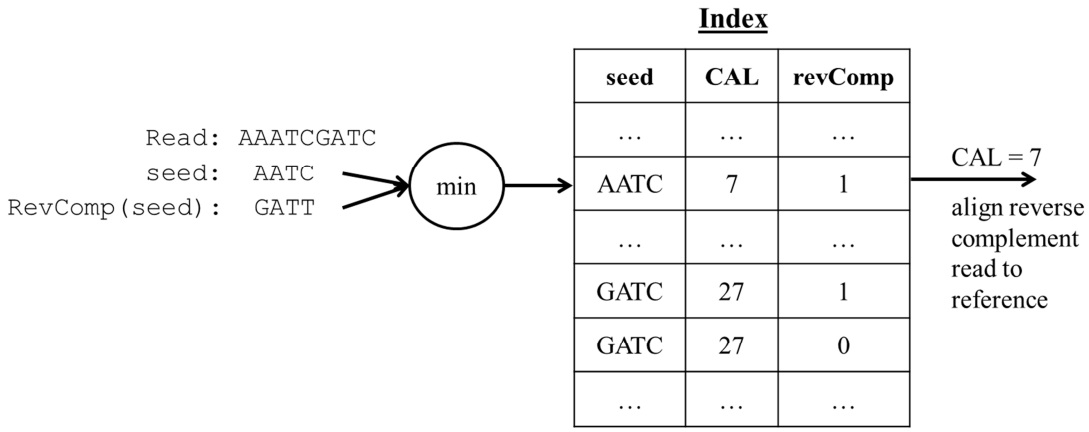

Figure 48: The lexicographically smaller seed or reverse complement of the seed should be used to access the pointer table, followed by the CAL table. If the forward seed is used and the reverse complement bit is not set for a CAL, or if the reverse complement seed is used and the revComp bit is set for a CAL, the forward read is aligned against the forward reference genome. Alternatively, the reverse complement of the read is aligned against the forward reference for the other two cases of seeds accessing the index.

This technique allows us to map forward and reverse complement reads at very little cost. The number of memory accesses and size of the CAL table remain the unchanged, but short reads get aligned to the forward and reverse complement strands, which doubles the average number of CALs per read, and will increase system runtime if the Aligner is the bottleneck.

6.2 M503 System

Neither the M501 nor the M503 boards contain sufficient memory to store the index for the full human genome, so one of two options must be pursued to map reads to the full genome. The system must comprise either 40 M501 boards or at least three M503 boards to hold the approximately 20GB reference genome index. Each M503 contains 8GB of DRAM, which is split into two 4GB DIMMs, which are controlled by separate memory controllers. We are able to store approximately one quarter of the full genome index in 8GB of DRAM, so four M503 boards will be needed for the final system. By chaining boards together, we can stream reads into one board and results out another board, creating a pipeline of reads aligning to different partitions of the reference genome, as shown in Figure 32. It is necessary for boards that are not the first in the chain to know the previous best score and alignment, so that data is now passed into a board with the read and ID, and the read and ID are also passed to the output, as shown in Figure 49. The host software sends in the most negative possible score with each read, and the track module of each partition will be loaded

with the current best score before performing any alignments. To avoid redundant work (work that was already performed by a previous FPGA) for the Smith-Waterman compute units, the CALs for a read can also be sent along with the read and then used to initialize the CALFilter. This efficiency improvement may be necessary for systems that are limited by the Smith-Waterman compute power, which is not currently the case in this system.

256-bit Stream Data				
Read ID	Alignment	Score	Reserved 32 bits	MS Read Data
LS Read Data				

Figure 49: Consistent streaming data structure to enable chaining of results from one FPGA to provide the input data to another FPGA.

The infrastructure for the M503 is very similar to that of the M501. However, the PicoBus, which is used for the streaming communication, is increased to 256 bits, instead of the 128 bits on the M501 system. This bus width tends to work well for this algorithm, because 76 base reads now fit with their ID, score, and alignment location in one transfer, and memory accesses to DRAM are already performed in 256-bit blocks. With the 256-bit bus, the seedFinder no longer needs to assemble reads, which were previously transmitted from the input stream in two transfers.

When mapping to the full genome, we partition the index amongst N M503 cards and their DRAM. One way to do this is to partition by CAL, and we can build an index for each set of consecutive bases in the genome, where each partition contains 3,000,000,000/N CALs. This means we need to store the full pointer table, one N^{th} of the full CAL table, and one N^{th} of the reference data in each partition. If we instead partition the data by the first $\log_2(N)$ bits of a seed, we store one N^{th} of the pointer table, one N^{th} of the CAL table, and the full reference in each partition.

These two options require roughly the same memory footprint, because the CAL table for each partition, which is the largest part of the index, is one N^{th} of the original CAL table size in both cases. However, the second option reduces the number of DRAM accesses in each partition by a factor of N, because the pointer table and CAL table only need to be accessed if the first $\log_2(N)$ bits of the seed are the target bits for that partition. A problem with this

implementation is that the distribution of bases throughout the reference genome is not uniform, and therefore we could incur a load balancing problem. Recall that the seed hashing function was designed to randomize seeds, and therefore should handle the load balancing issue.

The memory on the M503 is made up of two 4GB DRAMs, each with a separate memory controller, effectively doubling the bandwidth to the memory, as long as it is used efficiently. If we have a system comprising four M503 boards, we can now partition the index eight ways as described previously and achieve a factor of eight memory bandwidth improvement. Assuming the hashing function uniformly distributes CALs across all seeds, the 20GB full genome CAL table can be split into 8x2.5GB partitions. The 2GB pointer table is divided into 0.25GB partitions, and the full reference data, which is approximately 1GB, is loaded onto every partition, resulting in 3.75GB of the 4GB DRAM being filled in each partition.

This partitioning methodology creates an 8x improvement in memory bandwidth, which should result in a similar speedup of the memory limited system. This means full genome short read mapping for 200 million reads should take approximately 5.2 minutes. Comparing this full genome runtime to the performance of BFAST software mapping, this hardware system has a projected throughput that is 262x greater than the BFAST software running on a pair of Intel Xeon quad core processors with 24GB of available DRAM. The breakdown of the projected runtime for each non-overlapping portion of the hardware system is shown in Table 2.

Table 2: Full genome mapping time of 54 million short reads using a system comprising four M503 boards. System runtime is maximum time to perform any of these tasks, which can be performed in parallel.

Task	Runtime (s)
DRAM Accesses	85.91
Disk Access	8.71
Streaming Reads to Board	2.90
Streaming Results from System	1.45
Smith-Waterman Alignments	43.89
System Runtime	85.91

The speedup here can not only yield a great improvement in the mapping time, but it can also yield a large power savings used during the short read mapping. Assuming our host machine consumes 400 Watts of power during operation, and assuming each M503 FPGA in the system consumes 35 Watts of power during operation, we can compute the total energy consumption for the BFAST and the FPGA acceleration versions of the system. The BFAST system, which ran for more than six hours to map 54 million short reads to the full genome, consumes a total of 2.5kW-hours of power. This full genome FPGA system instead only takes about 1.4 minutes and consumes 0.01kW-hours of power, which results in a 194x improvement in power consumption for mapping 54 million reads.

6.3 Pipelined DRAM Reads

DRAM accesses are the bottleneck of this system, so to improve system runtime we need to either reduce the number of accesses to memory or increase the memory bandwidth. Previous sections have focused on reducing the number of accesses to the memory, so now we focus on improving the bandwidth to the DRAM. The RamMultiHead module only allows one read command to be outstanding to the DRAM at any one time. However, the Xilinx memory interface generator (MIG) supports pipelined read requests, meaning multiple reads can be issued to the DRAM before the first data is returned. Pipelining read requests to the DRAM allows the user to hide communication latency to the DRAM with subsequent reads.

The RamMultiHead module must be modified to enable pipelining of read requests. It is still necessary to handle the arbitration of multiple read ports, but it should not require the address and size to be held constant on the input for the duration of a read. Also, the controller needs to direct returning read data from the DRAM to the correct read port, because data returns in the same order as the read request, but may be from any of the read ports.

Once these changes have been implemented, assuming we are able to sustain N read requests in flight to the DRAM at one time, the memory bandwidth should improve by a factor of N. If N is equal to three, the memory bandwidth improves by a factor of three, and the memory ceases to be the bottleneck of the FPGA system. The time to perform Smith-Waterman alignments for all CALs becomes the bottleneck, which can be reduced by increasing the number of Smith-Waterman compute units on each partition. We are able to increase the

number of Smith-Waterman compute units per FPGA from four to six, because the pipelined reads now allow for each memory controller to support three compute units. The increased number of Smith-Waterman compute units per FPGA means the memory system is once again the bottleneck of the mapping system.

After these changes, the time required to map 54 million reads to the full human genome reduces to approximately 32 seconds. Comparing this projection to the previously determined runtime of the BFAST software, this full genome hardware system has a projected throughput that is 692x greater than the BFAST software running on a pair of Intel Xeon quad core processors with 24GB of available DRAM. The breakdown of the projected runtime for each non-overlapping portion of the hardware system is shown in Table 3. Once again, this increased throughput also leads to a power improvement compared to the software system. The system using pipelined DRAM reads now has 513x lower energy consumption compared to BFAST.

Table 3: System runtime for mapping 54 million reads assuming we can pipeline accesses to the DRAM, assuming we can have three outstanding read requests at one time, and knowing we can fit 6 Smith-Waterman compute engines per FPGA.

Task	Runtime (s)
DRAM Accesses	32.57
Disk Access	8.71
Streaming Reads to Board	2.90
Streaming Results from System	1.45
Smith-Waterman Alignments	29.26
System Runtime	32.57

6.4 Pointer Table in SRAM

Recall that the M503 board contains three SRAM chips, each of which is 72Mb, for a total of 27MB per board. Another option to further increase bandwidth to the tables in memory is to utilize this much faster SRAM. Both the reference and the CAL table, which are 1GB and 5GB per board respectively, are too large to be stored in the small SRAM. However, the size of the pointer table can be reduced to a point where it can fit in this SRAM. Since we have already partitioned the CAL table into two partitions per board, we need to maintain two pointer tables per board as well, one for each partition. Hence, the pointer table must be smaller than 13.5MB for one partition.

The original 2GB pointer table uses 26 bits for address and 4 bits for tag. Partitioning the CALs amongst eight memory controllers reduces the pointer table to 256MB per partition. In order to pack the pointer table into 13.5 MB, we can use five less bits for the address, which reduces the pointer table by a factor of 32 down to 8MB, assuming four bits are still used for tag and each offset field is 14 bits. If each base in the reference genome has an associated seed, which is not always accurate due to the existence of large unknown sections of chromosomes, three billion seeds get distributed through 4M addresses, resulting in each address pointing to an average of 1000 CALs. Therefore, 14-bit offsets should be large enough to offset between subsequent start pointers from the pointer table.

However, by reducing the number of address bits, we have increased the average number of CALs that will have to be parsed for each read of the CAL table. Recall the keyFinder takes a clock cycle to parse each result from the CAL table, and using 21 bits of the seed for address and four bits for tag means the average CAL table bucket will contain 64 CALs. Parsing 64 CALs for every CAL table access would require two keyFinder modules working in parallel to avoid slowing the system performance. However, as can be seen in Table 4, moving the pointer table into 27MB of SRAM actually slows the performance of the system. This is because the pointer table only fits within this SRAM if using 5 fewer bits for address, and therefore almost 100 CALs get read from each CAL table access. The pointer table in SRAM system still has a 289x speedup and a 214x energy improvement compared to BFAST.

Table 4: System runtime for mapping 54 million reads to the full genome if the pointer table on each M503 board is stored in the 27MB of available SRAM.

Task	Runtime (s)
DRAM Accesses	78.03
SRAM Accesses	5.99
Disk Access	8.71
Streaming Reads to Board	2.90
Streaming Results from System	1.45
Smith-Waterman Alignments	43.89
System Runtime	78.03

If a future version of the M503 were to have 128MB of SRAM, we could fit the two partitioned pointer tables, 64MB each, in SRAM by using 24 bits for address. Using 24 bits for address and four bits for tag implies we must parse approximately 12 CALs per CAL table access, which can be done easily in the number of clock cycles necessary to complete each CAL table read. If this happens, each pointer table access only takes one clock cycle from the SRAM, and the CAL table accesses only share the DRAM controller with the reference lookups. This leads to an improvement in the reads per second throughput for the memory system, as shown in Table 5, which can then support eight Smith-Waterman units per FPGA. The new system with the pointer table in a large SRAM has a 981x speedup and a 727x power reduction compared to BFAST.

Table 5: System runtime for mapping 54 million reads to the full genome if the pointer table is moved into a 128MB SRAM.

Task	Runtime (s)
DRAM Accesses	22.99
SRAM Accesses	5.99
Disk Access	8.71
Streaming Reads to Board	2.90
Streaming Results from System	1.45
Smith-Waterman Alignments	21.94
System Runtime	22.99

6.5 CALFinder on Host

The major issue that we have addressed in the previous few sections is how to reduce the number of accesses to the DRAM, as well as improve the performance of the required accesses. This is mainly due to the relatively low bandwidth between the FPGA and the DRAM for random access reads of short burst lengths. However, up until this point, we have neglected the large amount of high bandwidth memory that is available on the host machine. The host machine has two Nehalem 2.66GHz processors, each having four processing cores. Each processor is paired with three memory controllers, where each memory controller is responsible for 4GB of main system memory, for a total of 24GB memory installed in the system. This is ample space to store the index for the human genome.

If we pull the pointer and CAL table lookups back into the host system, as shown in Figure 50, we can use the high-bandwidth memory on the host to potentially improve the

performance of the CALFinder. However some of the operations done in the CALFinder, such as the hashing function, finding seeds, or computing reverse complements of seeds, are not as efficient on a software processor as in dedicated hardware. In other words, we want to run the CALFinder on the host processor to take advantage of the high-bandwidth memory, but in doing so, we may not be able to generate seeds fast enough to take advantage of the memory speed improvements.

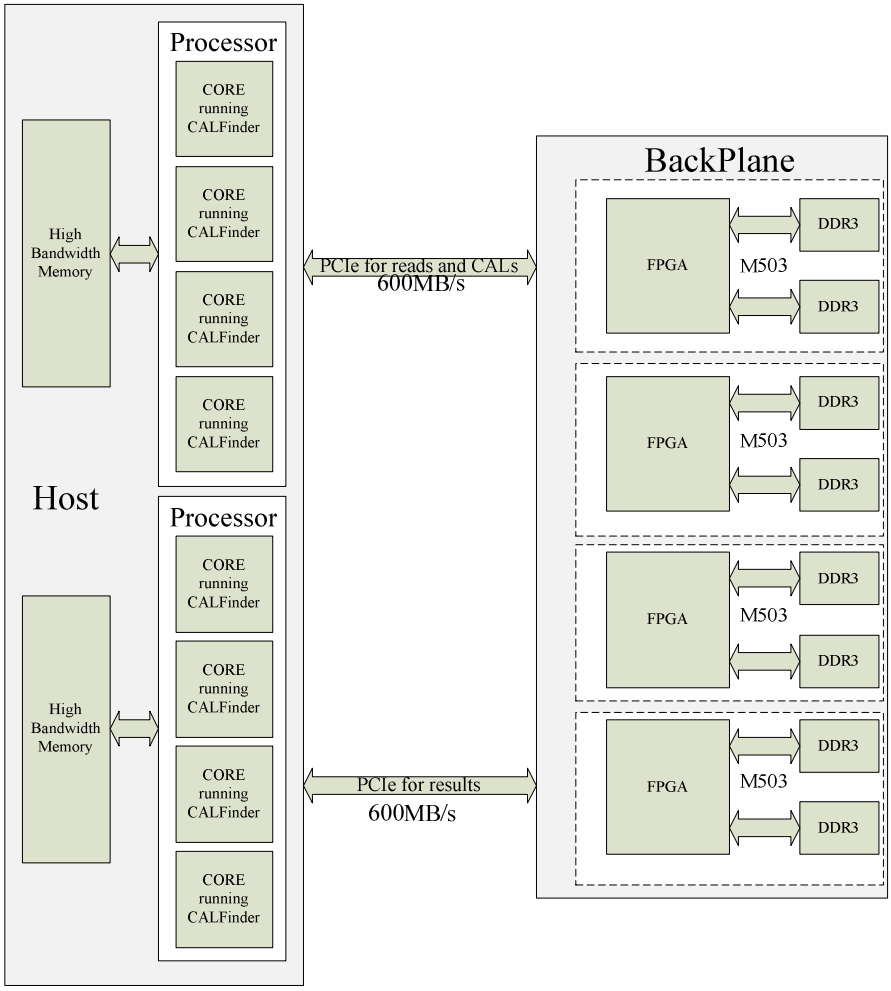

Figure 50: CALFinder running on the host to take advantage of the high bandwidth memory on the host system. The host must now transmit reads and CALs to the system, which is only responsible for computing Smith-Waterman alignments and reporting results back to the host.

For example, assume we could maintain 280 million memory references per second per processor, which is a very aggressive estimate, but can be done by allowing several read requests at a time. Using the bandwidth of both processors yields 560 million memory

references per second. Each seed must be looked up once in the pointer table and once in the CAL table, meaning 280 million seeds can be processed per second, or five million short reads per second, if using 76 base short reads and 22 base seeds. Five million short reads per second means that finding CALs for 200 million reads will take 40 seconds. To confirm that this will be the actual execution time of the system with the CALFinder on the host, we must ensure that we can generate seeds for the pointer table lookup, transmit CALs to the FPGA system, and align the CALs on the FPGA; all of these need to be done at a fast enough rate to sustain the estimated memory bandwidth.

Generating seeds on the host is rather simple by using some bit masking and shifting; however, hashing the seed will take much too long using a complex hashing function. At 280 million references per second on a 2.66GHz processor, we need to generate a new seed every nine clock cycles. However, by using all four cores on the processor, we can instead generate a new seed on a core every 36 clock cycles, in which time we must hash a seed, generate the reverse complement of the hashed seed, and determine which is lexicographically smaller. A simple hashing function can be used to reduce the number of cycles spent finding the hashed seed. Also, a lookup table can be utilized to quickly find the reverse complement of a hashed seed. However, determining which seed is lexicographically smaller may be difficult to accomplish in the remainder of the original 36 clock cycles.

Transmitting the read data and CALs to the FPGA via PCIe will require five million reads and 40 million CALs, assuming eight CALs per read, to be transmitted per second. We already pack a read into 32 bytes for transfers, and we can easily pack the other eight CALs into another 32 bytes, yielding 64 bytes per read. Transmitting five million reads and their CALs per second will then require approximately 500MB per second PCIe bandwidth, which should be easily attainable with the M503 board.

Lastly, we must align five million reads to eight CALs per read per second, so that we can compute the total number of required Smith-Waterman compute engines. Each alignment for a read to a CAL requires approximately 200 clock cycles, where the user clock is running at 125MHz, meaning the CAL alignment takes 1.6 microseconds. At that rate, aligning 40 million CALs takes 64 seconds; hence we must have 64 Smith-Waterman compute engines operating in parallel in order to align five million reads per second. Since finding the CALs

is moved into the host memory for this analysis, each memory controller can read enough reference data from DRAM to support 36 Smith-Waterman compute engines (with DRAM pipelining). However, that many Smith-Waterman compute engines cannot fit within a single FPGA, given the resource data from section 5.4.1. To combat this problem, a technique known as C-slowing can be used to interleave the computation for two different CALs while running on a clock that is double the original frequency. This technique can improve the throughput of the system without greatly increasing the resource requirements for the system. Using C-slowing, 64 Smith-Waterman units can fit into two M503 Virtex-6 FPGAs, and therefore the memory system on the host remains the bottleneck for the system, keeping the runtime at 40 seconds. If this system were to be used to map 54 million short reads to the full genome, for a comparison to BFAST, it would see a 2109x speedup and a 1795x reduction in energy.

6.6 Genomics System Architecture

The improvements discussed in the previous sections have aimed at improving the running time of this algorithm on an existing system, which is currently the M503 board. Some architectural changes can potentially lead to a further speedup of system runtime, as well as a general purpose reconfigurable system for accelerating genomics applications; we name this new system the Pico-Bio system and will further explore it in this section.

The proposed architecture of the Pico-Bio system, as shown in Figure 51, includes several aspects of the original M503 board. For example, each FPGA board in the new system should contain a PCIe interface, an interface to DRAM, and an SRAM. The SRAM on each FPGA board should ideally be large enough to store a mid-sized table, such as a 128MB pointer table, instead of the 27MB available on the M503. The system should use a PCIe switch with a bus to and from the host and each of the FPGAs, which can be programmed to form various communication links; we will use it to create a daisy-chain of the FPGAs, which begins and ends at the host processor. Based upon the computational requirements of the short read mapping algorithm, genomics applications seem to be memory bandwidth limited, so we must simply be able to keep the memory system running at the full bandwidth in order to accelerate these applications as much as possible. Therefore, the most critical

modification for a system such as the M503 is the improvement of the memory throughput for random access patterns reading small amounts of data with each access.

Scatter-gather DRAM units can provide greatly improved random access memory bandwidth by efficiently performing DRAM reads and returning data in out of order execution. The Convey HC-1 system uses a very intelligent memory system, consisting of eight memory controllers, each controlling two 1.5GB scatter-gather DRAMs [29]. In the HC-1, each of the FPGAs has a high-bandwidth link to each of the eight memory controllers, resulting in a peak system memory bandwidth of 80GB per second. Implementing this memory system can greatly improve the performance of many genomics applications, including short read full genome mapping.

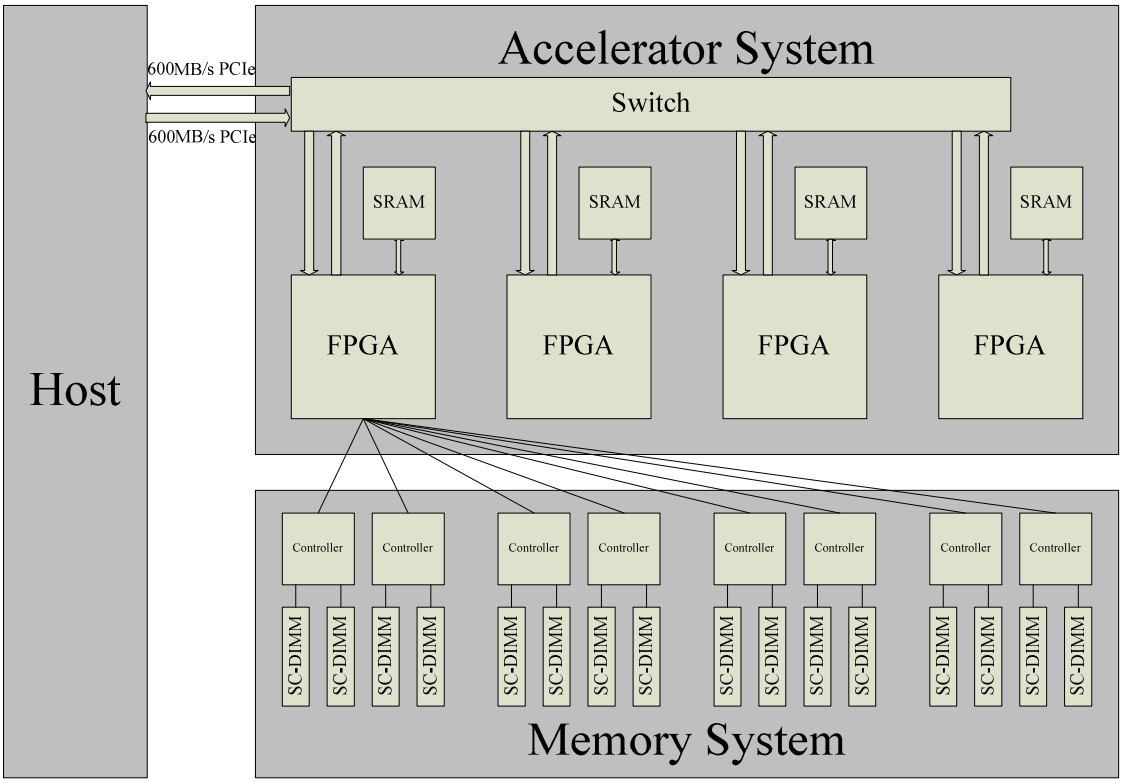

Figure 51: Proposed genomics accelerator system including four FPGAs, each paired with SRAM and communication links to a PCIe switch. The system memory is comprises 16x1.5GB Scatter-Gather DRAM, for a total of 24GB, and efficient memory controllers to handle out of order data retrieval. Each FPGA is connected to each of the DRAM controllers through a high-bandwidth communication channel.

Assuming we create the Pico-Bio hardware described in this section, and assuming we are able to implement an 80GB per second memory bandwidth spread over four FPGAs and

these eight memory controllers, we can evaluate the new system runtime for mapping 200 million 76 base pair short reads. Presuming for now the memory system will still be the bottleneck of the system (which may be wrong but can be adjusted later), each read requires 55 accesses to the pointer table, which may be stored in either SRAM or DRAM. Each short read requires 55 accesses to the CAL table in DRAM and eight reference lookups in DRAM per short read.

Regardless of where we store the pointer table, assuming we have eight CALs per short read, we read 48 bytes of reference data from DRAM for each CAL, or 384 bytes per short read. If we store the pointer table in 128MB of SRAM per FPGA, we need to use 24 bits for address; hence the CAL table stores an average of 11 CALs per CAL table bucket. Expanding this result to the requirement for one short read means we need to read 4.9kB of data from the CAL table in DRAM for every short read. These requirements result in 5.3kB of reference and CAL data being read from the DRAM and 1.7kB of pointer table data being read from the SRAM for every short read. At an 80GB per second bandwidth for the DRAM, this system can process 15 million short reads per second.

If we instead store the pointer table in DRAM, we can use 26 bits for address, and each access to the CAL table returns an average of 3 CALs, or 24 bytes of CAL data. This means each short read requires 1.2kB of CAL data to be read from DRAM. Since we now store the pointer table in DRAM as well, we must also read the 1.7kB of pointer table data per short read, bringing the total DRAM requirement per short read to 3.3kB. At an 80GB per second bandwidth for the DRAM, this system can process 23.7 million short reads per second. Therefore, the following analysis will assume the pointer table is also stored in DRAM. If the memory truly is the system bottleneck, the total execution time for 200 million short reads will be about 8.4 seconds.

In order for the memory system to be the bottleneck, the PCIe must be able to send 23.7 million reads worth of data to the system per second, and the FPGAs must be able to align 23.7 million reads worth of data per second. Each short read is transmitted to the system in 32 bytes, which contains the read ID, the read data, the best current alignment, and the alignment score. Transmitting 23.7 million short reads per second means the PCIe must be

able to support a continuous transfer rate of 760MB per second, which is slightly larger than the 600MB per second PCIe bandwidth of the current M503 system.

In order to align 23 million short reads per second, we simply need to be able to fit enough Smith-Waterman compute engines in each of the FPGAs in our system. Since each short read has an average of eight CALs, aligning 23 million short reads per second actually means aligning 190 million CALs per second. Assuming a Smith-Waterman engine clock rate of 125 MHz and that each alignment takes approximately 192 clock cycles, which is determined by the length of the reference against which the short read is aligned, a short read can be aligned against a single CAL by a Smith-Waterman compute engine in 1.536 microseconds. Given this compute time, aligning 184 million CALs per second requires 282 Smith-Waterman compute engines in the full system. Each FPGA must then be able to fit 72 Smith-Waterman compute units in order to keep the memory system the bottleneck. This number of Smith-Waterman compute units is too large to fit in a single FPGA, but using c-slowing, we may be able to fit 32 compute units per FPGA, which would result in 200 million reads being aligned in about 19 seconds. Using a larger FPGA that could hold more Smith-Waterman compute units could potentially reduce the runtime of the system by about a factor of two, so that may be another option to pursue in the future.

Table 6: Table showing system runtime for mapping 200 million short reads using the proposed Pico-Bio system with scatter-gather DRAM, enough SRAM to hold a partition of the pointer table, 600MB per second PCIe links, and four FPGAs with 32 Smith-Waterman compute engines on each FPGA.

Task	Runtime (s)
DRAM Accesses	8.43
Streaming Reads and Results	10.67
Smith-Waterman Alignments	19.20
System Runtime (s)	19.20

7 Conclusion

This thesis has provided an algorithm for accelerating short read human genome mapping, and it has demonstrated an implementation of the algorithm mapping reads to chromosome 21 of the human genome. Comparisons to BFAST illustrate not only a peak speedup for the hardware system of 14.8x for mapping to chromosome 21, but also a projected speedup for

the full genome of 692x when the system is implemented on the M503 boards. The decreased system runtime when mapping with the M503 system also yields a 513x reduction in energy for mapping reads to the full human genome.

We proposed algorithmic changes to improve the performance of the system when mapping to the full genome. We also we described the hardware and components of a system that would efficiently accelerate genomics applications, which we called the Pico-Bio system. A summary of the proposed changes and the resulting speedup and power improvements when mapping 54 million short reads to the human genome can be seen in Figure 52.

Approach	M503 Boards	DRAM	SRAM (each FPGA)	S-W Units (each FPGA)	Runtime (s)	Speedup	Energy Savings
BFAST	-	-	-	-	22,550	1x	1x
Full Genome System	4	-	-	4	85.91	262x	194x
Pipelined DRAM	4	3x pipelined reads	-	6	32.57	692x	513x
Pointer Table in SRAM	4	-	27MB	2	106.51	211x	156x
Pointer Table in SRAM w/ pipelined DRAM	4	3x pipelined reads	27MB	4	78.03	289x	214x
Pointer Table in Large SRAM	4	-	128MB	4	51.46	438x	324x
Pointer Table in Large SRAM w/ pipelined DRAM	4	3x pipelined reads	128MB	8	22.99	981x	727x
Host CALs	4	-	-	16	10.70	2109x	1562x
Host CALs w/ pipelined DRAM	2	3x pipelined reads	-	32	10.70	2109x	1795x
Pico-Bio	4	80GB/s Scatter-Gather	-	32	5.23	4316x	-
Pico-Bio w/ SRAM	4	80GB/s Scatter-Gather	128MB	32	5.23	4316x	-
Pico-Bio w/ SRAM and big FPGA	4	80GB/s Scatter-Gather	128MB	72	2.32	9711x	-

Figure 52: This figure shows the projected runtime, computed speedup, and computed energy savings versus BFAST when mapping 54 million short reads to the human genome for various proposed algorithmic changes. The Pico-Bio system shows the largest speedup, but it also requires the most work to achieve that speedup. The host CALs option also produces a very large speedup, but the changes required are much less than those for the Pico-Bio system.

Some of the proposed solutions, such as the Pico-Bio system, require a significant amount of work to achieve a substantial speedup and energy savings. However, other alternatives, such as moving the finding of the CALs to the host processor, can achieve a considerable speedup compared to BFAST with much less design effort.

References

[1] Illumina Inc. (2011, April) Illumina - Sequencing Technology. [Online]. http://www.illumina.com/technology/sequencing_technology.ilmn

[2] Applied Biosystems. (2011, May) 5500 Genetic Analyzer. [Online]. https://products.appliedbiosystems.com

[3] N. Homer, B. Merriman, and S. F. Nelson, "BFAST: An Alignment Tool for Large Scale Genome Resequencing," *PLoS ONE*, vol. 4, no. 11, p. e7767, November 2009.

[4] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biology*, vol. 10, no. 3, p. R25, March 2009.

[5] Heng Li and Richard Durbin, "Fast and accurate short read alingment with Burrows-Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754-1760, July 2009.

[6] Heng Li, Jue Ruan, and Richard Durbin, "Mapping short DNA sequencing reads and calling variants using mapping quality scores," *Genome Research*, vol. 18, no. 11, pp. 1851-1858, November 2008.

[7] Stephen M. Rumble et al., "SHRiMP: Accurate Mapping of Short Color-space Reads," *PLoS Computational Biology*, vol. 5, no. 5, May 2009.

[8] Edward Fernandez, Walid Najjar, Elena Harris, and Stefano Lonardi, "Exploration of Short Reads Genome Mapping in Hardware," in *2010 International Conference on Field Programmable Logic and Applications*, Milano, 2010, pp. 360-363.

[9] U.S. Department of Energy. (2009, October) Human Genome Project. [Online]. http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml

[10] Biochem.co. (2011, April) DNA Replication | Biochem.co - Biochem & Science Notes. [Online]. http://biochem.co/2008/07/dna-replication/

[11] Marcel Marguilies et al., "Genome sequencing in microfabricated high-density picolitr reactors," *Nature*, vol. 437, no. 7057, pp. 376-380, September 2005.

[12] Michael C Schatz, Arthur L. Delcher, and Steven L. Salzberg, "Assembly of large genomes using second-generation sequencing," *Genome Research*, vol. 20, pp. 1165-1173, May 2010.

[13] Erik Pettersson, Joakim Lundeberg, and Afshin Ahmadian, "Generations of sequencing technologies," *Genomics*, vol. 93, no. 2, pp. 105-111, February 2009.

[14] United States Department of Health and Human Services. (2011, April) Genome.gov | DNA Sequencing Costs. [Online]. http://www.genome.gov/27541954

[15] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Palo Alto, CA, Technical report 124, 1994.

[16] T. F. Smith and M. S. Waterman, "Identification of Common Molecular Subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195-197, March 1981.

[17] Paolo Ferragina and Giovanni Manzini, "Opportunistic Data Structures with Applications," in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, Washington, DC, 2000, p. 390.

[18] Heng Li and Nils Homer, "A survey of sequence alignment algorithms for next-generation seqeuncing," *Briefings in Bioinformatics*, vol. 2, no. 5, pp. 473-483, September 2010.

[19] Ian Kuon and Jonathoan Rose, "Measuring the Gap Between FPGAs and ASICs," in *International Symposium on Field Programmable Gate Arrays*, Monterey, 2006, pp. 21-30.

[20] Pico Computing. (2010) Pico Computing - the FPGA Computing Experts. [Online]. http://www.picocomputing.com/m_series.html

[21] Xilinx Inc. (2010, January) Virtex-6. [Online]. http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[22] Saul B. Needleman and Christian D. Wunsch, "General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443-453, March 1970.

[23] C. Yu, K. Kwong, K. Lee, and P. Leong, "A Smith-Waterman Systolic Cell," in *New Algorithms, Architectures and Applications for Reconfigurable Computing*, Patrick Lysaght and Wolfgang Rosenstiel, Eds. United States: Springer, 2005, ch. 23, pp. 291-300.

[24] Peiheng Zhang, Guangming Tan, and Guang R. Gao, "Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform," in *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications: held in conjunction with SC07*, New York, New York, 2007, pp. 39-48.

[25] Maria Kim, *Accelerating Next Generation Genome Reassembly: Alignment Using Dynamic Programming Algorithms in FPGAs*, University of Washington, Dept. of EE, MS Thesis, 2011.

[26] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403-410, October 1990.

[27] William H. C. Ebeling and Corey B. Olson, "A Proposed Solution to the Short Read Reassembly Problem," University of Washington, Seattle, 2010.

[28] Cooper Clauson, Chromosome 21 Index Proposal, 2010.

[29] Convey Computer. (2008, November) Convey Computer. [Online]. http://www.conveycomputer.com/Resources/ConveyArchitectureWhiteP.pdf

[30] Avak Kahvegian, John Quackenbush, and John F Thompson, "What would you do if you could sequence everything?," *Nature Biotechnology*, vol. 26, no. 10, pp. 1125-1133, October 2008.

[31] Michael Snyder, Jiang Du, and Mark Gerstein, "Personal genome sequencing: current approaches and challenges," *Genes and Development*, vol. 24, no. 5, pp. 423-431, March 2010.

[32] Jason D. Bakos, "High-Performance Heterogeneous Computing with the Convey HC-1," *Computing in Science & Engineering*, vol. 12, no. 6, pp. 80-87, November 2010.

8 Appendix

8.1 Hashing Function for Pointer Table

The following Verilog code is the actual code that was implemented in the hashFunction module of the system and is based upon the hashing function developed for this application by Cooper Clauson. The hash function accepts a seed and returns a hashed seed. The hashed seed is computed in a set of interleaved XOR and permutation stages. This combinational logic is reduced by the synthesis tools to create a Boolean equation for each of the bits of the hashed seed, based on bits of the original seed. The final hashed seed is simply the value of the final permutation stage.

```
// PARAMETERS
parameter   SEED_BITS=44,
            XOR_STAGES=6,
            POS_BITS=6;

// xorStage[0] is just the current seed being hashed
reg [SEED_BITS-1:0] xorStage [0:XOR_STAGES-1];

// one more permutation than XOR
wire [SEED_BITS-1:0] permute [0:XOR_STAGES];

// if a seed is: AGCT, then the A is associated with the MS bits of
//  the seed
assign permute[0] = {xorStage[0][0],
                     xorStage[0][20],
                     xorStage[0][43],
                     xorStage[0][36],
                     xorStage[0][40],
                     xorStage[0][18],
                     xorStage[0][13],
                     xorStage[0][6],
                     xorStage[0][2],
                     xorStage[0][8],
                     xorStage[0][29],
                     xorStage[0][19],
                     xorStage[0][28],
                     xorStage[0][34],
                     xorStage[0][27],
                     xorStage[0][35],
                     xorStage[0][9],
                     xorStage[0][32],
                     xorStage[0][10],
                     xorStage[0][3],
                     xorStage[0][5],
```

```
                        xorStage[0][21],
                        xorStage[0][30],
                        xorStage[0][23],
                        xorStage[0][37],
                        xorStage[0][39],
                        xorStage[0][11],
                        xorStage[0][1],
                        xorStage[0][24],
                        xorStage[0][25],
                        xorStage[0][7],
                        xorStage[0][4],
                        xorStage[0][14],
                        xorStage[0][12],
                        xorStage[0][38],
                        xorStage[0][17],
                        xorStage[0][42],
                        xorStage[0][31],
                        xorStage[0][15],
                        xorStage[0][26],
                        xorStage[0][41],
                        xorStage[0][16],
                        xorStage[0][33],
                        xorStage[0][22]};
assign permute[1] = {xorStage[1][10],
                        xorStage[1][4],
                        xorStage[1][38],
                        xorStage[1][35],
                        xorStage[1][15],
                        xorStage[1][20],
                        xorStage[1][12],
                        xorStage[1][28],
                        xorStage[1][31],
                        xorStage[1][18],
                        xorStage[1][7],
                        xorStage[1][42],
                        xorStage[1][0],
                        xorStage[1][14],
                        xorStage[1][9],
                        xorStage[1][19],
                        xorStage[1][30],
                        xorStage[1][22],
                        xorStage[1][43],
                        xorStage[1][33],
                        xorStage[1][16],
                        xorStage[1][5],
                        xorStage[1][41],
                        xorStage[1][17],
                        xorStage[1][25],
                        xorStage[1][11],
                        xorStage[1][2],
                        xorStage[1][37],
                        xorStage[1][24],
```

```
                        xorStage[1][13],
                        xorStage[1][23],
                        xorStage[1][32],
                        xorStage[1][21],
                        xorStage[1][39],
                        xorStage[1][6],
                        xorStage[1][27],
                        xorStage[1][8],
                        xorStage[1][3],
                        xorStage[1][40],
                        xorStage[1][1],
                        xorStage[1][26],
                        xorStage[1][29],
                        xorStage[1][34],
                        xorStage[1][36]};
assign permute[2] = {xorStage[2][8],
                        xorStage[2][39],
                        xorStage[2][20],
                        xorStage[2][25],
                        xorStage[2][4],
                        xorStage[2][43],
                        xorStage[2][14],
                        xorStage[2][35],
                        xorStage[2][30],
                        xorStage[2][33],
                        xorStage[2][38],
                        xorStage[2][26],
                        xorStage[2][19],
                        xorStage[2][11],
                        xorStage[2][27],
                        xorStage[2][16],
                        xorStage[2][13],
                        xorStage[2][21],
                        xorStage[2][41],
                        xorStage[2][28],
                        xorStage[2][31],
                        xorStage[2][40],
                        xorStage[2][7],
                        xorStage[2][2],
                        xorStage[2][32],
                        xorStage[2][22],
                        xorStage[2][12],
                        xorStage[2][1],
                        xorStage[2][9],
                        xorStage[2][42],
                        xorStage[2][34],
                        xorStage[2][29],
                        xorStage[2][0],
                        xorStage[2][15],
                        xorStage[2][37],
                        xorStage[2][24],
                        xorStage[2][3],
```

```
                        xorStage[2][5],
                        xorStage[2][17],
                        xorStage[2][36],
                        xorStage[2][18],
                        xorStage[2][10],
                        xorStage[2][23],
                        xorStage[2][6]};
assign permute[3] = {xorStage[3][30],
                        xorStage[3][0],
                        xorStage[3][36],
                        xorStage[3][33],
                        xorStage[3][6],
                        xorStage[3][31],
                        xorStage[3][10],
                        xorStage[3][17],
                        xorStage[3][22],
                        xorStage[3][34],
                        xorStage[3][41],
                        xorStage[3][2],
                        xorStage[3][13],
                        xorStage[3][3],
                        xorStage[3][15],
                        xorStage[3][40],
                        xorStage[3][43],
                        xorStage[3][35],
                        xorStage[3][26],
                        xorStage[3][16],
                        xorStage[3][12],
                        xorStage[3][18],
                        xorStage[3][27],
                        xorStage[3][32],
                        xorStage[3][24],
                        xorStage[3][25],
                        xorStage[3][29],
                        xorStage[3][42],
                        xorStage[3][20],
                        xorStage[3][23],
                        xorStage[3][14],
                        xorStage[3][37],
                        xorStage[3][11],
                        xorStage[3][38],
                        xorStage[3][5],
                        xorStage[3][39],
                        xorStage[3][1],
                        xorStage[3][19],
                        xorStage[3][21],
                        xorStage[3][8],
                        xorStage[3][4],
                        xorStage[3][9],
                        xorStage[3][28],
                        xorStage[3][7]};
assign permute[4] = {xorStage[4][0],
```

```
                        xorStage[4][36],
                        xorStage[4][22],
                        xorStage[4][42],
                        xorStage[4][11],
                        xorStage[4][4],
                        xorStage[4][13],
                        xorStage[4][12],
                        xorStage[4][14],
                        xorStage[4][15],
                        xorStage[4][37],
                        xorStage[4][28],
                        xorStage[4][33],
                        xorStage[4][29],
                        xorStage[4][40],
                        xorStage[4][5],
                        xorStage[4][19],
                        xorStage[4][23],
                        xorStage[4][31],
                        xorStage[4][25],
                        xorStage[4][32],
                        xorStage[4][3],
                        xorStage[4][39],
                        xorStage[4][35],
                        xorStage[4][38],
                        xorStage[4][18],
                        xorStage[4][10],
                        xorStage[4][21],
                        xorStage[4][27],
                        xorStage[4][1],
                        xorStage[4][6],
                        xorStage[4][16],
                        xorStage[4][17],
                        xorStage[4][34],
                        xorStage[4][41],
                        xorStage[4][24],
                        xorStage[4][9],
                        xorStage[4][26],
                        xorStage[4][30],
                        xorStage[4][43],
                        xorStage[4][7],
                        xorStage[4][20],
                        xorStage[4][8],
                        xorStage[4][2]};
assign permute[5] = {xorStage[5][0],
                        xorStage[5][21],
                        xorStage[5][14],
                        xorStage[5][16],
                        xorStage[5][43],
                        xorStage[5][37],
                        xorStage[5][8],
                        xorStage[5][5],
                        xorStage[5][41],
```

```
                        xorStage[5][22],
                        xorStage[5][12],
                        xorStage[5][38],
                        xorStage[5][31],
                        xorStage[5][32],
                        xorStage[5][13],
                        xorStage[5][15],
                        xorStage[5][23],
                        xorStage[5][11],
                        xorStage[5][40],
                        xorStage[5][3],
                        xorStage[5][18],
                        xorStage[5][27],
                        xorStage[5][10],
                        xorStage[5][35],
                        xorStage[5][2],
                        xorStage[5][29],
                        xorStage[5][24],
                        xorStage[5][28],
                        xorStage[5][9],
                        xorStage[5][6],
                        xorStage[5][30],
                        xorStage[5][25],
                        xorStage[5][4],
                        xorStage[5][1],
                        xorStage[5][17],
                        xorStage[5][42],
                        xorStage[5][26],
                        xorStage[5][39],
                        xorStage[5][20],
                        xorStage[5][19],
                        xorStage[5][34],
                        xorStage[5][36],
                        xorStage[5][7],
                        xorStage[5][33]};
assign permute[6] = {xorStage[6][18],
                        xorStage[6][35],
                        xorStage[6][11],
                        xorStage[6][25],
                        xorStage[6][8],
                        xorStage[6][37],
                        xorStage[6][6],
                        xorStage[6][27],
                        xorStage[6][39],
                        xorStage[6][30],
                        xorStage[6][28],
                        xorStage[6][32],
                        xorStage[6][42],
                        xorStage[6][14],
                        xorStage[6][9],
                        xorStage[6][17],
                        xorStage[6][23],
```

```
                        xorStage[6][34],
                        xorStage[6][40],
                        xorStage[6][10],
                        xorStage[6][7],
                        xorStage[6][16],
                        xorStage[6][33],
                        xorStage[6][13],
                        xorStage[6][22],
                        xorStage[6][43],
                        xorStage[6][36],
                        xorStage[6][2],
                        xorStage[6][4],
                        xorStage[6][5],
                        xorStage[6][20],
                        xorStage[6][15],
                        xorStage[6][31],
                        xorStage[6][3],
                        xorStage[6][21],
                        xorStage[6][12],
                        xorStage[6][24],
                        xorStage[6][1],
                        xorStage[6][0],
                        xorStage[6][41],
                        xorStage[6][26],
                        xorStage[6][19],
                        xorStage[6][38],
                        xorStage[6][29]};
// XOR the MS half of the previous permute stage with the LS
//  half of the previous permute stage and store in the LS
//  half of the current XOR stage
always @ (*) begin
      xorStage[0] = currentSeed;
      for (i=1; i<=XOR_STAGES; i=i+1) begin
            xorStage[i] = {permute[i-1][SEED_BITS-1:SEED_BITS>>1],
                        permute[i-1][SEED_BITS-1:SEED_BITS>>1]^
                        permute[i-1][(SEED_BITS>>1)-1:0]};
      end
end
assign hashedSeed = permute[XOR_STAGES];
```