FPGA Accelerated Bioinformatics: Alignment, Classification, Homology and Counting

Nathaniel McVicar

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington

2018

Reading Committee:

Scott Hauck, Chair

Sreeram Kannan

Walter L. Ruzzo

Program Authorized to Offer Degree:

Electrical Engineering

University of Washington

Abstract

FPGA Accelerated Bioinformatics: Alignment, Classification, Homology and Counting

Nathaniel McVicar

Chair of the Supervisory Committee:
Professor Scott Hauck
Electrical Engineering

Advances in next-generation sequencing technology have led to increases in genomic data production by sequencing machines that outpace Moore's law. This trend has reached the point where the time and money spent processing human and other genome sequence data could be greater than that spent producing it. Field-Programmable Gate Arrays (FPGAs) can provide a solution to this problem. Bioinformatics accelerators running on FPGAs achieve order of magnitude speedups across a variety of genomics applications important in both biological research and clinical medicine.

This dissertation presents three accelerators. The first addresses the short read alignment problem, where millions of short DNA or RNA reads, with lengths on the order of 100 base pairs, are aligned to an index built from a reference genome. Our aligner combines an FPGA accelerator with the greater memory bandwidth of our host system to produce a fast and flexible short read aligner. Using this aligner, we developed a classifier to determine which of two possible species each read originated from. In a case study with RNA-Seq reads from mouse and human retinal cultures our aligner produced more accurate classification results and better performance than software-based aligners.

Our second accelerator tackles the problem of non-coding RNA (ncRNA) homology search. The biologically important functions these ncRNAs perform are determined by their two- or three-dimensional structure, and ncRNAs with different sequences can perform the same functions if they share a similar structure. Homology search scores sequences using models of ncRNA families in an effort to find previously unknown members. Our accelerator greatly improves the speed of filters that identify candidate sequences using the Viterbi and CYK algorithms.

The final accelerator uses the Hybrid Memory Cube (HMC), a stacked DRAM, for K-mer counting. In many areas of bioinformatics, including de novo assembly, K-mer counting is a filter with important roles including removing read errors. Our approach stores K-mer counts in a Bloom filter on the HMC leveraging the greater random access rate for increased performance over both the host and FPGA-attached DRAM. Throughout this dissertation we demonstrate that FPGA accelerators can achieve excellent speedups across a variety of genomics applications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

So many people helped this thesis come into being, directly and indirectly. I will never be able to thank them all on these few pages, and nobody's absence should be taken as a minimization of their contribution. Hopefully, the following paragraphs can at least summarize my gratitude.

First and foremost, I want to thank my advisor Scott Hauck. Scott has guided me through so many academic and life transitions, and I am so thankful for his investment and belief in me and my work, which is, of course, his work too.

The support of my family has also been critical throughout my time in graduate school. I am incredibly grateful to my wife, for everything. None of this would have been possible, literally, without my parents, and I don't think my path would have been the same without my grandfather. From start to end, my sister has been wonderful support for me and my family. Finally, as my children grow so does my understanding of their contribution to my spirit.

I also wouldn't be here without the insight of the WUSTL professors who started me in this direction and helped clarify things at difficult junctures. I'd particularly like to thank Roger Chamberlain, Ron Cytron, Sally Goldman, Ron Indeck and Robert Pless.

Although my research journey through graduate school has been more solitary than most, the start came with the Mosaic team including Carl Ebeling, Stephen Friedman, Adam Knight, Robin Panda, Brian Van Essen, Aaron Wood and Ben Ylvisaker. I couldn't have asked for a more supportive group or one more willing to give their time to a new student. I also want to thank the many students I have had the honor of having as labmates in the ACME lab, whose friendship and support I appreciate tremendously.

None of my bioinformatics work would be possible without the collaboration and advice of Larry Ruzzo. I would also like to thank everyone who worked on the projects that my systems were based on, particularly Daniel Jones, Corey Olson, Maria Kim and Zasha Weinberg.

I am so appreciative of everyone in the SAMPA lab for tolerating a hanger-on, attending way more of my practice talks than they had to, and generously sharing their time and perspectives on numerous occasions. Particularly Luis Ceze for many things, including pointing me in Scott’s direction on my very first visit to UW, after assuring me that getting trapped in the PAC elevator is atypical.

It can be hard to remember for those of us who are so fortunate as to be in a position to forget, but graduate school is really expensive. I am grateful to the sources of financial support for myself and my projects, the Egtvedt Endowed Engineering Fellowship, the National Science Foundation Graduate Research Fellowship Program and Pico Computing of Micron Technology, Inc (particularly Mark Hur).

Lastly, some passages of this thesis have been quoted verbatim from the works mentioned at the start of each chapter, and I would like to thank my coauthors, Akina Hoshino, Anna La Torre, Rick Lin and Tom Reh.

DEDICATION

For Elzena. You’ve been supremely patient.

Chapter 1. INTRODUCTION

High-speed sequencing machines are revolutionizing many areas of biological and medical science. With the ability to quickly and cheaply sequence almost any biological organism, they create huge masses of data to help guide research and medical treatment. However, these tremendous increases in data production create an equally tremendous computational bottleneck. We must somehow convert the short DNA snippets produced by the sequencers to complete catalogs of the organisms' DNA strands. In this thesis we consider novel approaches to harnessing the computational abilities of reconfigurable hardware to accelerate several important computations on DNA and RNA data. These include: quickly aligning the short read output of modern high-speed sequencing machines to reference genomes, in order to recreate the source genome's overall sequence; automatically identifying non-coding RNA molecules from different species that perform the same biological function; exploring the application of advanced high-density memories to the problems of sequence filtering and reconstruction.

Since this research involves the application of advanced hardware resources to problems in bioinformatics, we first need to cover the essential background in these fields. This chapter begins by reviewing Field Programmable Gate Array (FPGA) technology. FPGAs are electronic chips that can be programmed and reprogrammed to implement high-performance hardware for specific applications. These chips form the computing substrate on which we will achieve orders of magnitude speedups when compared to standard software approaches. We will also discuss the Hybrid Memory Cube (HMC), a high-density, high-performance memory technology that holds significant promise for some random access tasks inherent in genome processing.

We then shift gears from electronics to biology. To help put our work in context, we first consider DNA itself, both in its structure and how it is processed to produce RNA. These RNA can serve in a functional role, often determined by the structure of the non-coding RNA (ncRNA), or as an intermediate stage before translation into proteins (messenger RNA or mRNA). These mRNA contain optional coding regions, and therefore the same DNA strand can code for multiple proteins. Finding the occurrence of these different isoforms will be one application of the research contained in this thesis. We also consider the DNA sequencing process itself – by understanding what kinds of data is produced by modern sequencing machines, we can better understand the requirements on the processing algorithms that follow.

1.1 FIELD PROGRAMMABLE GATE ARRAYS

FPGAs are among the most adopted reconfigurable logic devices. These devices can be programmed to implement a wide variety of functions, including the equivalent of an application-specific integrated circuit (ASIC) with tens of millions of gates and millions of flip-flops [Xilinx12]. In addition to these basic elements, modern FPGAs include dedicated resources to implement clock management, integer arithmetic, I/O translation (including Ethernet, PCIe and memories), large memory blocks and even small processors. Manufacturers also produce more specialized FPGAs, such as Xilinx's RFSoC and MPSoC devices containing RF-DACs, error correction encoders, GPUs and dedicated video decoding hardware.

FPGA logic functions are implemented, at an abstract level, as an array of Look-Up Tables (LUTs) connected with a programmable interconnect network. By writing to the configuration memory of the LUTs and network, FPGAs can implement logic functions limited only by device size and propagation delay. This simplified view appears in Figure 1.1.

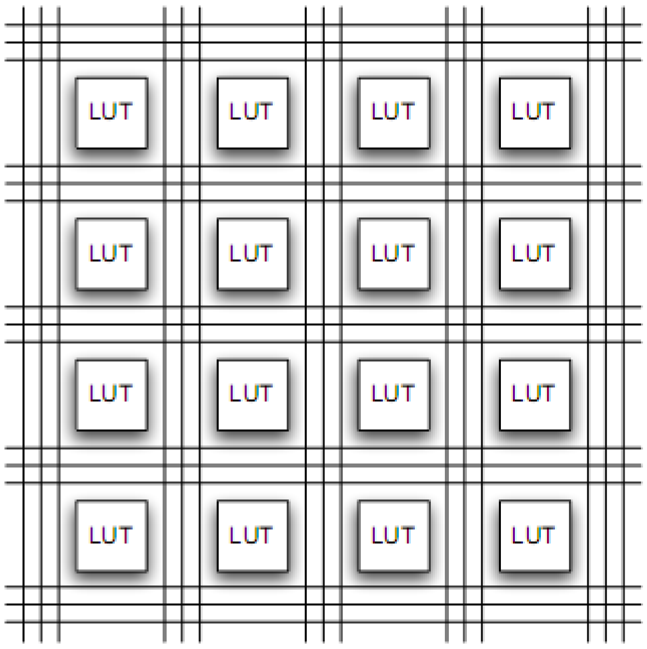


Figure 1.1 Abstract FPGA architecture

FPGAs are popular devices comprising an estimated market of $4 billion in 2010 [Manners10], an estimated $5.6 billion by 2014 [Research11] and $6.4 billion in 2015 [Research16]. Part of the popularity of FPGAs stems from the fact that they are powerful devices that can often come within an order of magnitude of ASIC performance, without the high initial cost [Kuon07]. The majority of their market is telecom, industrial, automotive and aerospace. High performance computing and data processing is a small but growing component, with a roughly 10% market share in 2015 [Research16].

1.1.1 *FPGAs as Accelerators*

Although it has never been the dominant force in the market, FPGAs have been used for computing since their early history [Hauck08]. Early applications included cryptography, multiplication, statistical simulations and computer vision; bioinformatics was also a popular application from early on [Gokhale91][Hoang92].

Although a huge variety of FPGA accelerators exist, they generally share a few properties. First, they are relatively close to a host machine, but have their own attached memory. This can take forms including the more traditional, such as the Pico Computing board [Micron Technologies17] used in our research, as well as the more exotic, such as the DRC RPU110 [Storaasli07] which slotted an FPGA directly into a HyperTransport CPU Socket. Other possible configurations include FPGAs directly attached to biomedical devices, such as the proposed base calling FPGA in MinION [Walter17], and much larger systems that use FPGAs in multiple roles such as the Covey Computer HC-1 [Bakos10] which contains dozens of FPGAs to manage compute tasks, communication with the host, and memory access.

An example of a traditional FPGA accelerator card is the Pico Computing M-503 [Pico Computing, Inc.14]. This daughtercard, shown in Figure 1.2, contains a large Xilinx Virtex-6 FPGA, 8 GB of DDR3 DRAM and 21 MB of QDRII SRAM. Communication with the host is provided by a x8 PCIe Gen2 interface. This card exhibits many of the features common in FPGA accelerators. Fast connections to local memory are provided, as these are important for many acceleration tasks. The host is also close by, and round-trip communication is not prohibitively high-latency. Finally, the FPGA is large enough for a variety of acceleration tasks, with considerable FPGA IO dedicated to the various connections described previously.

Figure 1.2 Pico Computer M-503 Module from [Pico Computing, Inc.14].

One challenge of using FPGAs as accelerators has been the cost and complexity of deploying the actual hardware – FPGA boards for computation can cost in the tens of thousands of dollars (or more for a large system like the Convey HC-1, described above), and can be complex to install and maintain. In recent years cloud service providers have started including FPGAs into the servers themselves, an approach pioneered by Microsoft in its Catapult project [Putnam14]. This trend has continued with Amazon's F1 instances [Amazon Web Services, Inc.16] and Microsoft's Azure Configurable Cloud nodes [Caulfield16]. In these systems FPGAs are becoming a commodity element of the cloud computing infrastructure, one that brings the promise of hardware-accelerated speed to a much wider audience.

There have been also been a number of different projects in recent years that have harnessed FPGA accelerated computation for bioinformatics applications; these will be covered in section 1.3.

1.1.2 *FPGA Attached Memories*

For many FPGA accelerators, the attached memory system is simply DRAM or SRAM, as described above. For example, Amazon's F1 instances have DRAM and SSD storage on the host and DRAM attached to the FPGA. On the other hand, some accelerator boards include significantly more memory options, with DRAM, SRAM and more exotic memories like HMCs.

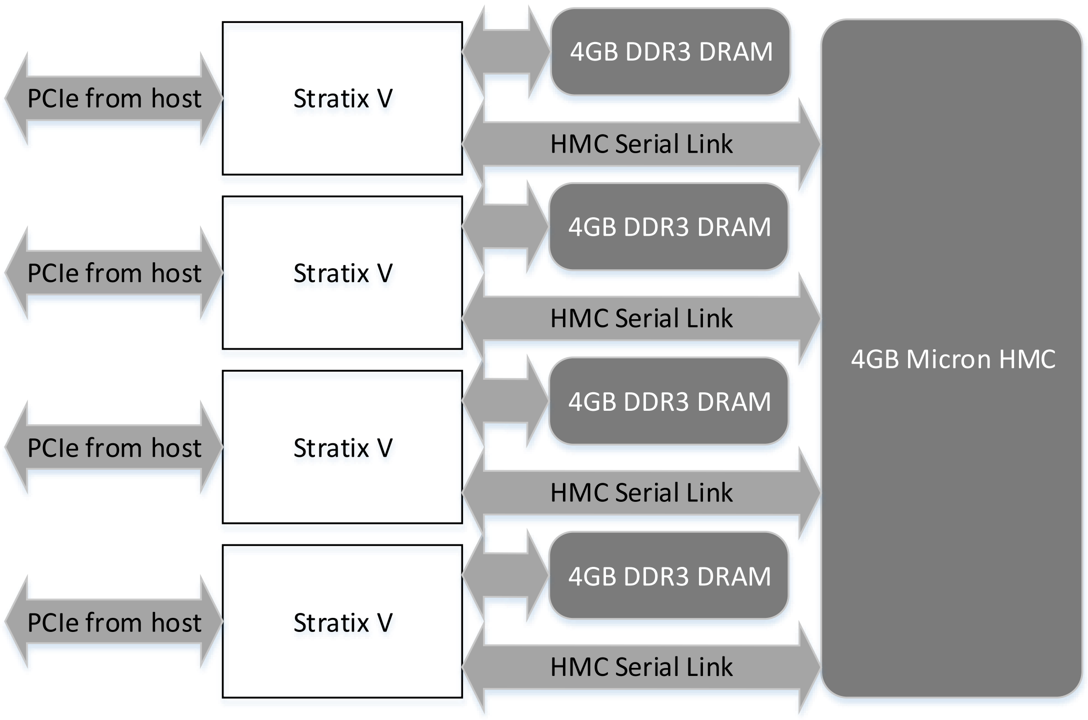


Figure 1.3 Micron SB-801 block diagram

1.1.2.1 The Hybrid Memory Cube

Some algorithms require a great deal of memory bandwidth, with either sequential or non-sequential (random) access patterns. Although DRAM sequential access performance continues to improve with every new standard and process generation, the ratio between DRAM sequential bandwidth and random access Operations Per Second (OPS) continues to worsen. DRAM bandwidth (and IO in general) is also scaling more slowly than computational resources on FPGAs.

This creates an opportunity for accelerators to improve performance using non-traditional memories. Instead of gaining performance compared to the software implementation only by accelerating the computation, an FPGA tightly coupled to a faster memory can also take advantage of increased memory bandwidth and operation rate. This observation has led to efforts to tightly couple FPGAs with these non-traditional memories [Ramalingam16][Gokhale15], including through shared packaging.

In particular, some of our work uses the Micron Hybrid Memory Cube (HMC) [Micron Technology15]. The HMC is a memory chip built using stacked DRAM to provide high bandwidth and, most importantly, very high random access performance compared to traditional DRAM configurations. Instead of the wide parallel interface provided by a DIMM, the HMC has up to sixteen 20 Gbps serial interface channels. On a multi-FPGA board, different links can be connected to different FPGAs, for example four FPGAs each with four serial links to a shared HMC, as in the Micron SB-801 (Figure 1.3) used in our work.

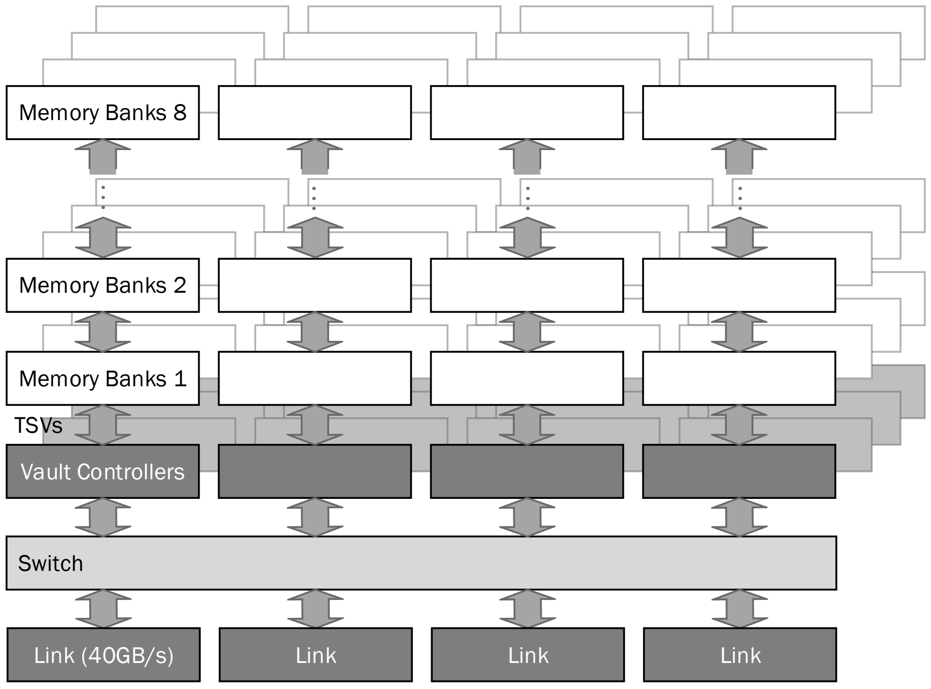


Figure 1.4 Micron HMC architecture block diagram

Other HMC configurations are also possible. In addition to those already discussed, an FPGA and HBM sharing a package, as with the Stratix 10 MX, and many FPGAs connected to a single HMC, as with the SB-801, it is possible for a single FPGA to use many of the HMCs external links. This is the case with the Micron AC-510, where a single FPGA is connected to an HMC over two separate links. Finally, it is possible for a single or multiple FPGAs to be connected to multiple HMCs, although further chained HMCs will have increased access latency [Pawlowski11].

1.1.2.1.1 HMC Performance Considerations

The HMC gets its performance advantage over traditional DRAM by having many banks that can be accessed in parallel by the VCs, as shown in Figure 1.4. The controllers themselves are designed to handle a large number of transactions in flight, servicing them quickly as banks become available. This architecture allows for both very high bandwidth and improved random access performance when compared to traditional DRAM.

There are some potential downsides to the architecture. First, if the HMC receives too many operations to the same small set of banks, it can slow down significantly since little exploitable parallelism is available. Although internal buffering and reordering can mitigate this to some degree, a sufficiently pathological access pattern can greatly degrade HMC performance. The second issue is the reordering itself. Because of its more complex architecture, the HMC reorders operations at a number of points, including between the VCs and within each VC. The only ordering that is guaranteed to be maintained is that among operations to an individual bank. Finally, the HMC crossbar privileges a set of VCs for each link. Accessing memory only through the privileged links will result in double the performance of accessing non-privileged memory across all links simultaneously, as our work demonstrates below. However, this has the obvious downside of effectively dividing the HMC into four separate address spaces, instead of appearing as a uniform shared memory to all linked devices. Even when accessed to achieve the greatest possible bandwidth, the many control layers required by the HMC increase latency to roughly double that of DRAM. This and other performance details, including atomic operations, are also described in Chapter 4.

1.2 BIOINFORMATICS

Bioinformatics is the use of computation to answer questions about biological data. Often these questions require the analysis of the sequence and structure of biologically important molecules like DNA and proteins. The explosion in the volume of available biological data, discussed previously, has put an even greater focus on the performance of bioinformatics applications.

1.2.1 *Motivation*

Bioinformatics is a critical component in basic biomedical research. Applications range from medical, such as identifying disease and developing new treatments and therapies, to agricultural, such as developing new high-yield crops and creating targeted pesticides and herbicides. Although bioinformatics applications often include complex pipelines with many stages of analysis and data processing, the early stages of DNA sequence alignment and assembly are key components of many of these pipelines. Recently many academic and open source software packages have attempted to accelerate these processes. These packages can dramatically speed the analysis of genetic information, but they require significant computing power from CPU clusters or cloud computing platforms.

1.2.2 *History*

During the 1990s the Human Genome Project (HGP) created the first draft sequence of the entire human genome [Lander01][Venter01]. This reference sequence provided an initial consensus of the nucleotide, or base, for a “typical” human at every position in each of 24 chromosomes. Each DNA base is one of adenine (A), thymine (T), cytosine (C) or guanine (G), so the reference sequence is essentially a string of length ~3.2 billion in a four character alphabet. The first HGP draft sequence required nine years and almost $3 billion to complete due to the relatively immature sequencing technology used [NHGRI10].

Next-generation sequencing machines were first introduced in the mid-2000s and have made extremely rapid gains, both in terms of speed and cost. In recent years the decrease in NGS cost has actually outpaced Moore’s law [Wetterstrand14]. NGS technology has been hugely influential across a number of fields, including personalized medicine, drug discovery and oncology. While there are a number of different technologies employed in NGS, they are all based on performing many short sequencing operations in parallel. Instead of sequencing an entire chromosome at once (human chromosomes range from ~48 million to ~249 million base pairs [bp]), NGS technology sequences many DNA fragments, ranging from tens to thousands of bp in length, in parallel. These fragments come from replicating the DNA to be sequenced many times and then dividing each copy somewhat randomly, resulting in many overlapping fragments (see Figure 1.5). Because of the parallel nature of this technology, a full human genome can currently be sequenced at a cost of roughly $1k to $10k and in a period ranging from a day to a few days. Due to this trend of declining sequencing costs, the bioinformatic analysis required to gain medical or scientific insight from NGS data becomes an ever-increasing percentage of the total time and expense.

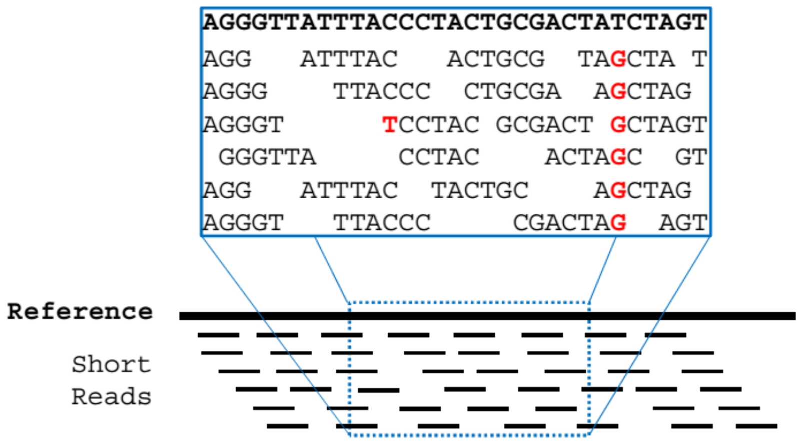

AGGGTTATTTACCCTACTGCGACTATCTAGT
AGG ATTTAC ACTGCG TAGCTA T
AGGG TTACCC CTGCGA AGCTAG
AGGGT TCCTAC GCGACT GCTAGT
GGGTTA CCTAC ACTAGC GT
AGG ATTTAC TACTGC AGCTAG
AGGGT TTACCC CGACTAG AGT
Reference
Short Reads

Figure 1.5 Short reads aligned to a reference gnome

1.2.3 *Biologically Important Macromolecules: Deoxyribonucleic Acid, Ribonucleic Acid and Proteins*

Although they are likely sufficiently familiar to most readers that previous discussions in this work did not include any lengthy explanations, it will be helpful going forward to describe some important molecules in more detail. In particular, this section will contain a few details about deoxyribonucleic acid (DNA), ribonucleic acid (RNA) and protein.

DNA, which encodes the genetic information required for life, is made up of a chain of nucleotides. These chains, or polynucleotides, consist of a sequence made from an alphabet of four distinct organic molecules, typically represented by the letters C, G, A and T. Each of these bases forms a hydrogen bond with another base (A to T and C to G), and the bonding of these two polynucleotides creates the classic double helix structure of DNA. Although DNA has many important chemical properties and interactions, such as its durability and relations with functional molecules inside of cells for repair and replication, a full discussion of these is beyond the scope of this work.

What is significant for our work is that DNA mostly performs its function by encoding for other molecules. These are RNA and protein. First, the RNA polymerase separates the two halves of a DNA helix and runs along one polynucleotide using pairing to create an exact copy of the other polynucleotide (with the exception that a uracil [U] nucleotide is used in place of thymine). This process is called transcription, and occurs at specific regions known as promoters (Figure 1.6). A complex regulatory system controls which RNA is created and in what quantities, to maintain healthy operation of the organism. Some of the RNAs that are transcribed go on to perform important functions of their own. These are known as functional or non-coding RNAs, and are discussed in more detail in Chapter 3.

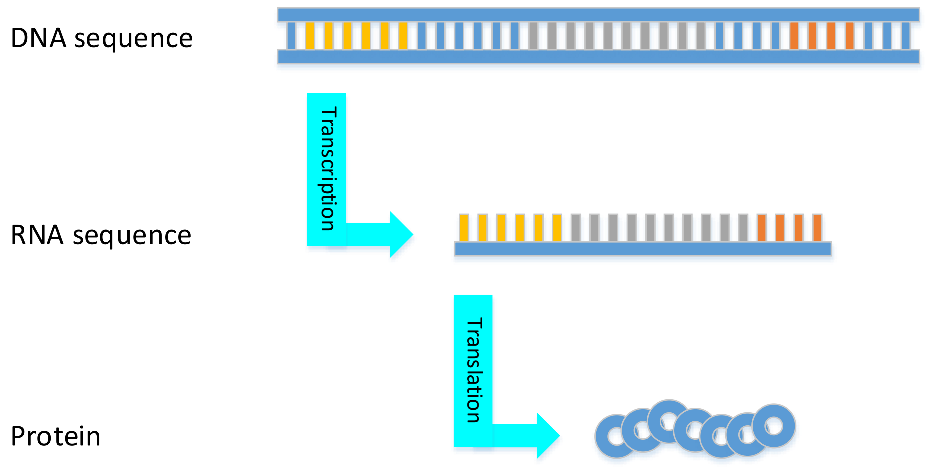


Figure 1.6 The central dogma of molecular biology

The more well-known RNA is the messenger RNA (mRNA). This RNA encodes a gene which is converted to protein through a process known as translation. The mRNA is made up of 3-base groups, called codons, and each codon represents an amino acid. During translation, a functional RNA known as a transport RNA (tRNA) binds with an amino acid on one end and a codon on the other to construct large protein molecules out of an amino acid chain. This process can be repeated to create many proteins from a single mRNA and is orchestrated by a ribosome. A ribosome itself is a complex molecule made up of both RNA and multiple proteins.

The role of the ribosome is just one of the myriad functions that proteins perform in cells. A typical bacteria cell, such as E. coli, contains roughly two million proteins, while a yeast cell may contain fifty million proteins and a human cell one to two billion [Milo13]. Through their complex three-dimensional structure and ability to bind to many other types of molecules, proteins carry out many of the important functions within and between cells, as well as regulating the creation of other ncRNAs and proteins depending on the surrounding environment and cell type. Many different proteins are required to carry out this diverse array of functions. In fact, it turns out that there are many more proteins than there are separate protein coding regions or genes. Humans have roughly twenty thousand genes, but more than one hundred thousand different proteins, with some estimates as high as one million. There are a number of mechanisms that make this possible, and one of those is alternative splicing.

1.2.3.1 Exons, Introns and Isoforms

In the case of alternative splicing, a single gene can make many different proteins. This is possible because the entire DNA sequence is not transcribed to mRNA. Instead, only specific regions of the sequence, known as exons, actually become mRNA. First, the entire gene sequence creates a precursor mRNA transcript. Next, through a process known as splicing, non-coding regions called introns are removed, resulting in the final mRNA. There are a number of mechanisms for splicing, include an RNA and protein hybrid molecule known as the spliceosome in eukaryotes.

In addition to removing intronic regions of DNA that do not code for proteins, splicing is used as a regulatory mechanism where one gene can code for many different proteins. This is possible because exons can alternatively be included or excluded from the final mRNA. The resulting proteins, which are created from the same gene but have different sequences and functions, are known as protein isoforms. An example of possible isoforms for a DNA sequence with three exons is shown in Figure 1.7. Note that while alternative splicing is very powerful, and occurs in the case of most human genes [Pan08], not every possible isoform exists in practice. This means that while there are many isoforms for a given gene, there are typically less than the maximum possible 2^{exon}. Additionally, the possible number is not much higher because alternative splicing does not reorder exons. Exonic regions always appear in the same order in the gene and the final mRNAs.

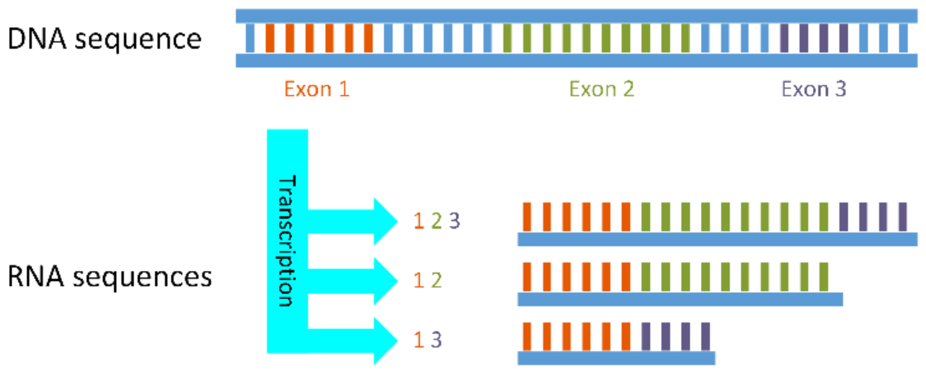


Figure 1.7 Possible alternative splicing for a DNA sequence with 3 exons

1.2.3.2 Reverse Complements

Earlier, we noted that the RNA polymerase only interacts with one DNA strand; however, this may be either strand. In either case, the polymerase begins at what is designated the 3’-end of the strand and proceeds towards the 5’-end. Since the two DNA strands are bonded, these are opposite directions and because the polymerase is creating a complimentary copy, it creates the RNA from the 5’-end to the 3’-end.

These complimentary copies are known as reverse compliments, and two reverse complement strands of DNA or RNA will bond to each other. For example, a strand with bases “ACC” has a reverse complement, and would bond to, a strand with bases “GGT”. This is discussed in more detail in Chapter 2, as it relates to our work, but the concept is important to be aware of when considering genome sequencing in the following section.

1.2.4 *Genome Sequencing*

As described previously, next-generation sequencing machines, such as the Illumina HiSeq 3000 shown in Figure 1.8, have greatly reduced the cost and increased the speed of sequencing an organism's genome. What previously took weeks or days can be done in hours. One of the greatest challenges facing modern sequencing is the time and computational resources required to processes the massive amount of data these machines are able to produce.

Figure 1.8 Illumina HiSeq 3000 Sequencer [Illumina, Inc. 2016]

This computational problem is nontrivial for a number of reasons, and in the case of short reads extracting meaning from whole-genome sequencing results is particularly challenging.

1.2.4.1 Short Reads

Previously, we outlined the properties of NGS machines and highlighted the parallelism they use to achieve high throughput. It is important to note that this parallelism comes at a cost. Recall that short read based NGS technology sequences small DNA fragments of 50 to a few thousand bps.

These fragments come from replicating the DNA to be sequenced many times and then dividing each copy somewhat randomly into fragments, which will overlap each other as seen in Figure 1.5. This introduces significant computational challenges, discussed below. There are also tremendous benefits. Fast and cheap whole genome sequencing opens up new avenues of medicine and therapies as well as new avenues of pure genetics research [Taber14].

Among the computational steps needed to make sense of NGS data, one of the most significant is alignment. NGS machines are often used for whole-genome sequencing. In these cases, the user typically wants to know something about how the sequenced individual differs from a known reference genome. NGS introduces significant computational challenges to the relatively simple problem of genome alignment. First, as shown in Figure 1.5, the short reads may contain both errors in an individual read (introduced by the sequencer, the replication process, or another source) and actual differences between the individual sequenced and the reference genome. Additionally, because the short DNA fragments are not sequenced in any particular order, the short reads arrive from the sequencer without any information about the part of the genome to which they align. Potential errors and differences from the reference cannot be determined without first knowing this alignment, so a process called short read alignment is required.

Through alignment, each short read is assigned zero, one or more potential alignments, as well as a score attempting to quantify how likely that alignment is to be correct. Given the errors introduced in the sequencing processes and the extremely repetitive nature of some portions of the genome, the accuracy of these alignments varies significantly. Short read alignment is typically one of the first steps of a long genomics pipeline, which must potentially process a few billion reads for one individual, so quickly achieving accurate alignments with good information about when an alignment is unreliable is critical to the performance of the pipeline. Because the cases where an individual's genome differs from the reference are often the significant ones, the alignment algorithm must be able to tolerate some difference between the short read and the reference. Later pipeline stages typically evaluate all of the reads aligned to a particular reference position to determine which reads contain errors and which describe actual genetic variation.

There are many software-based short read aligners, and a survey of all available software aligners is outside the scope of this work. These alignment tools each tend to be optimized for a specific NGS technology. Every NGS company produces reads with differences in their error rates and distributions as well as read lengths. For example, our work is with Illumina reads which tend to have relatively few errors, and these cluster towards the ends of the reads. Illumina is one of the least expensive, and therefore most common, sequencing technologies and produces reads from 50 to a few hundred bp long. Because of the low cost per base it is also possible to get very good coverage with Illumina sequencers. On the other end of the spectrum is Pacific Biosciences (PacBio) which produces very long reads (10,000 bp and up) with a higher error rate along the entire length of the read. Between these technologies fall many other sequencers such as SOLiD and Ion Torrent. Because of their differences, each sequencing technique has applications for which it is stronger or weaker; however, whole-genome sequencing and variant calling is a fairly broad application for which most of the sequencers are applicable.

In addition to being designed with a specific sequencing technology or category of technologies in mind, most aligners fall into one of two categories. Some aligners, such as BFAST, FASTER and mrFAST use a hashing or seed based approach [Shang14]. Others such as BWA, Bowtie and SOAP use a BWT or suffix array based approach. Finally, some more specialized aligners such as GMAP, use entirely different algorithms appropriate for their specific problem. Each algorithm has its own strengths and weaknesses but in general hashing based algorithms are superior if there is very little repetition in the relevant parts of the reference, while suffix arrays are better for highly repetitive regions. This makes intuitive sense because with the suffix based approach many similar reference positions will be near each other and can all be looked up together. When hashing, having to look up neighboring, but not identical, reference locations is typically no faster than if they had nothing in common.

1.2.4.2 De Novo Assembly

One way that NGS machines are able to produce so much genomic data so quickly is through the use of short reads, as discussed previously. This sequencing technique involves the high-bandwidth reading of huge numbers of small subsequences of the DNA in question, conducting multiple samplings of each region of a DNA strand. This is important, because each individual short read (roughly 100 base pairs for Illumina machines) of the DNA is likely to include some errors due to the underlying chemical and electrical processing. Multiple reads of the same region help resolve these errors [Quail12], but also greatly increase the problem size.

In reference-based assembly, the short reads are individually aligned to a known reference, such as the human genome discussed above. De novo assembly is a more computationally expensive problem [Zerbino08]. In this case no reference is available, so the short reads must be assembled only in relation to each other. This can occur in many biologically important cases, such as a species which has not yet had its reference constructed (currently the vast majority of species on Earth). It can also happen if the source of the reads is not known, for example in a metagenomic study of the life in seawater, dirt or the human gut. In these cases, the reads must be assembled into a coherent sequences in much the same way the first human reference genome was during the Human Genome Project.

1.3 Bioinformatics Accelerators

Custom and reconfigurable chips have a long history of use as bioinformatics accelerators, predating even NGS. For example FPGAs were accelerating homology search as early as 1995 [Lemoine95] and many of the algorithms we currently use, such as Smith-Waterman, were being implemented on Application-Specific Integrated Circuits (ASICs) to solve biology problems even earlier [Chow91]. Other FPGA accelerators for genomics problems include phylogeny reconstruction [Bakos07] and sequence alignment not based on Smith-Waterman [Chamberlain07]. Between 2005 and 2010 the number of accelerators for genomics and other biology applications being implemented on FPGAs expanded, going from a few accelerators all tackling similar problems to dozens across many disciplines.

Depending on definitions, some systems in related areas can also be considered bioinformatics application accelerators. For example, Anton 2 is a large custom supercomputer developed by D.E. Shaw Research [Shaw14]. Anton is designed to simulate large molecular dynamics problems using ASICs, and one of the primary applications is protein folding simulations, which falls under the broad category of analyzing biological data.

In addition to other bioinformatics accelerators, there are many FPGA and GPU based short read alignment accelerators. Our accelerator, discussed below, is based on earlier work from our lab [Olson12]. Other accelerators use entirely different algorithms including banded Needlman-Wunsch [Chen12], a combination of BWT and S-W similar to BWA-SW [Arram13] and even full string comparison against the genome [Knodel11]. A summary of many other aligners, as well as other genomics accelerators, can be found in [Aluru14]. Overall, much like the software algorithms they implement, these accelerators fall into one of two categories. Some, including Olson, execute a dynamic programing string matching algorithm, sometimes proceeded by hashing to filter potential matches. Others use an FM-index. The FM-index is based on the BWT and allows for exact matches to be looked up very quickly. The data structure is not ideal for FPGA acceleration and requires somewhat unpredictable memory accesses of large amounts of table information.

Areas that were particularly amenable were quickly saturated with accelerator designs. For example, many FPGA-based HMMER accelerators were published within a short time period from 2007 to 2010, since the implementation was very straightforward the speedups were impressive. Some examples include [Derrien07], [Oliver08], [Takagi09], [Sun09], [Abbas10], [Derrien10] and [Eusse Giraldo10]. The move to floating point arithmetic with the forward algorithm, discussed in section 3.2.3, removed much of the low hanging fruit for FPGA designs in this area.

Chapter 2. SHORT READ ALIGNMENT AND CLASSIFICATION

Of the computational challenges presented by NGS, one of the most fundamental is finding the position of a read in a reference genome, known as short read alignment. This is challenging due to differences between the reads and the reference introduced by both read errors and true individual variation, as shown in Figure 1.5. Often, the goal of whole-genome sequencing is to find all instances of actual variation, while ignoring read errors, but variation from the reference can't be detected until each read is aligned. This process must be done with next to no guidance about where each read aligns because replicating the individuals DNA and cutting it up typically mixes the entire genome together. This stage of the pipeline is called short read alignment, and simply tags each read with zero, one or more potential reference genome alignments. Later pipeline stages filter out errors, call variants and perform additional tasks. It is this first stage that the work discussed in this chapter accelerates.

In some situations, it is not possible to determine in advance which of a set of references to use for a given read. This can occur in the case of xenografts, where, for example, human cancer cells are placed in a mouse so their behavior can be studied in vivo. This provides a number of advantages for scientists but also the disadvantage that the short reads will be contaminated with some unknown percentage of mouse DNA. Sorting this out is a challenging problem, particularly because some areas of the mouse and human genome are similar or identical. Reads can contain DNA (or RNA) from two or more species for other reasons as well, such as mixing cells from a fast and slow developing species to cause the slower developing cells to accelerate. This is the motivating situation for the work discussed in this chapter. The main results presented in this chapter have been published as [McVicar18].

2.1 Classification of Retinal Co-culture Short Reads

One of the most common and medically important uses of implanted cells is tumor xenografts. In this case, human tumor cells are grown in a host, such as a mouse, enabling detailed studies of human tumors in living hosts that would not otherwise be possible [Bradford13]. However, total RNA (which includes all of the various forms of RNA found in a cell) extracted from the xenografts will contain RNA from human tumor cells as well as contaminating mouse RNA. Thus, the RNA sequencing (RNA-Seq) reads that are obtained need to be classified as relevant human RNA or contaminating mouse RNA. Artificial chimeras, created by researchers, are similar to xenografts. Human or other cells are implanted into a mouse and studied there in ways that would not be possible in a human host [Behringer07]. More recently, human chimeras have been created with larger mammals [Wu17]. Chimeras can be more or less integrated, depending on when during development cells are implanted and whether embryonic stem cells are used.

The read classification problem addressed in this chapter involves identifying the source of simulated co-cultured mouse and human retinal reads. Co-cultures contain cells from two or more different species, but are not performed in vivo as they are in xenografts and chimeras. Co-cultures are also extremely useful in research and present similar alignment problems, where the correct reference for a specific read is not known.

Retinal cell transplants are another significant use of implanted cells. Implanted embryonic stem cells offer a potential therapy for diseases involving retinal degeneration [Lamba06][Karl08][Wilken16]. Among the challenges facing this research is the slow division of human retinal cells and the inability to perform early stem cell injection experiments on humans. One solution to these problems is implanting human derived embryonic stem cells into the subretinal space of newborn mice or non-human primates [Chao17]. When performed correctly, human retinal cells will develop and function within the mice. However, this creates its own challenges, including identifying whether DNA or RNA extracted once the retinas matured came from mouse or human cells. Before attempting these experiments, co-cultures provide a proof of concept, and additional benefits including potentially more rapid division of the human retinal cells. The same read classification problem also arises with co-cultures, and it is this problem that we simulate using reads from mouse and human cultures.

The read data itself is generated using RNA sequencing (RNA-Seq), a technique where Next Generation Sequencing (NGS) technology is used to determine the expression level and splicing of genes, as well as the presence of other RNAs in an organism or chimeric tissue [Pan08]. As opposed to some other methods, RNA-Seq provides a nearly complete view of expressed RNA, but only as short fragments. As with other short read alignment problems without a single known reference species, understanding and use of these short fragments requires information on their relative position that must be gathered through either assembly or alignment to multiple references.

2.2 Short Read Aligners

There are a number of software packages that perform short read alignment. These include BFAST [Homer09], BWA-SW [Li09a], Bowtie2 [Langmead12], GSNAP [Wu10] and SOAP2 [Li09d]. These tools represent a spectrum of tradeoffs between accuracy and performance [Hatem13].

The fundamental problem faced by short read aligners is that most reads have at least one error, with longer reads tending to have more errors or possible individual variations, so exact string matching cannot be used to align short reads to a reference. Algorithms that can find alignments while tolerating errors, such as Smith-Waterman, are much slower, and thus it is typically not practical to use such algorithms to compare each short read to the entire reference genome (over 3 billion nucleotides for humans).

To address this problem, BFAST and other aligners perform alignment in two stages. The first winnowing stage identifies promising candidate loci in the reference sequence. This is done by splitting the short read into smaller sequences of length K and identifying reference locations that are exact matches to these subsequences, called seeds. Exact matching can be performed relatively quickly using an index lookup, resulting in one or more Candidate Alignment Locations (CALs) in the reference for each read. The index is precomputed from the reference and does not have to be regenerated for each run. This approach works because as long as the seeds are short enough relative to the read, some of them should be error- and variation-free and thus match the reference exactly.

The second stage scores the alignment of the entire read against each CAL using a slower but more powerful method that can tolerate some errors and variation while still identifying likely matches (Smith-Waterman in our work). Because of the high level of filtering provided by the first stage, this can be done in a reasonable amount of time despite the additional computation.

2.2.1 *Basic Techniques*

Most aligners approach the first stage in one of two ways. Some aligners hash a seed or seeds from the read. Alternatively, aligners may use a BWT or other suffix array based indices. There are also aligners, such as GSNAP, that use unique algorithms to address more complex or specialized use cases [Wu10]. Although the performance of each approach is situational, suffix based aligners are particularly strong for repetitive references as discussed in 1.2.4.1. Most modern alignment tools make use of multiple filters, including different algorithms, to produce high quality alignments.

2.2.2 *Aligner Overview*

At this point it is helpful to go over a quick example of the operation of the candidate identification and candidate scoring stages for a hashing aligner like ours. This process is depicted in Figure 2.1. With reads of length 100 bp and seeds of length 20 there will be 5 non-overlapping seeds, and some of those are often exact matches to the reference, despite a few differences per read due to errors or individual variation. Most of these length 20 seeds are sufficiently unique that there will be a limited number of CALs that require full alignment for this read. However, depending on the size and properties of the reference, this seed length, and the way the seed is broken down into addresses, may be inappropriate. Considering these issues introduces a number of tradeoffs in the areas of speed, memory and alignment accuracy. Smaller seed lengths will increase sensitivity and error tolerance, but also increase the number of CALs for each seed. Shorter strings repeat more often in the genome. Similarly, overlapping seeds improve error tolerance but also significantly increases the number of seeds that must be examined for each read (in our example there are 5 non-overlapping seeds per read but 81 overlapping seeds.) There are also significant engineering tradeoffs to be made in compactly storing and efficiently accessing the index data structure.

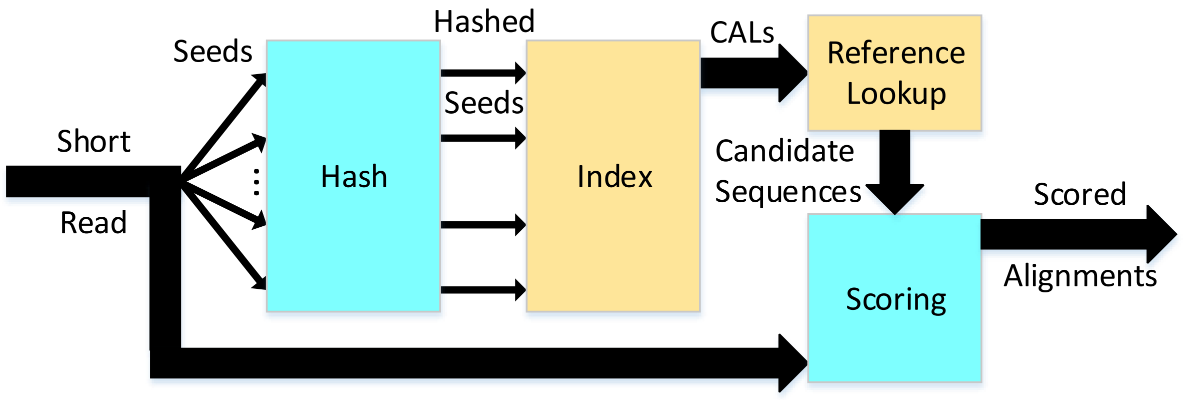


Figure 2.1 Block diagram of generic short read aligner with a hash and index. Blocks using large precomputed data structures are yellow.

Our aligner uses a similar indexing strategy to BFAST, so a more detailed discussion of BFAST's index is helpful to better understand these tradeoffs.

2.2.2.1 BFAST Index

The BFAST index was designed to use large search keys (seeds) to attempt to find as close to a unique genome location as possible (one CAL per read) while still allowing for an O(1) lookup for each key [Homer09]. In order to achieve this, BFAST creates a number of indexes from the reference genome, each using a different (generally not contiguous) subset mask (Figure 2.2). These masks provide gaps that can ignore different SNPs and read errors. Because CALs are only generated from a portion of the read that is an exact match with a masked portion of the genome, using a larger key size results in fewer CALs per read but can also reduce the sensitivity of the matching in the case of read errors or SNPs.

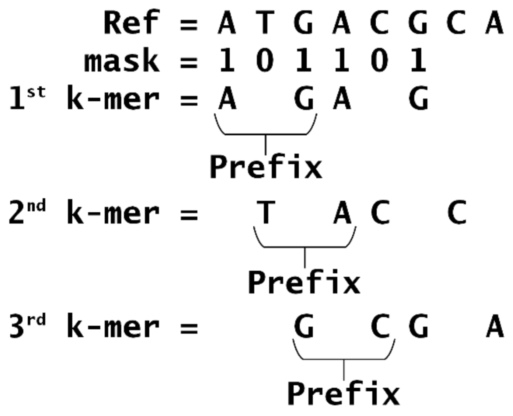


Figure 2.2 Simplified BFAST index construction example, where K = 4 (key size) and J = 2 (prefix size)

In order to get only a few CALs per index per read, BFAST actually uses K = 22 for the human genome (K is the K-mer length, i.e. the length of the substrings selected from the read). Note that for large K's like this, all possible K-mers do not occur in the genome. For this reason, an index of all K-mers would be very sparsely populated and inefficient. Instead, BFAST uses indexes of only K-mers that actually occur in the genome and accesses them using a dense table of the first J bases of the K-mer being looked up, where J < K. For example (see Figure 2.2), for the sequence “ATGACGCA” and the mask 101101 with K = 4 and J = 2, the 2-mer “AG” would be looked up in this dense table. For the human genome and a K of 22, BFAST uses J = 16 so that this table remains manageable at 4G entries. Each entry in the table provides a position in the index for the first and last position of K-mers that start with the J-mer, and because the K-mers are sorted in the index, a binary search can very quickly find the desired K-mer and therefore the relevant CALs. The two stage CAL lookup process described above provides one inspiration for our system but lacks flexibility as discussed below.

2.3 V1 FPGA System

This section will summarize the architecture of the first version (V1) of our hardware accelerated short read aligner, described in detail in [Olson11]. The V1 system, inspired by BFAST, implemented a short read alignment pipeline primarily on FPGAs. This pipeline accepts a stream of short reads and outputs these reads aligned to the reference genome using the following modules: Index Construction, CAL Finder, CAL Filter, Reference Retrieval, Smith-Waterman Aligner, and Score Tracker. I was not involved in the design or development of the V1 system, but a conscience explanation of the index and alignment methods used is important to understand the updated (V2) system. Additionally, the comparison of the V1 to V2 system provides a useful case study in software/FPGA co-design.

2.3.1 *Index Construction*

Index construction in V1 is not an FPGA task, but rather a step that must be done before running FPGA accelerated alignment. This step is performed on a host machine and need only be run once for each reference and seed parameter combination. Thus, the index would generally be precomputed when a user's computation methodology was defined, and used from then on, potentially for weeks or months. The index consists of two tables: the pointer table and the CAL table. These tables together indicate, for a given seed, the location of all occurrences of that seed in the reference genome. For a seed length of 8, there would be an entry for all occurrences of "AAAAAAAA", another for "AAAAAAAC", etc.

In order to construct these tables, the entire reference genome is broken up into overlapping seeds. See Figure 2.3 for an example of this process. In the case of the V1 system, the seed length was configurable, but typically set to 22 bases. Each seed is further subdivided into address and key section, as shown in the figure. The address is used as the index into the pointer table; the length of the address determines the size of the pointer table because the pointer table has one entry for every possible address. Since there are four bases, this works out to $4^{\text{address\_length}}$ entries. Note that there will be address table entries for every possible string of the address length, even those that do not show up in the reference. For this reason, the pointer table is a fixed size for a given seed and key length.

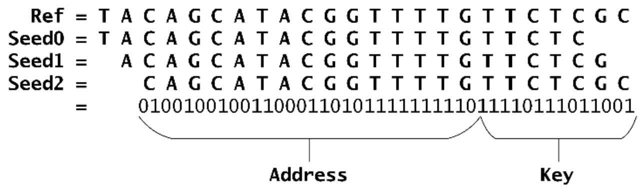

Ref = T A C A G C A T A C G G T T T T G T T C T C G C
Seed0 = T A C A G C A T A C G G T T T T G T T C T C
Seed1 = A C A G C A T A C G G T T T T G T T C T C G
Seed2 = C A G C A T A C G G T T T T G T T C T C G C
= 010010010011000110101111111110111101111011001
Address
Key

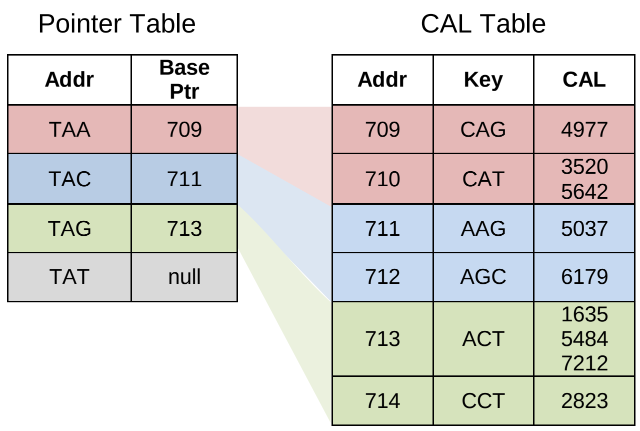
Figure 2.3 The seeds for a reference sequence, with seed 2 divided into address bits and key bits

The V1 system uses a 22 base seed for the same reasons as BFAST. This length provides good specificity, avoiding too many false positives in the form of numerous extraneous CALs for each read, while also being short enough to allow many mismatches per read. At 22 bases for non-overlapping seed, an 88 base read with fewer than 4 errors is guaranteed to have at least one perfectly matching seed and is therefore likely to identify some CALs. However, as with BFAST, the data structure implied by a 22 base seed would be very large ($4^{22} = 1.76 \times 10^{13}$ entries) and mostly empty, since many seeds never occur in a particular reference genome or.

To solve this problem, the V1 system uses two tables. The CAL table contains <key, CAL(s)> pairs. Each of these entries stores all of the one or more CALs associated with a seed actually found in the reference. If the key in the CAL table matches the key bases from the end of the seed (7 nucleotides in this example), that CAL is considered a match. This means the relevant portion of the reference should then be looked up and scored, as discussed later.

Although the CAL table contains CALs for each sequence in the reference, it will not contain entries for all possible sequences. Instead, a second table, called the pointer table, is used to index into the CAL table. The pointer table contains one entry for each possible address where an address is a fixed length prefix of the seed (15 bases in the example from Figure 2.3). Lookups into the pointer table using the address are fast, and the pointer table contains pointers into the CAL table or a null entry if the reference contains no sequences matching the address. The V1 system provides additional features to compress the table and improve lookup speed, and details of these are provided in [Kim11].

To illustrate this, we will use a shortened version of Seed0 from Figure 2.3 to improve clarity. Let the seed be “TACAGC” with a three base address “TAC” and a three base key “AGC”. Using the pointer table found in Figure 2.4, the address “TAC” is found in the second position (blue). Because the pointer table is complete any possible address found in the seed will be in the table, although it may not have any CALs, as is the case with “TAT” (grey). Continuing with the example, the base pointer for Address “TAC” points to position 711 in the CAL table. The CAL Finder walks the CAL Table for a specified distance, looking for matching Keys. The key at position 712 in the CAL table matches the key from the seed “AGC”, so position 6179 in the reference is a CAL for the seed “TACAGC”. Note that multiple CALs for the same key are possible, such as the three CALs for seed “TAGCCT”. The tables also include additional information, not mentioned in the example, such as the offset information for <key, CAL(s)> pairs for each address.



Pointer Table

Addr	Base Ptr
TAA	709
TAC	711
TAG	713
TAT	null

CAL Table

Addr	Key	CAL
709	CAG	4977
710	CAT	3520 5642
711	AAG	5037
712	AGC	6179
713	ACT	1635 5484 7212
714	CCT	2823

Figure 2.4 An example of pointer and CAL tables

Additional complexity is introduced to this table by the fact that any given read may be from either the forward or reverse strand of the DNA double helix. Because the reverse strand is made up of the complementary bases of the forward strand, but running in the opposite direction, we call this the reverse complement. For example, for the read “TACAGC” the reverse complement is “GCTGTA”. Note that the initial “T” base pairs with the final “A” in the reverse complement, the second base “A” in the read pars with the penultimate base “T” in the reverse complement, and so on. Since there is no way to know which strand the short read is from, the system must be able to handle either seamlessly. This is done by generating the reverse complement of each seed and only looking up the lexicographically smaller of the two. This operation works correctly regardless of the original strand, since the reverse complementing operation is symmetrical. A similar operation is used when generating the tables, so only CALs from the lexicographically smaller seeds are present (this is ignored for the above example). The only additional complexity then required is keeping a bit with each CAL to signal if it is from the forward or reverse reference and performing the final Smith-Waterman operation against the reference in the correct direction.

In the basic organization described above, the number of CAL table entries associated with each pointer table entry would vary wildly due to the non-uniform distribution of seeds in the reference genome. To avoid this problem and provide a more even distribution of CAL table buckets, the entire seed is hashed. With this hash in mind, the entire process consists of comparing the seed to its reverse complement, hashing the smaller of the two seeds, looking up the correct pointer table entry using the address bits, following the pointer table to the correct CAL table bucket and then checking each key found there against the key from the seed. This last step requires walking the entire region of the CAL table specified in the pointer table to check for CALs with matching keys.

2.3.2 *CAL Finder*

The short read pipeline relies on a preconstructed index, built from the entire reference genome or some other sequence, as described above. Each read is divided into many overlapping or non-overlapping seeds depending on the biologist's preference (this is different than index construction, where overlapping seeds are always used). In the case of overlapping seeds, every possible full length seed is generated from the read for a total of (read length – seed length + 1) seeds. In the case of non-overlapping seeds, every full length seed is generated, starting at the beginning of the read, such that no base is in multiple seeds. This results in $\lfloor$read length / seed length$\rfloor$ seeds, and if the read length is not divisible by the seed length there will be some bases at the end of the read that are not part of any seed. An intermediate approach, with seeds overlapping by fewer bases, is also possible.

Each seed is looked up separately using the pointer and CAL tables. For the V1 system, overlapping seeds of 22 bases are used, and the CAL Finder module must walk the pointer table and CAL table as described above. The CAL Finder has one FPGA DRAM port for pointer and CAL table access and another for reference lookups. After loading the correct portion of the CAL table from DRAM, the CAL Finder sends all of the CALs with matching keys to the next module in the system (see Figure 2.5).

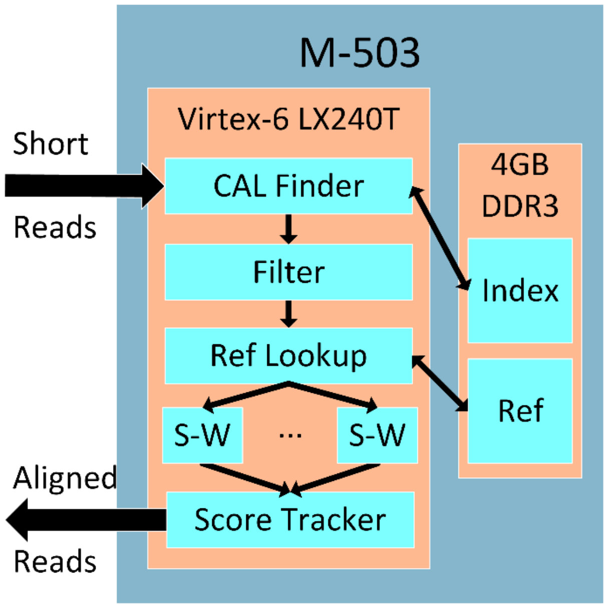


Figure 2.5 Architecture of V1 short read alignment system

2.3.3 *CAL Filter*

The CAL Filter has two functions. First, the CAL Filter combines duplicate CALs into a single reference lookup to avoid wasting DRAM bandwidth and S-W resources. This is necessary because many of the CALs for seeds of a given read will often be near each other in the reference. For an exact match with overlapping seeds, each seed will return a CAL one base after the previous seed. Second, the CAL filter translates the CAL to determine which 256-bit DRAM words of the reference must be read. Depending on the length of the short reads (typically 76 bases for V1) and the way the CAL falls relative to the start of a DRAM word, one or more words may be required for each CAL. After the CAL filter determines which reference words are required, the reference retrieval submodule loads them from DRAM, using a single port. For reads that have CALs that are not near each other, this may require loading many non-contiguous portions of the reference.

2.3.4 *Reference Lookup*

This very simple module manages memory accesses on the FPGA, and keeps the alignment units supplied with the appropriate sections of reference genome as specified by the CALs.

2.3.5 *Smith-Waterman Aligner*

Once the reference is retrieved, each read-CAL pair is passed to a Smith-Waterman (S-W) aligner, along with the appropriate reference. Unlike the other modules, which exist in a single chain in the V1 system, there are multiple S-W aligners in parallel to help alleviate what would otherwise be the performance bottleneck. The aligners in V1 use Smith-Waterman with the affine gap model, which allows the short read to have long insertions or deletions of bases compared to the reference without a significant score penalty. This scoring allows for more biologically relevant results. Details of the dynamic programming algorithms used in scoring can be found in [Zhang07].

The S-W implementation uses a 1-D systolic array of processors, which parallel-loads each short read (one base per processor) and matches it against a streaming reference as shown in Figure 2.6. The S-W algorithm requires a 2-D dynamic programing table, where the vertical axis represents the sequence of the read and the horizontal axis represents the sequence from the reference. The algorithm can calculate the value of a cell in the table using only the adjacent cells above, to the left, and the up-left diagonal cell. This means that the 1-D array of processors in each S-W unit can parallel load the read and then stream the reference sequence through them, producing a computation wavefront along the antidiagonal of the table, progressing from left to right. The cell with the largest score in the final row of the table represents the best score for the entire read, and is the output of the aligner.

READ \ REFERENCE	-	A	G	T	A	C	T	G	C	G	A
-	0	0	0	0	0	0	0	0	0	0	0
A	-2	2	0	-2	2	0	-2	-2	-2	-2	2
C	-3	0	0	-2	0	4	2	1	0	-1	0
T	-4	-1	-2	2	0	2	6	4	3	2	1
G	-5	-2	1	0	0	1	4	8	6	5	4
A	-6	-3	-1	-1	2	0	3	6	6	4	7
C	-7	-4	-2	-2	0	4	2	5	8	6	5

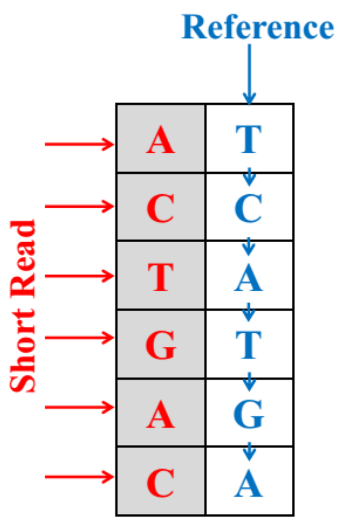


Figure 2.6 Smith-Waterman dynamic programming table (top) and hardware array based implementation (bottom)

2.3.6 *Score Tracker*

After an S-W score is computed for each unique CAL, the Score Tracker picks the reference position that has the best score for the read and is therefore the closest match. The best score may not start precisely at the candidate location, but may instead be off by a number of bases bounded by the read length (in the case where the seed is close to the end of the read). This is determined by the highest scoring cell in the final row of the S-W dynamic programming table as discussed above. There are multiple modes of operation available in V1, but typically only the best scoring result for each read is returned from the FPGA to the host, while the others are ignored.

2.3.7 *V1 FPGA Implementation*

The V1 implementation uses Virtex-6 LX240T FPGAs on Pico Computing M-503 modules [Pico Computing, Inc.14], discussed in more detail in 2.3.9. Each module is equipped with two 4GB DDR3 DRAMs running at 400 MHz. This DRAM is used to store the index and reference. For the human genome, the memory requirements are as follows:

$$\text{reference} = \sim 3.2\text{ Gbases} * 0.25\text{ bytes/base} = 0.8\text{ GB}$$

$$\text{pointer table} = \begin{array}{c} 2^{\text{address bits}}\left(\text{start pointer bits} + 2^{\text{tag bits}} * \text{offset bits}\right) = \\ 2^{26}(32 + 2^{4} * 14) = 2\text{ GB} \end{array}$$

$$\text{CAL table} = 1\text{ CAL/base} * \sim 2.44\text{ Gbases} * (4\text{-byte location} + 4\text{-byte key})/\text{CAL} = 19.5\text{ GB}$$

This results in a total memory requirement of over 23 GB, when using 26 address bits or 13 address bases (the default value for V1). The CAL table has fewer Gbases than the entire reference because of unknown bases or Ns.

As these calculations show, the size of the CAL table is much larger than the DRAM available on an M-503, so the design was partitioned across multiple modules. This means that each module was given the entire pointer table and a subset of the CAL table, split up using the first n bits of the seed. For example, using the first 3 bits of the seed, the index could be partitioned across 8 FPGAs, with each FPGA receiving 2.4 GB of the CAL table. This, along with the complete reference (which is required on each card) results in a total memory usage of only 5.2 GB, while also improving effective memory bandwidth across the system. When a read is streamed to each of the eight FPGA modules, the CALs for that read can be looked up in parallel across all of the modules. As discussed in the next section, memory bandwidth is a significant bottleneck for the V1 system, so this partitioning is desirable.

The CAL Filter is implemented using a hash table made up of FPGA block RAMs as described in [Olson11]. Each FPGA is given as many S-W modules as can fit to process all of the CALs being returned from the CAL Finder. All of the modules run at a 250 MHz clock, except the S-W processors which can only run at 125 MHz due to the significant sequential computation required.

2.3.8 *Performance Considerations*

In the V1 system, the S-W modules are fully pipelined during computation, but each module can only work on a single short read and reference segment at a given time. For efficiency and simplicity, the V1 system always processes the reference in 256-bit DRAM word sized sections. When the relevant reference from the CAL falls across a DRAM word boundary the S-W module must align the read against 512-bits of reference, or 256 bases. Because each read is loaded in parallel the engine is not pipelined between different reads, so there is a lead-in time equal to the length of the read, in this case 76 bases. The total time to process one reference section for one read is 256 + 76 = 332 cycles at 125 MHz or 377 kCALs / sec per S-W module. In the best case, assuming many CALs per read, performance could approach 256 cycles per CAL or 488 kCALs / sec per S-W module.

The V1 memory controller on the FPGA is clocked at 200 MHz and requires 45 controller clocks for each access, resulting in about 4.4 M transfers / sec. In this system, the memory accesses are not pipelined. Because of this, for each read the CAL Finder requires the full time to access the pointer table for each seed, followed by accessing the CAL table for each seed. Assuming an average of about 8 CALs being found for each read, as in [Olson11], that also requires 8 reference loads per read. Accessing the pointer and CAL tables once per seed results in (76 base read - 21 incomplete seed positions for a 22 base seed) * 2 + 8 reference access = 118 total memory operations per read. At 4.4 Mops this results in 37 kreads / sec, or 75 kreads / sec given the two independent memory channels on the M-503. However, given the partitioning scheme described above, a four M-503 system like V1 would only have to look up ¼ of the seeds for each read on each FPGA, effectively quadrupling performance while also allowing for the entire human genome and associated tables to fit in M-503 DRAM. The total performance of this system would be memory limited at 300 kreads / sec.

With this rate of memory access, only two S-W engines are required per FPGA to fully saturate the memory. Given that the V1 systems can fit a maximum of 6 S-W modules per FPGA, it is clear that the system is memory bandwidth limited. Pipelining the memory access portion of a 4 M-503 V1 system to saturate 6 S-W modules per FPGA would require 380 kCALs / sec / S-W module * 1 read / 8 CALs * 6 S-W modules / FPGA = 285 kreads / sec / FPGA. Given the current memory limit of 75 kreads / sec / FPGA, it would take memory pipelining with 4 reads in flight at a time to fully saturate the memory system on an FPGA with 6 S-W modules. Although this option is not available in the V1 system, such a memory controller is feasible and could result in a system bottlenecked by FPGA resources instead of memory bandwidth.

2.3.9 *Pico M-501 and M-503*

As discussed previously, the V1 system is implemented using Pico Computing M-503 FPGA modules as well as M-501 modules [Pico Computing, Inc.14]. The M-501 implementation is described in detail in [Olson11]. Significantly, the M-503 module has two 4 GB DDR3 DIMMs while the M-501 has only 512 MB of DRAM. The human genome requires CAL and pointer tables totaling 21.5 GB, in addition to the reference (3.2 Gbp or 800 MB). This means that depending on the partitioning, 48 M-501s might be able to align reads to the full human genome, but fewer would not. 48 is the maximum number of M-501 modules a 4U Pico Computing system can support using 8 PCIe slots with 6 M-501s on each EX-500 backplane.

For the V2 system (see section 2.4) the tables are stored on the host. However, the 512 MB on the M-501 would still limit how much of the human genome reference could be stored near each FPGA. Furthermore, some applications might require larger references, including the case study discussed in 2.1. Partially to address this issue, the V2 system is implemented exclusively on Pico Computing M-503 modules. These include the same Virtex-6 LX240T FPGA, but have 8 GB of DRAM per FPGA. This allows the system to store the complete reference genome with each FPGA, even for larger references. The theoretical maximum bandwidth of the M-503 DDR3 is ~10.8 GB/s, compared to a theoretical maximum of ~34 GB/s and measured performance of over 27 GB/s on our host.

2.4 V2 FPGA System

In light of the detailed explanation of the workings and limitations of the V1 system, this section will describe the thinking and modifications that lead to our current V2 system.

To summarize the previous section, the bottleneck of the V1 system is the memory operation rate on the FPGA boards. This bottleneck could be alleviated by pipelining the CAL Finder's DRAM interface, allowing for better utilization of the available memory bandwidth. This change would result in the CAL Finder and Filter modules taking up more FPGA resources, and those limited FPGA resources would become the bottleneck by reducing the number of S-W modules that could fit on the FPGA. Additionally, no amount of pipelining can overcome the fact that the aggregate memory bandwidth on the FPGA board is significantly less than the multiprocessor host system. These statements are analyzed in more detail in Section 6 of [Olson11] and will be discussed further below.

Even disregarding the performance, the V1 system has additional disadvantages. Because the CAL Finder and CAL Filter are FPGA modules hand coded in Verilog, any changes to this logic tend to be time consuming and error prone. Furthermore, although index construction itself is performed on the host, the CAL Filter is hardcoded with many of BFAST's assumptions related to the seed and key length, as well as the maximum number of CALs per seed and total genome size. These restrictions are particularly problematic for biologists who are accustomed to making frequent small changes to the parameters used by their algorithms. The appeal of an accelerated solution is significantly lessened if it is brittle and difficult to modify. Additionally, biology is full of problems that require finely tuned sensitivity parameters to achieve acceptable quality of results.

In order to address all of these issues, our current version of the system (V2) was designed to use the host for index lookups before sending the reads and all matching CALs to the FPGA for alignment. The design of this system can be seen in Figure 2.7.

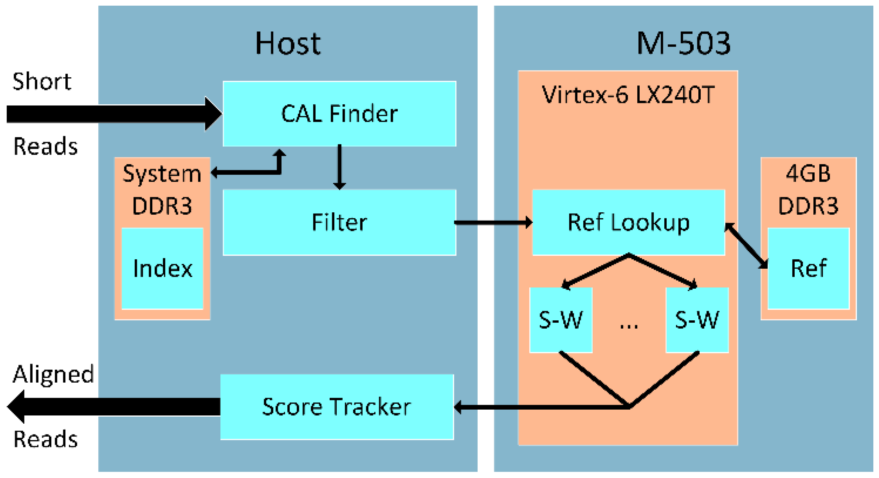


Figure 2.7 Architecture of V2 short read alignment system

2.4.1 *Index Construction and Traversal*

Index construction is similar to the V1 system. However, because the index lookups are performed in software, most index features can be configured at compile time or runtime. In addition to the ability to configure the number of bases that make up the seed, the breakdown of the seed into address and key bits is completely configurable. Increasing the seed length reduces runtime, since fewer seeds are generated from each read, but also reduces sensitivity. Increasing the portion of the seed dedicated to address bits results in a larger pointer table, since the pointer table is directly indexed. The advantage of this is a reduction in the time required to scan the CAL table, since there will be fewer non-matching CALs in each pointer table slot. All seed bits that aren't used for address are used for the key and offset, to match CALs in the CAL table. Changes to these values allow the system to mimic the results of other software packages more closely. For example, although they are similar in some ways, mrFAST and mrsFAST use very different seed configurations than BFAST [Alkan09][Hach10]. Being able to easily support both is a significant advantage over the V1 system.

In addition to the seed, the all-software indexing solution allows for configuration of the size of fields in the tables. Because the tables are loaded from disk for each alignment, different table configurations can be used with indexes built from different reference genomes without recompiling software or reconfiguring the FPGAs. This was not possible with the V1 system. This configuration includes the number of bits at each offset position. Increasing this value increases the size of the pointer table, but allows for more CALs at each offset position. As in the V1 system, the maximum number of CALs for a single seed is configurable, but it can now be set on an index by index basis.

2.4.2 *Software System*

The V2 CAL Finder and CAL Filter modules are implemented as software threads that communicate through Liberty Queues [Jablin10]. The organization of threads is shown in Figure 2.8. The greater aggregate memory bandwidth available on the host machine (section 2.3.9) to perform index lookups depends on an intelligent organization of threads to hide memory latency and achieve the maximum memory operation rate. As discussed below, the thread structure used here is sufficient to provide CALs to up to 4 FPGAs without becoming the bottleneck. A system that used C-slowing or some other technique to fit more S-W engines on the FPGAs could require more memory bandwidth on the host to keep up.

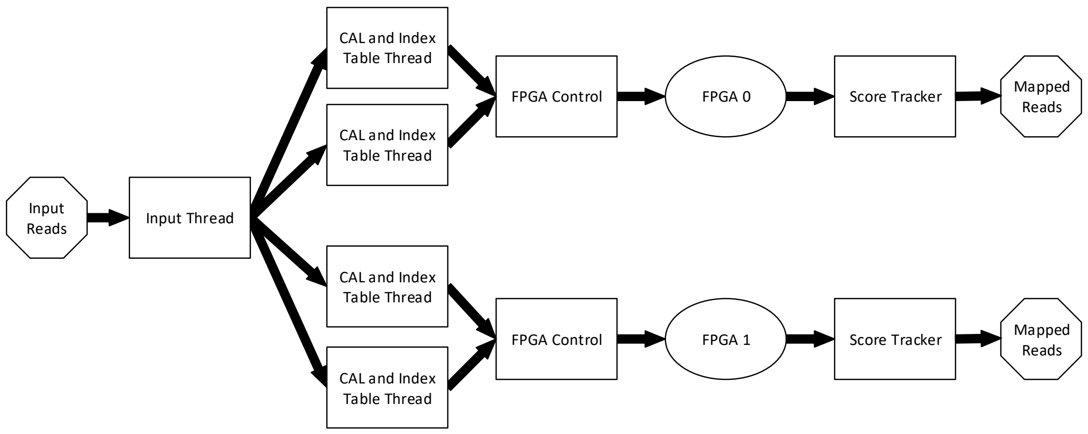


Figure 2.8 V2 software threads for 2 FPGAs

To summarize, a single thread reads the short reads from disk, and dispatches each read to a pool of index threads in a round robin fashion. Each index thread pair is assigned to an individual FPGA, and both index threads write to a single FPGA control thread that performs the same operations as the CAL Filter described in the previous section. These threads are paired to optimize memory access for our system, in which memory is local to one of two CPU sockets, and could be configured differently in a different system. The index thread sends the short read, followed by the filtered CALs for that read, to the FPGA; a similar thread receives results from the FPGA. The receive thread performs the operations of the Score Tracker, but is significantly more configurable than the V1 firmware version. The filtering options include returning the single highest scoring CAL, some number of highest scoring CALs, or all CALs tied for the highest score. This thread also formats the output into a standard format like SAM [Li09b]. This output is then written to one file per FPGA, to be aggregated later if required.

The V2 implementation uses the same FPGA boards as V1. Because only the reference has to be stored in DRAM on the FPGA modules, no partitioning of the reads is required for a human sized genome. CALs from any read can be aligned on any FPGA in the system and systems with a single FPGA are possible. The results below are reported for such a system, unless otherwise noted.

2.5 FPGA RESULTS

This section presents results for the V2 aligner in a few categories. First, we will provide more detail on our test platform and case study. Next, we will quantify the improvements in the V2 system. After this, we will discuss the performance of various configurations of the system. Finally, we will compare performance to other software and accelerated aligners.

2.5.1 *RNA-Seq and Classification*

Section 2.1 described situations, such as co-cultures, chimeric tissues and xenografts, where the short read data cannot be mapped to a single reference. In these cases the reads must be classified as one of two or more species at some point in the genomics pipeline. We will use mouse and human to represent the two species, as in our retinal co-culture case study. A number of approaches exist to address this problem.

Xenome [Conway12] decomposes RNA-Seq reads into K-mers (overlapping subsequences of length K) and attempts to identify K-mers in the read that could only be mouse or human. Another approach for RNA-Seq data is to use a standard aligner to align reads to the transcriptome or genomic reference sequences and use the alignment information to classify each read. This is the approach most similar to ours, as well as other recent work including Disambiguate [Ahdesmäki16]. Finally, [Tso14] evaluates other techniques including setting a score threshold or building a single reference from multiple genomes. That work concludes that aligning directly to the human genome, and ignoring the mouse component, produces low quality results but there is no obviously better existing co-alignment solution for all situations when both references are included.

In our work, classification is performed by aligning the reads to the mm10 mouse and hg19 human reference, including both the genome and additional transcriptome sequences from Ensembl release 72. These increased the total human reference genome size by 9% by adding 195k individual isoforms and the mouse genome size by 6% with 93k isoforms. Unlike the combined reference used in [Tso14], we aligned each read to the mouse and human reference separately. Although this increases run time, and actually disadvantages our aligner due to its ability to handle large references more easily, it allows us to produce a set of high quality Smith-Waterman scores against both references. We make the initial classification using the highest of these scores, but both scores can be preserved for later pipeline stages to make a more definitive classification if necessary, for example using paired-end information or a more advanced statistical model.

The use of alternative splicings (reference sequences for each known isoform, see 1.2.3.1), is due to the fact that our system is not a spliced aligner. This would require considerable development effort in the FPGA, and is outside the scope of this work, but it does create a somewhat unbalanced comparison with a spliced aligner like GSNAP. Specifically, we appended the previously mentioned Ensembl 72 isoforms to our reference genome before building the tables. We also did this with the other aligners in our comparison to avoid an unfair quality advantage in the results. These were treated such that alignments across isoform boundaries were not scored.

Including the full reference genome not only covers some additional reads, as discussed below, but it preserves some of the complexity to be expected in a full implementation of a spliced aligner, such as the impact of repeats and CAL limits.

The case study reads are RNA-Seq (section 1.2.3) mouse and human retinal reads. Human fetal tissue was obtained from the Birth Defects Research laboratory at the University of Washington without identifiers. For the mouse short read data, mice were housed in the Department of Comparative Medicine at the University of Washington. All procedures were carried out in accordance with protocols approved by the UW IACUC. Total RNA was isolated from postnatal day 0 mouse retinas and 80d whole human fetal retinas using Trizol (Invitrogen, Grand Island, NY). RNA was treated with RQ1 DNase (Promega, Madison, WI) for an hour and then purified using miRNeasy kit (Qiagen, Germantown, MD). 5 µg of each sample was then processed using Ribozero (ScriptSeq Complete kit; Epicenter, Illumina) to deplete ribosomal RNA. Libraries were generated and sequenced on an Illumina HiSeq for 100 bp paired reads. The classification was done on simulated co-culture data, with the goal of correctly identifying the species of origin. This approach has the advantage of knowing with certainty the true origin of each, resulting in an accurate evaluation of the classifier performance.

The read data described above is available as SRR6942748 and SRR6942780. Because these are ribosome-depleted total RNA, many reads will be aligned to parts of the full reference genome not included in the Ensembl 72 catalog. Reads aligning across splice junctions that are not included in the catalog may score poorly.

2.5.2 *V1 vs. V2 Performance*

For testing purposes, our system contains two Nehalem E5520 processors (8 cores total), 48 GB of DDR3-1333 DRAM and two Pico Computing M-503 modules (section 2.3.9). Although the specifications of this host are not identical to those used in [Olson12], this has no impact on the evaluation because the V1 system is almost entirely reliant on the FPGAs for performance. When comparing V1 and V2 performance, we use the 50 million short read sample dataset from the V1 evaluation. This is not the datasets described in the previous section, which will be used for all further evaluations, but we wanted to maintain as much of an apples to apples comparison as possible for the V1 and V2 systems.

In addition to the improved flexibility, the V2 system is significantly faster than V1 (see Table 2.1). Note that this table measures performance in CALs / sec and not reads / sec as used throughout this section, since users can adjust CALs / read to tradeoff between sensitivity and performance.

Table 2.1 Performance comparison between V1 and V2 systems using default V1 parameters.

	V1 System	**V2 System**
CALs / s / FPGA	632 k	2231 k
System Power (W)	435	384
Energy / CAL (J)	688 μ	172 μ

The performance improves by a quantity very similar to what was estimated for the "CALFinder on Host" system described in [Olson11]. That calculation was performed by estimating the number of seeds the host could generate per second based on the number of clock cycles required to generate a seed and the number of memory references per second supported by the host. If the host can generate enough seeds, the total CAL alignments per second supported by the FPGAs gives the total for the system. Because our current system only has two FPGA modules, these results are computed per FPGA in order to allow a direct comparison with [Olson12]. The V2 system is 3.53x faster and 4.00x more energy efficient than the V1 system. The exact improvement is highly dependent on the reference size and partitioning used, as well as the number of CALs per read, but substantial improvement is seen across the board. One of many uses for the ability to carefully tune CALs per read will be illustrated by the following case study.

Understanding the CALs / secs performance metric requires a brief discussion of the variability and distribution of the number of CALs / read. In particular, this number is highly dependent on both the seed and key length as well as the source of the reads. The frequency and length of repetitive regions is highly variable across genomes and within a single genome, as illustrated in Figure 2.9.

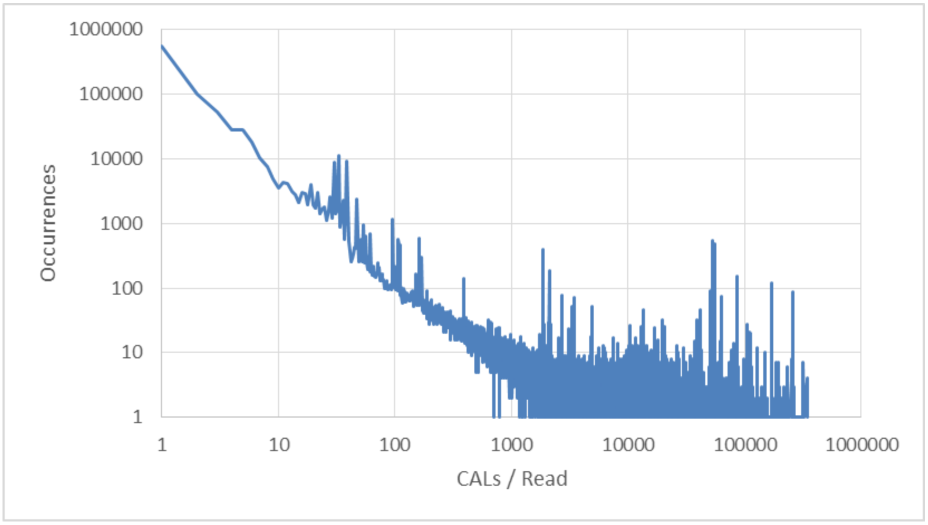


Figure 2.9 CALs / read distribution for 1 M human RNA-Seq reads aligned to HG19

On the left side of the graph the vast majority of reads only result in one or a few CALs. This makes intuitive sense because neighboring CALs are combined so a read that maps to a single unique region of the genome will only send one CAL to the FPGA for scoring. As the number of CALs / read increases the frequency decreases exponentially, but there is a very long tail where the number of CALs / read is extremely high and these occur more often than would be expected if one were to only look at the left half of the graph. These are the highly repetitive regions of the genome where a small string of bases is repeated many times. Some aligners limit the maximum number of CALs for a single seed, reasoning that for reads that map to so many places in the reference it is highly unlikely that one mapping will be significantly better than any other. Our system supports indexes built in this manner, and in fact gets a significant performance boost from doing so, but in the interest of doing an equal amount of computation we have not enabled this for the above comparisons to the V1 system. Given the fact that our current system is CAL scoring limited when using a single FPGA (see Table 2.2), it is clear that much of the variation in reads / sec measurements is due to changes in CALs / read. In other words, by limiting CALs / seed the number of CALs the system must process for each read decreases significantly (by more than an order of magnitude) and performance measured in reads / sec increases by roughly the same factor, as seen in the case study below.

To summarize, moving the CAL Finder and CAL Filter from the FPGAs to the host system, freeing significant FPGA resources for S-W alignment and removing all traffic from the FPGA DRAM except for reference lookups, offers performance improvements as well as significant benefits to configurability and genome support.

2.5.3 *Score Threshold and Tie Δ*

The improved configurability of the V2 system provides a number of knobs which can have a significant impact on results in some situations. Figure 2.10 shows a heat map of the S-W scores of a random sample of 5 million mouse and 5 million human reads. The simplest score-based classification scheme would classify the reads above the diagonal as mouse and reads below the diagonal as human.

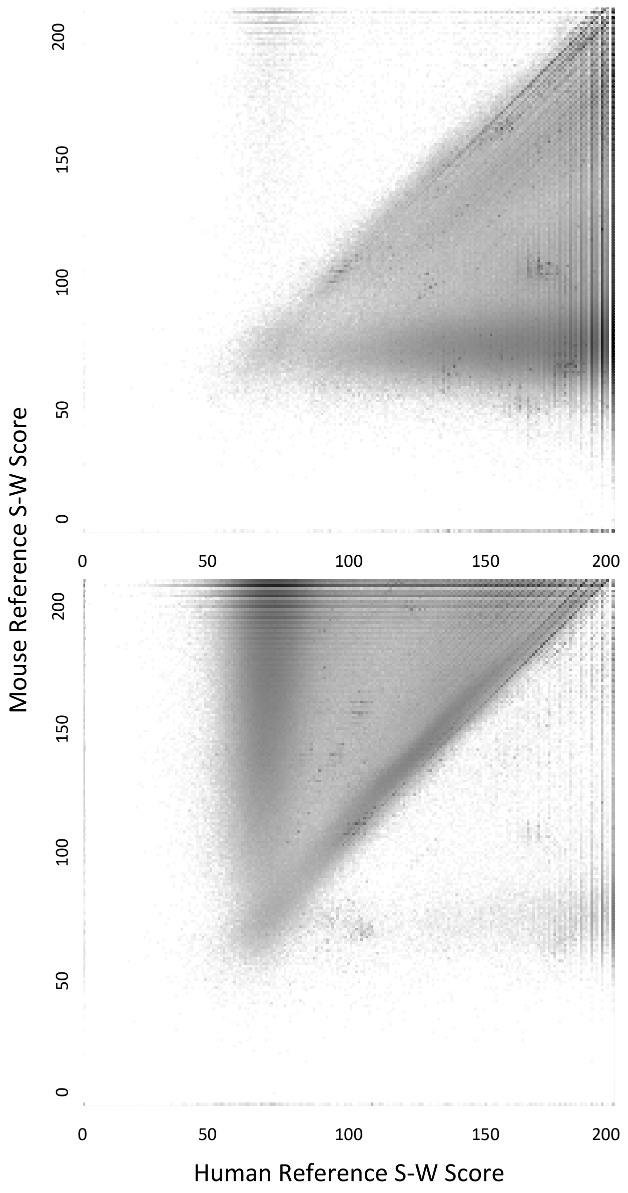


Figure 2.10 Smith-Waterman scores for 5 million 100 bp human reads (top) and mouse reads (bottom) aligned with our method to human and mouse reference, with seed length of 16 bp and no CAL limit. The color scale is logarithmic, with the lightest grey representing a single read and black representing > 2^{16} reads.

Most misclassifications occur in the low score region (with Smith-Waterman scores below 100), representing situations with a great deal of uncertainty in the alignment. For these tests Smith-Waterman matches and mismatches are scored as +2 and -2 respectively, as well as a gap open penalty of -2 and a gap continuation cost of -1. For this reason, a perfect match on a 100 bp read has a maximum score of 200. These values can be adjusted, although certain adjustments on the FPGA do require reprogramming the device.

These misclassifications can be divided into two categories. First there are cases near the axes with scores below one hundred. These represent a mouse read aligning better to the human reference, but often with a low score representing an alignment that is unlikely to be correct, or vice-versa. In the other case, there are misclassifications near the diagonal, with a similar alignment to both references. However, the incorrect one has a slightly higher score. We present solutions for both of these cases below.

For the first case (misclassification near the axes), introducing a score threshold can reduce incorrect classifications. Since very low scores are unlikely to be meaningful, setting a minimum score threshold to perform classification will result in fewer incorrect classifications and more reads going unmapped. The results of introducing this threshold can be seen in Figure 2.11. This figure shows the relative distribution of correctly (green), incorrectly (red) and unmapped (grey) aligned reads as well as ties (blue). A threshold of 0 means that all possible alignments are considered valid, while increasing the threshold results in lower scoring alignments being treated as unmapped. Note that this has no impact for very low scores, since our technique requires at least one perfectly matching seed.

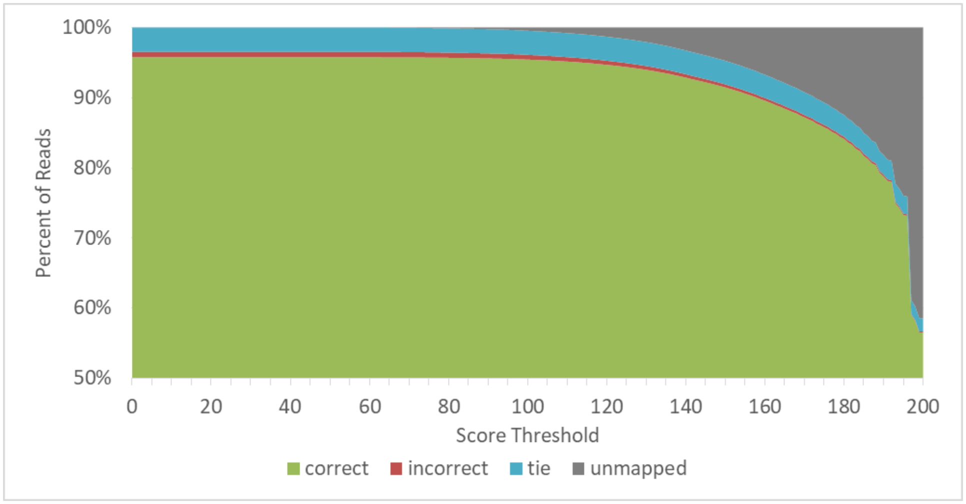


Figure 2.11 Impact of introducing a score threshold on distribution of correctly, incorrectly and unmapped reads. Clipped lower area (remaining 50% of reads) are correct.

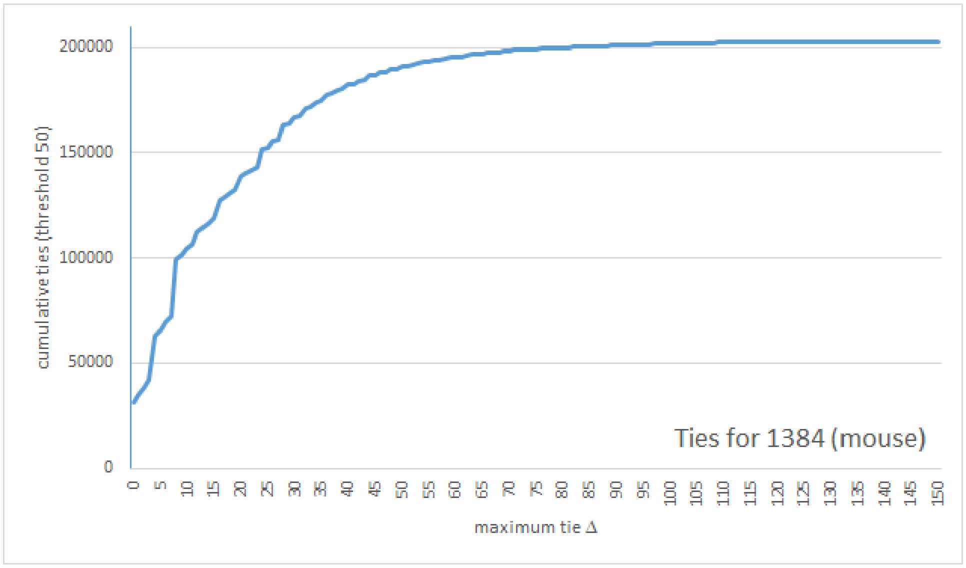


Figure 2.12 When using a score threshold of 50, introducing a tie-delta quickly discards otherwise meaningful results. Ties are out of a total of one million reads, and both figures were generated with 25bp seeds.

In the second case, very similar scores result in a number of incorrect classifications. In these situations, score alone is not sufficient to classify the reads correctly, so they may be tagged as similar (ties) so that the downstream systems can use additional information to resolve the tie. Our system addresses this with a tie delta, which classifies reads that score similar, within a difference of Δ, as ties instead of as belonging to the slightly higher scoring reference. Figure 2.12 illustrates the impact of this technique. It increases the frequency of ties at the expense of correct and incorrect alignments, while having no effect on the percentage that are unmapped.

Although we explored a score threshold to filter low score alignments, and a range for ties (instead of requiring exactly the same score), we found that neither of these significantly improved the quality of the final results. Thus we don't apply these methods further in our results. Despite the lack of positive impact in this case, our technique and classification tool makes this and many similar options trivial to implement, as they could increase result quality on a different data set or as part of a specific pipeline. Observe that increasing the range of ties up to about 50 greatly increase the number of ties, but using a low score threshold makes very little difference. Score thresholds of nearly 100 are required before a significant impact becomes visible, and it is likely that many correctly mapped reads are being excluded at this point.

2.5.4 *Classification Speed and Quality*

Given the emphasis placed on the configurability of our software/FPGA hybrid solution up to this point, we must evaluate the impact of these configurations on both speed and classification accuracy. In particular, it is possible to adjust sensitivity via the seed length and CAL limit. Shorter seeds increase the number of CALs analyzed per read, and therefor sensitivity, but reduce the number of reads processed per second, as discussed previously. CAL limits cap the number of CALs scored for a given seed, essentially eliminating the most problematic cases on the right hand side of Figure 2.9. In practice this improves performance without a large impact on quality because seeds with very high CAL counts are usually repeated regions in the genome and there is little value in scoring all of them.

Table 2.2 shows a comparison between seed length, accuracy and speed. For each seed length the table includes rows for human and mouse reads, with reads from each specie aligned against both references. “Correct” is the percentage scoring higher with the correct reference, while excluding ties and unmapped reads. Note that no effort is made to resolve ties even though many approaches could improve this correctness score here, including using the paired end information or analyzing the read nucleotides with a probabilistic model. Reads and kCALs / sec measure the alignment speed averaged across a number of runs.

Table 2.2. FPGA Classification Results vs. Seed Length with reads / sec calculated with unlimited CALs aligning to human and mouse references, including alternative splicings

Seed	Reads	Correct	Reads / sec	kCALs / sec
16bp	Human	95.7%	290	3,670
	Mouse	95.2%	919	4,096
20bp	Human	95.6%	940	4,017
	Mouse	95.4%	2,389	3,937
25bp	Human	95.4%	3,071	3,952
	Mouse	94.3%	7,142	4,005
29bp	Human	94.8%	9,976	4,068
	Mouse	93.3%	17,200	3,898

As expected, reducing the seed length generally increases the sensitivity, but at the cost of reduced throughput. These seed lengths are selected to produce the best possible coverage of the read for non-overlapping seeds, given a 100 bp read. For example, there are six 16bp seeds, five 20bp seeds and four 25bp seeds. 29bp seeds are used instead of 33bp because of a limitation in our current FPGA programming files, but this is in no way fundamental to the design and could be improved in future versions.

Performance is generally significantly better when aligning mouse reads, instead of human reads, to the human reference. This is expected with unlimited CALs per seed for a few reasons.

First, the high CAL seeds are very frequent repeats. Many of these are evolutionarily recent hence highly conserved with little variation. Furthermore, these repeats do not exist in the last common human-mouse ancestor [Bourque04]. The less recent repeats that the human and mouse reference genome share are not as highly conserved, so they do not result in so many CALs. Ensembl 72 also contains more human isoforms than mouse, resulting in more CALs per seed for regions in exons.

For each seed length the performance more than doubles when transitioning to the next length. This is because there are fewer CALs to consider in the unlimited CALs per read situation. Observe that the reads / sec is much greater even though kCALs / sec remains roughly unchanged.

When using short seeds, reads that otherwise might have gone unmapped find low scoring alignments that may not be biologically meaningful. For human reads and 16bp seeds almost none go unmapped, and only 0.1% have an S-W score under 76. For a 25bp seed, 0.49% of reads go unmapped and only 0.05% score below 76. Introducing a threshold reduces this effect and similarly decreases the advantage of 16bp reads in terms of “correct” classification. Note that introducing a threshold does not improve throughput.

As discussed above, CAL limits reduce the number of possible matches in the CAL table for a given seed. Since each unfiltered matching CAL has to be passed through a Smith-Waterman engine, and the CAL table entry traversed linearly, limiting CALs can have a dramatic impact on performance. Since many cases where there are very many CALs per seed are repeat regions with tens of thousands of CALs or more, introducing a reasonable CAL limit has a small, but non-negligible impact on result quality.

Table 2.3 shows the impact of a CAL limit on performance. For 16bp and 25bp seeds, a limit of 1024 and 128 CALs per seed was applied. Lower limits are also possible, but do not result in significant performance benefits. These levels of performance assume CIGAR string [Li09c] generation has been disabled. CIGAR strings encode the mismatches in an alignment and are non-trivial to generate. Because this process is not accelerated on our current FPGA system it becomes a bottleneck at high throughputs and accelerating CIGAR generation is outside the scope of this work. BWA-SW generates the CIGAR string with the alignment, at very little performance overhead, so the comparison remains fair.

Table 2.3 Alignment performance against the human reference when adjusting seed length and CAL limit, without CIGAR string generation

Seed	CAL Limit	Reads	Reads / sec	kCALs / sec
16bp	Unlimited	Human	290	3,670
		Mouse	919	4,096
	1024	Human	53,000	5,873
		Mouse	71,000	6,089
	128	Human	100,000	4,861
		Mouse	128,000	5,461
25bp	Unlimited	Human	3,071	3,952
		Mouse	7,142	4,005
	1024	Human	148,000	3,907
		Mouse	289,000	4,515
	128	Human	203,000	2,364
		Mouse	336,000	2,222

Introducing a CAL limit greatly improves performance. For the 25bp seed length case, aligning human reads to a human reference is 67x faster when limiting CALs to 128. For mouse reads, the system approaches its maximum throughput of roughly 330k reads / sec. At this point, alignment on the FPGA is no longer the bottleneck for the system (observe the drop in CALs/sec), and more software threads and host memory bandwidth would be required to increase performance. As shown in these results, introducing a CAL limit is hugely beneficial in terms of performance.

The impact of the CAL limit on result quality can be seen in Figure 2.13. The vertical axis is the portion of reads that are incorrectly called or remain unmapped, excluding ties. The horizontal axis is seed length. As seen previously, increasing seed length decreases accuracy, particularly for the mouse reads. Introducing a CAL limit also decreases accuracy and the effect is cumulative with increasing seed length. The results are worse in the mouse case, possibly due to fewer alternative splicings being included with our reference. This is consistent with our performance results, in that alignment to the mouse reference is faster due to fewer CALs being processed, even before applying a CAL limit. Given these results, introducing a CAL limit is highly desirable for performance, but correctness suffers, particularly for smaller limits, so the parameters must be selected carefully depending on the requirements of each use case.

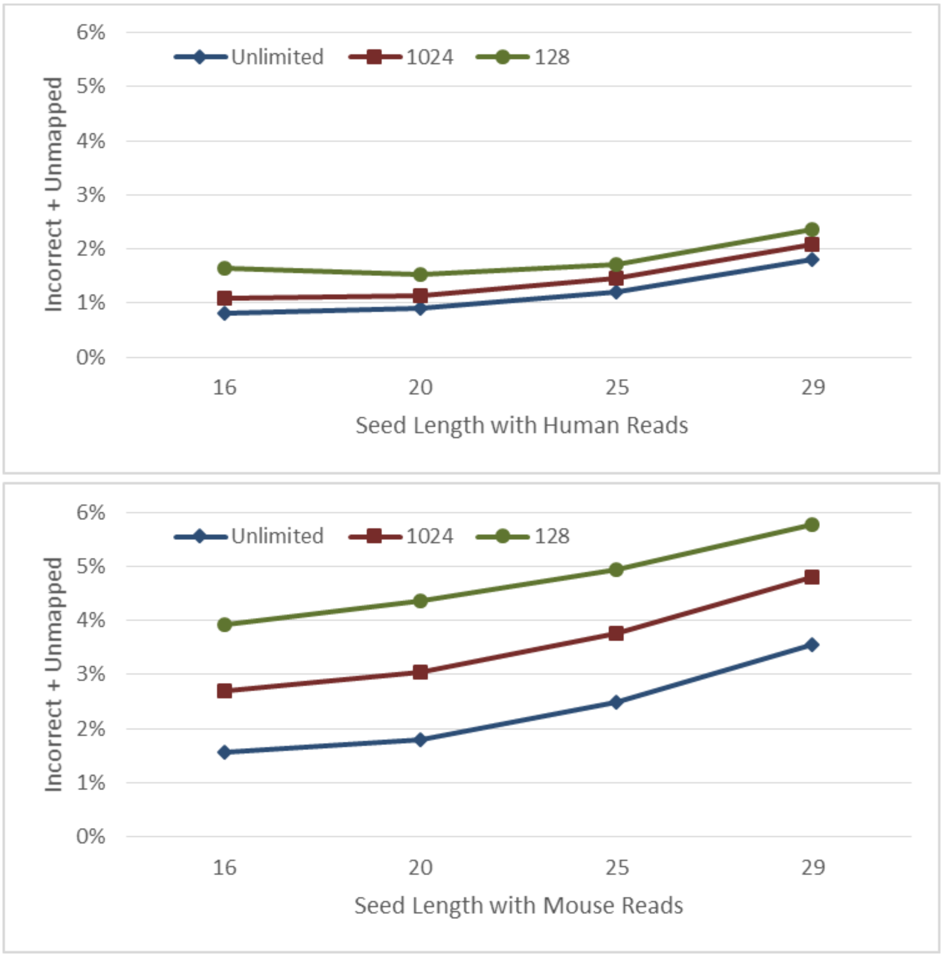


Figure 2.13 Incorrect and unmapped share for various CAL limits for human (top) and mouse (bottom) reads

2.5.5 *Performance Comparison to Software Aligners*

To evaluate our system, we compare its performance to five software based aligners. These are BFAST [Homer09], GSNAP [Li09a], BWA-SW [Li10], BWA-MEM [Li13] and HISAT2 [Kim15]. BFAST has already been discussed in detail. GSNAP is an aligner specifically designed to get good results in the case of alternative splicing events, as well as supporting common biological variation better than BFAST. BWA is a Burrows-Wheeler transform based aligner that has traditionally been extremely fast, although it has a poor quality of results in the face of variation or large read errors. More recently, an enhanced version for longer reads, called BWA-SW, has been produced which uses a Smith-Waterman aligner in the final stage, significantly improving variation tolerance. BWA-SW is not specifically designed for alternative splicing, but like our FPGA system it can handle very large references without trouble, so the same method of including each isoform can be used. For the evaluation, the same S-W penalties were used with our system and BWA-SW. BWA-MEM is a more recent version of BWA, optimized for short reads and without spliced alignment. Finally, HISAT2 is another FM-index based aligner with support for spliced alignment. For HISAT2 we did not include alternative splicings in the index. The versions of the aligners in the evaluation are found in Table 2.4.

Table 2.4. Software aligner versions

Aligner	Version
GSNAP	2014-11-25
BWA-SW	0.7.10
BWA-MEM	0.7.10
HISAT2	2.1.0

The results of the first experimental runs, without alternative splicings, are shown in Table 2.5. Each row contains the results for an aligner, and the columns show the percentage of reads that were either correctly identified as being of human origin or incorrectly identified as having a mouse origin. A correct identification comes from aligning the read to both the human and mouse genomes and having the alignment to the human reference scoring better than the alignment to the mouse reference. A read that aligns to the human reference, but not the mouse reference, is also marked as correct. Similarly a read that aligns to both with a higher mouse score, or only aligns to mouse, is considered incorrect. The last two columns show ties, where the read aligned equally well to both references and unaligned reads. For the FPGA system, unaligned means that none of the seeds returned any CALs. For the BWT aligners it is more difficult to summarize the significance of an unaligned read other than to say that it failed to pass program-specific significance thresholds.

Table 2.5 Aligner quality results (Human reads only) without alternative splicings

Aligner	**Correct**	**Incorrect**	**Tie**	**Unaligned**
BFAST	86.1%	8.00%	05.87%	00.00%
GSNAP	75.2%	0.03%	18.20%	06.36%
BWA-SW	93.8%	1.23%	03.14%	01.83%
HISAT2	80.3%	0.81%	03.46%	16.26%
FPGA	93.3%	2.57%	03.32%	00.80%

When alternative splicings are not included, the results are still very good overall. BFAST, which is the oldest aligner, unsurprisingly gets the worst results in terms of incorrect classifications. It is included here because it uses techniques somewhat similar to our FPGA aligner, as described above. GSNAP has many more ties than the other aligners, presumably due to its coarser grained scoring system. The results are very similar between BWA-SW and the FPGA aligner. Both use S-W for scoring and were configured to use the same S-W mismatch penalties, so this is an expected result. Without alternative splicing, BWA-SW has fewer incorrect alignments but more reads going unaligned. This is likely due to the FPGA system's more permissive CAL Finder, which can handle more mismatches than the BWT based initial matching in BWA-SW. HISAT 2 has few incorrect results or ties, but leaves by far the most reads unaligned.

Table 2.6 shows data for the case where alternative splicings are included in the reference. Note that we were unable to build a BFAST index with the full set of alternative splicings, and thus it is not included in this comparison. GSNAP has a very low percentage of incorrect calls but high unmapped and many ties, due to the lack of a Smith-Waterman score to resolve close alignments. BWA-SW is very accurate, but with a relatively high unmapped percentage, due to the absolute nature of the algorithm. Our FPGA based solution (using 25bp seeds) has the highest percentage of correct mappings, however a significant number of unmapped locations in the other algorithms become ties. This is due to a much greater ability to detect low scoring alignments, and by using a low score threshold these reads could be moved to unmapped instead. As discussed previously, we don't feel that this technique has a major impact on quality, but could be applied on a case-by-case basis. A tie aligns to both genomes with the same score, and can often be resolved later using the paired-end read or other methods.

Taking all of these results in conjunction, the FPGA system produces results of noticeably better quality than BWA-SW when alternative splicings are included. The only exception is for mouse reads with a 128 CAL limit, where the FPGA solution produces significantly more incorrect reads while BWA-SW has more unaligned. There are situations where this can be disadvantageous. For example, in our case study RNA-Seq data is often used to determine gene expression levels. In these cases it is most important to filter out reads with uncertain alignments. This can be addressed using a higher score threshold, essentially marking more reads as unaligned in our system.

Table 2.6 Aligner accuracy results with alternative splicings

Aligner	Reads	Correct	Incorrect	Tie	Unaligned
GSNAP	Human	80.0%	0.213%	19.1%	0.697%
	Mouse	75.6%	0.763%	22.2%	1.39%
BWA-SW	Human	94.4%	1.01%	03.04%	1.54%
	Mouse	92.9%	2.40%	02.49%	2.26%
BWA-MEM	Human	95.6%	0.54%	3.83%	< 0.0001%
	Mouse	95.2%	1.35%	3.47%	< 0.001%
FPGA unlimited	Human	95.7%	0.807%	03.46%	< 0.0001%
	Mouse	95.2%	1.55%	03.29%	< 0.001%
FPGA 128 CALs	Human	95.3%	1.627%	03.43%	0.015%
	Mouse	92.9%	3.716%	03.18%	0.211%

In addition to the significant change in the sensitivity, the performance of each system changes dramatically with the size of the reference. Performance results for references with and without alternative splicings are shown in Table 2.7. On our test platform, the FPGA accelerated aligner has significantly higher performance, with a 5.57x speedup over BWA-SW with alternative splicings included and a 433x speedup when using a 128 CAL limit, while still maintaining superior quality. Compared to BWA-MEM under the same conditions the FPGA is 94x faster with similar quality. The FPGA is 22.9x faster than HISAT2, but with far fewer unmapped and incorrectly mapped reads. These speedups are calculated using the performance values from the 3rd column of the table. As with the previous comparison, we were unable to successfully build a BFAST index that included all of the isoforms used with the other aligners.

Table 2.7 Impact of alternative splicings on aligner performance

Aligner	Without Alternative Splicings (kReads / sec)	With Alternative Splicings (kReads / sec)	Alternative Splicings Slowdown
BFAST	0.266	N/A	N/A
GSNAP	0.936	0.273	3.53x
BWA-SW	0.698	0.623	1.12x
BWA-MEM	3.07	2.87	1.07x
HISAT2	11.8	N/A	N/A
FPGA (unlimited)	5.83	3.47	1.68x
FPGA (128 CALs)	324	270	1.20x

Introducing the CAL limit also reduces slowdown for the FPGA system from 1.68x to 1.20x. This is likely because multiple isoforms contain the same exons, and reads that align to those sections result in a significant increase in average CALs per read. This increase is smaller than what might be anticipated, since most CALs should be in genes and our data includes roughly 10 isoforms per gene. However, there are a number of mitigating factors. First, each exon is only in some isoforms. Second, many isoforms are relatively short so the additional CALs will be removed during CAL filtering. Finally, since this is total RNA data, many reads do not map to a coding region at all.

These results demonstrate a scenario where the flexibility and robustness of our FPGA accelerated solution really shine. In particular, BFAST is completely incapable of handling the alternative splicings and GSNAP suffers a severe degradation in performance (although this is not entirely fair, since GSNAP is intended to discover alternative splicings during operation and not necessarily include them directly in the reference as we have done here). Although the FPGA system appears to suffer more than BWA-SW, it is important to note that no parameter tuning was done for this problem whatsoever, and it is our belief that by tweaking CAL / seed limits and other software parameters, none of which requires rebuilding bitfiles on the V2 system, this performance degradation could be mitigated or even eliminated without decreasing the quality of results. In particular, adding complete alternative splicings (as we did in these benchmarks) – instead of only the exon junctions and enough bases to cover the read length on either side – creates a number of identical regions where the same exon configuration repeats. For example, in the case of 5 exons, the overlap between exon 1 and 2 would exist for splicing 1-2-3, 1-2-4, 1-2-3-5 and so on. RNA-Seq data is likely to hit these regions frequently, greatly increasing the number of CALs per read. The current CAL Filter is not designed to detect and remove these sorts of pseudo-overlapping CALs because they have entirely different addresses in the combined reference. Instead, each possible splicing produces a separate CAL, which is a situation in which BWT based aligners like BWA-SW are much less susceptible to performance degradation. Introducing a CAL limit mitigates this somewhat, as seen in the data.

2.5.5.1 Comparison to Software Classifiers

The accuracy of our classifier with the V2 system compared with Xenom and Disambiguate can be found in Table 2.8. A direct comparison to other recent classifiers such as bamcmp [Khandelwal17] is not included because their results are very similar to Disambiguate [Dai18] when high quality alignment scores are available, as they are in this case. For this comparioson the human reads used are SRR387400 and the mouse reads are SRR1930152. All reads are RNA-Seq and Xenome and Disambiguate results are taken from [Ahdesmäki16]. The Xenome human sum is < 100% in that work. The Ambiguous column for the FPGA classifier is the sum of ties and unaligned reads.

Table 2.8 Classification accuracy of FPGA aligner compared to software

Reads	Classifier	Correct	Incorrect	Ambiguous
Human	Xenome	90.3%	0.14%	3.43%
	Disambiguate	83.3%	0.16%	16.56%
	FPGA	93.3%	0.64%	6.06%
	FPGA w/ Paired End	94.1%	0.81%	5.05%
Mouse	Xenome	95.9%	0.18%	2.60%
	Disambiguate	96.1%	0.34%	3.53%
	FPGA	98.1%	0.32%	1.60%
	FPGA w/ Paired End	98.4%	0.35%	1.29%

For both human and mouse RNA-Seq reads the FPGA outperforms the software classifiers by a substantial margin. Our classifier correctly identifies many more reads in both cases, with a slight uptick in incorrect classifications for the human reads. This can be mitigated with score thresholds and tie deltas, as discussed in 2.5.3, while maintaining a correctness advantage.

Our FPGA-based classifier can also make use of paired end information, like Disambiguate. Our technique classifies based on the higher scoring of the two ends for each reference, and uses the lower of the two ends in the case of a tie. Doing so correctly classifies an additional 0.8% of human RNA-Seq reads and 0.3% of mouse reads, with a smaller increase in incorrect classification. Again, we believe that this increase in incorrect classification can be mitigated or eliminated through careful tuning. In this case our solution correctly classifies 39% of the human reads that were not correctly identified by Xenome and 62% of the mouse reads that were not correctly found by Disambiguate.

Finally, it is worth noting that our aligner and classifier work very well with DNA reads, as well as the RNA-Seq reads that we have consider here. A more thorough analysis is not included, simply because DNA read classification is already extremely accurate with all tools and our system was optimized for RNA-Seq classification. For SRR1176814 mouse DNA reads (classifying the mouse reads is typically more difficult than human) the FPGA classifier is 99.4% correct, 0.35% incorrect and 0.29% ambiguous. These results are better than Xenome and slightly worse than Disambiguate.

2.5.5.2 Power Consumption

In addition to performance, it is important to evaluate the power requirements of our FPGA system compared to software only aligners, so we compare the FPGA-based solution and BWA-SW. One way to measure the energy required for a computation is to look at the power consumption of the entire system for the length of that computation. Using this method we calculated the energy required to align one million reads in each system, seen in Table 2.9. Each row represents an aligner, with the first row being the idle power of the system. The 2 FPGA system results are included to demonstrate the benefit of adding additional FPGAs that can share the software threads feeding them data. Although the FPGA system used the most power, it was also the most energy efficient due to the shorter runtime. Adding more FPGAs increases efficiency even further as expected, and although we only had 2 FPGA boards available for testing this effect should scale with more. The energy per read, in joules, is simply computed as Alignment Power * (seconds / read).

Table 2.9 Aligner energy comparison

Aligner	Power (W)	Time (sec / 1M reads)	Energy (J / read)
Idle	286	N/A	N/A
BWA-SW	303	1433	0.434
1 FPGA	350	300	0.105
2 FPGAs	384	175	0.067

Using a single FPGA decreases energy per read by 75.8% compared to BWA-SW and using two FPGAs decreases energy by 84.6% compared to a single BWA thread. Because additional BWA threads do not scale as well as FPGA threads, due to the lack of shared compute resources between them, the advantage actually increases when comparing multiple BWA threads to multiple FPGAs, but that analysis is outside the scope of this work. Additional, our test platform only included 2 FPGAs. Finally, note that these results were performed without a CAL limit. When a CAL limit is introduced the advantage of the FPGA system increases significantly, as seen in the above performance results.

2.5.6 *Performance Comparison to Accelerated Aligners*

Finally, we compare our work to other FPGA accelerated short read aligners (Table 2.10). We have done our best to compare results fairly, but the limited experimental details provided for some systems and the fact that each evaluation uses different datasets, may introduce some additional variation. Our aligner, without alternative splicings, performs similarly to [Waidyasooriya16] and [Sogabe17], despite being more accurate than BWA. We believe our higher accuracy comes from using a CAL table based method, as opposed to the FM-index based method of [Waidyasooriya16] and BWA. Our system performs well in situations with low repetition in the reference, as discussed previously. In situations with a highly repetitive reference, the CAL limit boosts performance at the cost of accuracy. However, in many cases, the alignments to highly repetitive regions are either equivalent or meaningless giving our system best of both worlds results. This explains some of our advantage over more modern, but essentially straightforward FM-index based, aligners.

Although [Sogabe17] uses a hash-based approach, similar to ours, the seeds are masked, as in GSNAP. This may explain the lower accuracy but increased performance for real world data, with potentially many mismatches from the reference (see Fig. 8 in that work). A significantly larger FPGA is also required, and given that our performance scales with PEs, it’s not clear which system would perform better on an FPGA of the same size. [Arram16] is different, in that it is very FPGA resource efficient but requires a great deal of off-chip DRAM to store expanded FM-indexes. The system described uses 8 FPGAs, with slightly lower per-FPGA performance. This is very difficult to compare to our system, since it is unlikely we could maintain linear scaling up to 8 FPGAs with a single host, due to the memory bandwidth requirements.

Table 2.10 Comparison to FPGA accelerated aligners, normalized to a single FPGA

Aligner	**Device**	**kLUTs**	**BRAMs (Mb)**	**kReads / sec**
Our V2	Virtex-7	191	05.6	336
[Waidyasooriya16]	Stratix V	235	50.0	313
[Sogabe17]	Virtex-7	336	31.5	511
[Arram16]	Stratix V	075	17.6	198

2.6 Conclusions and Future Work

In this chapter we presented the use of an FPGA-based short read aligner for classification of simulated co-culture RNA-Seq reads. Our aligner performs at least 5.57x faster than similar software aligners with better result quality and up to 433x faster with similar result quality, compared to a single CPU. The ability to use the parallelism available on the FPGA to calculate many more full Smith-Waterman scores than BWA-SW appears to have a significant positive impact on result quality. The FPGA system makes use of the available memory bandwidth on the FPGA board as well as the much greater host memory bandwidth to achieve this high performance while also improving flexibility and configurability to support configurations similar to a wide variety of popular software-based alignment algorithms.

Using this platform for classification, we demonstrated that a simple Smith-Waterman alignment score comparison acts as a very good classifier in the case of mouse and human retinal RNA reads. In addition to this simple classification scheme, providing a set of scores for each read against each reference enables down-pipeline applications to perform more complex classifications, including those using paired-end information.

There are a number of possible improvements for future versions of the system. For classification, a system that supports spliced alignment would have many advantages, including the ability to identify new isoforms and potentially better classification accuracy. More generally, for the FPGA-based aligner, use of a higher bandwidth memory like the HMC could allow index lookups to be moved back onto the FPGA, as in the V1 system. In this case, the HMC could store the index and the reference could be stored in DRAM, since access is lower bandwidth and sequential. Performance of this best of both worlds system could easily scale across many FPGAs eliminating our current systems bottleneck in terms of FPGAs per host and therefor total performance per host.

Chapter 3. NCRNA HOMOLOGY SEARCH

Prior to discussing ncRNA in detail, it will be valuable to explain a specific characteristic of protein structure. Proteins are, as discussed in section 1.2.3, among the best known and most important biological molecules. They perform many key cell and intercellular functions, and many sections of the human genome, called genes, encode proteins. Through alternative splicing, a single gene can encode many proteins, and many proteins with different sequences actually perform very similar functions. These similarly functional proteins can be grouped into families, with identifiable characteristics, usually due to sharing a common ancestor or co-evolution, and these families are available in a database called Pfam [Finn16]. The savings in complexity available when viewing proteins in terms of families is enormous. For example, the largest family, MFS-1 contains over 200,000 sequences across over 5000 species.

The function of a protein is typically a property of its shape in three dimensions, also known as tertiary structure, as well as the properties of the amino acid sequence in specific regions to create sites that interact with specific molecules. These regions are mostly the same across all of the proteins in a family. In other words, they are conserved. Similarly, the sequence of the remaining regions does not change very much, in order to preserve the tertiary structure, so using the sequence of a protein to identify the family, or using a library of protein families to find new members based on gene sequences, is relatively straightforward [Eddy98]. The following sections discuss ncRNA and the way they differ from proteins with this information in mind. The work described in this chapter has been published in [McVicar13].

3.1 NCRNA BACKGROUND

ncRNA, or non-coding RNA, are RNA molecules that are transcribed in the same way as mRNA, but are not translated into proteins. Although proteins are often considered the most important biological molecules, ncRNAs have been shown to play a similar functional role. This realization was first made with transfer RNA (tRNA), discovered in the 1960s. During protein synthesis, tRNA bind with amino acids on one end and a specific codon in an mRNA strand on the other end, thereby playing a critical role in the construction of protein. This is just one of the most basic examples of the thousands of functions ncRNA perform.

Although the importance of ncRNA was recognized soon after its discovery, recent work has identified many more ncRNAs with important biological roles. Over 70% of the genome is transcribed into RNA in humans, even though only roughly 1.5% directly codes for proteins. Although containing many repeats and other "junk" regions, much of the remaining 68.5% of the genome is ncRNA, and have been shown to have an important role in cancer, neurological disorders and cardiovascular disease [Magistri17]. While there is still a great deal of debate, estimates put functional regions between 5 and 20%, although some metrics are much higher [Kellis14].

The roles of ncRNA are not minor or incidental, either. For example, "non-coding RNAs have been identified as critical novel regulators of cardiovascular risk factors and cell functions and… new therapeutic tools were developed from endogenous ncRNAs serving as blueprints" [Poller17]. In cancer studies, researchers at the Hanover Medical School have identified ncRNAs that perform a critical regulatory function helping the body maintain blood homeostasis. These systems undergoes predictable changes in some leukemia patients [Schwarzer17].

3.1.1 *ncRNA Families*

Given its biological importance, it is valuable to be able to identify ncRNAs with key roles across locations in the genome of a single species, as well as between species. Before the extent of the roles ncRNA played was understood, a similar problem was addressed for protein using homology. Homologous proteins share a common ancestor and perform a similar function. Typically, these proteins also have a similar sequence. Because of this, homology search, which builds a model representing a family of proteins and then finds novel proteins that fit the model, is a very useful way of identifying proteins that may serve a specific function without performing costly in vivo experiments. Since ncRNAs are functional, they typical have similarly preserved homologs, which can be identified using ncRNA homology search.

3.1.2 *Covariance*

Although the complex chemical interactions of its tertiary structure allow proteins to function, the sequence of bases that make up the protein, called primary structure, is typically enough to model and identify its family, as discussed above [Finn16]. Similarly, after transcription, the bases of an ncRNA molecule will form a complex three-dimensional structure determined by base pairing. However, unlike proteins, without considering a two-dimensional model of the shape (secondary structure) formed by these pairings, it is not possible to accurately classify ncRNAs by family or build family models.

For insight into why this is the case, consider Figure 3.1. The sequence (primary structure) of an Iron Response Element (IRE) ncRNA is provided for four different species. Missing bases in some species appear as dots. Notice that the sequences are very different in some places but similar in others. Also observe that in the highlighted regions it is the pairing that is preserved, not the bases themselves. In the human the highlighted regions form U-A C-G C-G bonds while in the red junglefowl those are replaced with very different U-A A-U U-A bonds. This is possible because the sequence of homologous ncRNAs can change significantly from their common ancestor, but still remain part of the same functional family if the secondary structure is preserved. This is highlighted in the lower part of the figure where the secondary structure is shown. Bases that are less likely to change are represented with warmer colors. Notice that in some cases the bases themselves, such as those in the loop on the right hand side, must not vary since these bases serve a function such as binding to a particular protein. This data comes from Rfam, a large protein family database [Burge12]. To get an idea of the size of the ncRNA homology search problem, consider that the IRE_I model shown here represents around 2,500 sequences across over 100 species. In the time since we developed our accelerator, over 600 redundant sequences have been removed from the IRE_I entry [Kalvari17], improving model accuracy and performance.

```
rainbow smelt   AUUCUUGCCUCAACAGUGAUUGAACGGAAC
red junglefowl  AUUAUC..GGGGACAGUGUUUCCC.AUAAU
human           UUUCCUGCUUCAGCAGUGCUUGGACGGAAC
gray wolf       UCGUUC..GUCCUCAGUGCAGGGC.AACAG
```

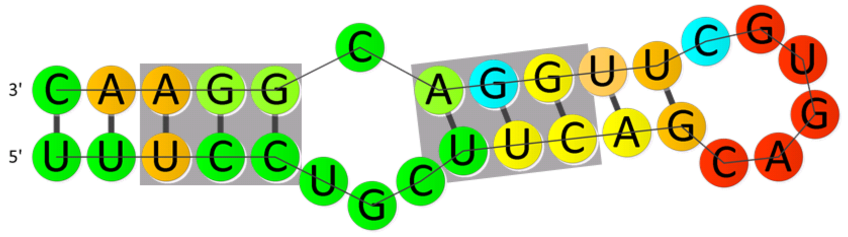


Figure 3.1 IRE ncRNA sequence in four species (above) and the IRE secondary structure with conserved pairings highlighted (below)

In addition to the regions of covariance, the missing bases adds to the complexity of any model of ncRNA families. In some cases, the exact length of a non-pairing region isn't important, and bases can be inserted or deleted freely. Any useful model must also be able to express this variable length and not assume any specific gap between regions or covarying pairs. Unlike proteins, where any significant change to the sequence will change the amino acids and therefor the complex molecular dynamics that lead to the functional tertiary structure, for most portions of an ncRNA, the pair bonding leading to the correct secondary structure is all that's important. Paired nucleotides are relatively inert, and only the specific sequence of highly conserved functional regions that interact with some other molecule matter [Nawrocki09b].

3.1.3 *Hidden Markov Models*

Because of the critical importance of secondary structure, the problem of identifying ncRNA homologs that belong to a family based on sequence alone is very difficult. The bases that form Watson-Crick pairs across all species are said to covary because for the secondary structure to be preserved these pairings must be as well. For example, in Figure 3.1 the 3rd position from the left and the 3rd position from the right covary and can change to potentially any pair without destroying the function of the ncRNA. The models of ncRNA families that capture this relationship are called Covariance Models (CM) [Eddy94], which will be discussed in the following section.

Protein homology search, because it only considers primary structure, does not need to consider covariance. For this reason, protein families are often represented as hidden Markov models (HMMs). HMMs, in the general case, are a statistical model used to determine probabilistic values for the hidden states in a Markov process. In these models there are some number of hidden states and some number of output variables. For each state there is a set of transition probabilities, from the state to other states and a set of emission probabilities, giving the possibility of producing each output for the given state. In HMMs relevant to protein and ncRNA family modeling, the emission probabilities are for each of the four bases and the transition probabilities are to specific groups of Match, Insert and Delete states, that represent matching an expected base from a model or having an insertion (which can loop) or a deletion (which skips the group and emits nothing). The details and value of this structure will become clearer in the following sections.

Software that scores protein sequences against Pfam, such as HMMER [Eddy98], does the scoring against an HMM built from the consensus protein sequence. When used in this way the HMM contains a probability for each position in the sequence to be each individual base (or amino acid in the case of proteins) as well as the possibility of a position being deleted or having something else inserted there, as in the groups described above. Depending on the specific application the model can also be more complex, which increases computation time but may be able to represent some families more accurately. Whatever states are added, however, HMMs are not capable of modeling the flexibility of the specific nucleotides in a given base paired position in an ncRNA model. Because of this limitation, Rfam makes use of both HMMs, similar to those used in Pfam, and a Covariance Model (CM) that can account for this pairing.

3.1.4 *Covariance Models*

Covariance models, used for ncRNAs, are conceptually similar to HMMs but contain more complex states and transitions that the algorithms used to score sequences against an HMM can’t handle. CMs were originally envisioned as “a generalization of hidden Markov models” [Eddy94] with support for covarying, or paired, bases capturing the conserved pairs that preserve RNA secondary structure even if the nucleotides themselves change. The states are similar, except for the addition of pairing. In particular, Match is replaced with Match Pair (MATP), Match Left (MATL) and Match Right (MATR), while Insert is replaced with Insert Left (INSL) and Insert Right (INSR). The Delete (DEL) state remains essentially the same, as does the concept of transition probabilities. Finally, CMs introduce an additional Bifurcation (BIF) state.

The following example illustrates the use of the various CM states. Using only the MATP state, it is possible to describe a secondary structure featuring a string of paired bases (Figure 3.2), known as a stem or hairpin. This is an important structure for RNA, including ncRNA, and is expressed by a series of MATP states where each state emits one paired base on each side of the stem. With only MATP states, this is the only possible structure, so MATP is not sufficient to express important ncRNA.

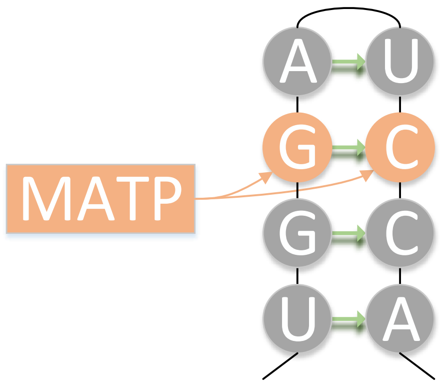


Figure 3.2 Stem secondary structure emitted by MATP states

It is common for a stem to form in such a way that one side has one or more unpaired bases, known as a bulge. This can be expressed with MATR states (as in Figure 3.3) or MATL states. It is also possible to have bulges on both sides of a stem, known as an internal loop.

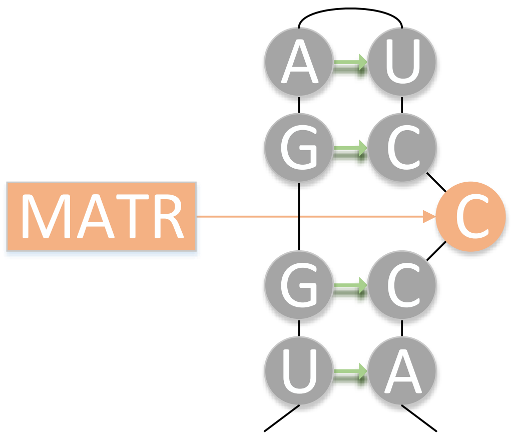


Figure 3.3 Bugle secondary structure emitted by MATR state

RNA stems are typically part of a more stable stem-loop structure, where there is a loop on the end of the stem, as in Figure 3.4. This loop is emitted from MATL states, which occur at the end of the node tree expressing the stem-loop.

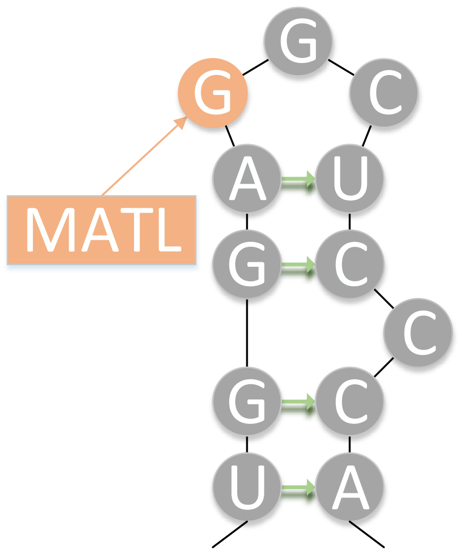


Figure 3.4 Loop secondary structure emitted by MATL states

The states described above are insufficient to model all cases of covariance. While the pairing of a single stem-loop structure is possible, multiple stem-loops, as are often found in ncRNA, are not. In particular, there would be no way to put MATL or MATR states between the pairs of the two stems that would allow their MATP states to specify the local pair, instead of the more distant one. Eddy describes this case as an ordered binary tree of nodes, but with no branching. To introducing branching in the tree, he introduces the Bifurcation state. This state, as used in Figure 3.5, creates branching in the tree and allows for representation of multiple stems. The BIF state, while necessary for many ncRNA secondary structures, is also extremely computationally expensive.

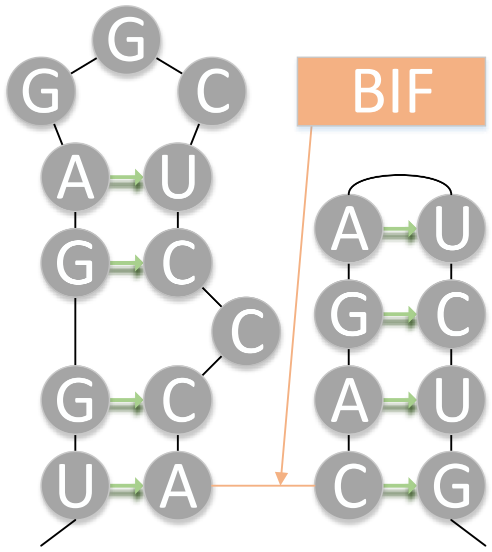


Figure 3.5 Multiple separate loops with an intervening BIF state

Using a tree of these states, with BIF providing branching, it is possible to model all important ncRNA families. Figure 3.6 contains the state tree for the above example, and the resulting secondary structure.

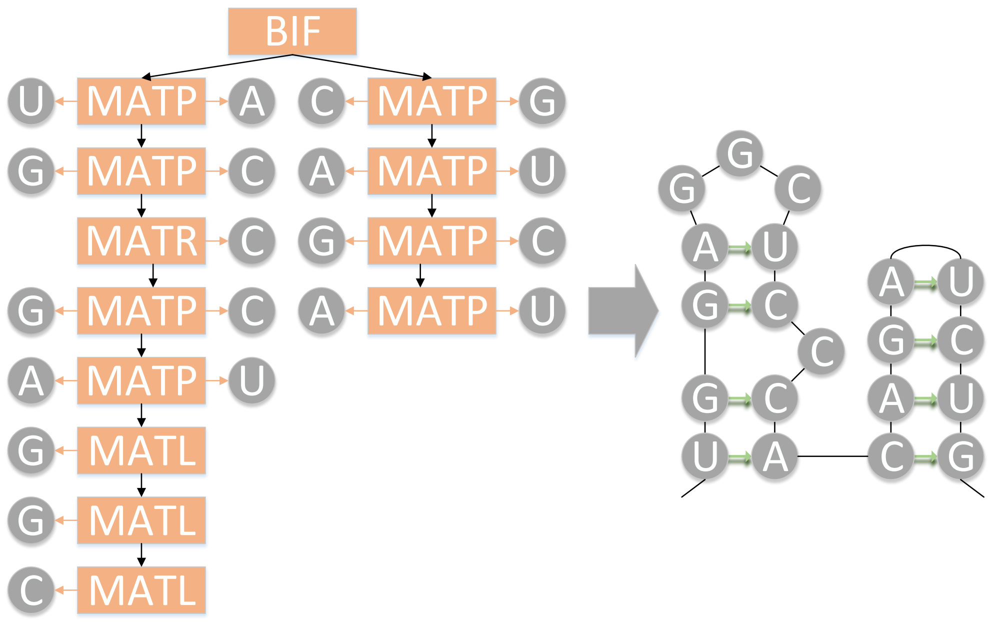


Figure 3.6 State tree and resulting RNA secondary structure

Up until this point we have discussed the states individually. In a full CM, the states are part of nodes, similar to the state groups of the HMM. These groupings are summarized in Table 3.1, and in more detail in Table 3 of [Eddy02].

Table 3.1 Covariance Model nodes and states

Node	**Split Set States** (exactly 1 visit)	**Insert Set States** (0 or more visits)
Match Pair	MATP MATL MATR DEL	INSL INSR
Match Left	MATL DEL	INSL
Match Right	MATR DEL	INSR
Bifurcation	BIF	

The first column of the table represents nodes. Eddy refers to a CM as “a binary guide tree of nodes representing the consensus secondary structure” [Eddy02]. The four nodes in the table can emit the bases of the ncRNA, and also include pairing information (hence the secondary structure). Each of these nodes contains the split set of states, and some nodes contain the insert set states. For a given structure generated or scored with the CM, exactly one split set state will be visited. For example, when scoring an ncRNA with the CM a single Match Pair node may visit the MATP state or the MATL state, but not both. The insert set states, on the other hand, may be visited zero times, if there is no insert, or as many times as inserts are required to generate the best score (which in practice limits the number).

The above example assumed the main state was used for every node, which Eddy calls the consensus state. This means that the Match Pair node will use the MATP state, and so on for the other nodes. In practice a Match Pair node could emit from a MATL or MATR state or many INSL or INSR states when scoring a sequence.

The emission probabilities are another important aspect of scoring a sequence against a CM, and here there are significant changes from HMMs. Instead of 4 possible nucleotides to emit (A, C, G and U) there are 24 to account for pairing (A, C, G, U on the left and right as well as all 16 paired combinations). Note that this allows the model to include pairing relationships aside from the traditional four, which can more accurately describe some ncRNAs. In particular, the G-U “wobble pair” is quite common, with some instances in the previous IRE_I example (Figure 3.1). A particular node in a CM will not necessarily emit all 24 possibilities. Instead, it may emit either a single base on the left or right (with MATL and INSL states, or MATR and INSR states, as well as the DEL state) or a base on both sides (the MATP state). In the case of the Match Pair node, all states except for BIF are present, as are all emissions.

Covariance models are powerful, but they cannot represent all RNA secondary structures found in nature. In particular, there are some interactions involving more than two bases, such as base triples, and covariance that is not simply nested, such as pseudoknots, which are not represented through the states available to covariance models. This is not a significant problem, since the models built with the available states are sufficiently powerful to describe and detect all ncRNA families [Eddy94].

3.1.5 *Cascading Filters*

The previous sections provide two methods of modeling ncRNA families: HMMs which are faster but unable to capture covariance, and CMs which are slower but more powerful. Weinberg and Ruzzo described a method where fast "rigorous" HMMs can be created from the CMs that represent an ncRNA family [Weinberg04]. These rigorous HMMs can be used as a filter to quickly identify many sequences that are not possible matches for the CM, while rejecting none that are. Later work describes a method to create a series of heuristic HMMs that lose small amounts of accuracy (typically $< 1\%$ for bacteria) but can be run orders of magnitude more quickly [Weinberg06].

The canonical software to perform ncRNA family scoring for Rfam is Infernal [Nawrocki09c]. Infernal makes use of the above filtering method. However, for large sequences homology search is a challenging software problem that can take a great deal of time to run. This time can be reduced with highly optimized code and many filtering stages, but the algorithmic complexity of a sequence scoring HMM is $O(N^2)$ and the complexity of a CM is $O(N^3)$. Further complicating matters, in addition to potentially using multiple HMMs, Infernal uses two algorithms, one faster and one more sensitive, to score sequences against CMs.

3.2 ALGORITHMS

Rfam 11.0, and thus our accelerator, primarily makes use of two sets of algorithms: Viterbi/Forward for scoring sequences against HMMs, and CYK/Inside for scoring against CMs. The first set of these (Viterbi and CYK) find the best score of a single path, requiring only integer arithmetic, while the second set (Forward and Inside) compute the sum across all paths. This gives potentially better results, although it requires floating point arithmetic, and is the preferred method for more recent versions of Rfam. These algorithms are discussed in the following sections.

3.2.1 *Viterbi Algorithm*

The Viterbi algorithm is a two-dimensional dynamic programming algorithm [Forney Jr73]. Given an observed sequence and an HMM featuring a set of states with emission and transition probabilities for those states, the algorithm finds the highest scoring path through the states for that sequence. This is the most likely path to have emitted the particular input sequence. Although the Viterbi algorithm itself is general and can be used with any HMM, the implementations used in HMMER and Infernal are limited to supporting only HMMs of a specific structure, called Plan 7 and Plan 9 respectively. Although they are very similar, there are some significant differences that affect the CM Plan 9 (CP9) FPGA design. The states of CP9 HMMs are shown in Figure 3.7.

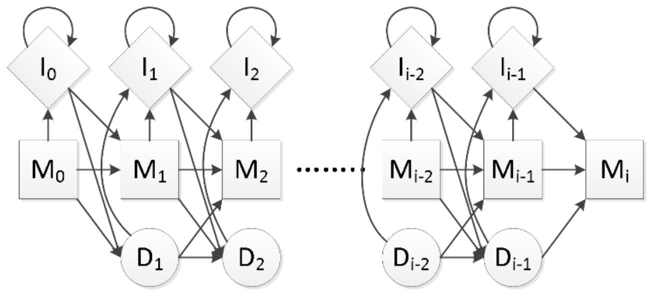


Figure 3.7 CM Plan 9 HMM states with transitions

In CP9, each node consists of three states: Insert, Match and Delete. The Match state emits a base that matches the one expected by the model. The Insert state emits an unexpected base, optionally advancing to the next node. Finally, the Delete state advances to the next node without emitting. Note that every node is the same except for the first and last. For the first node, there is no Delete state and M_0 is equivalent to a Begin state, which represents the start of the model. For the final node there is no Insert or Delete and M_i represents an End state. The CP9 HMMs are built by Infernal from each family's CM. This process is based on the techniques described in [Weinberg06] and creates a filter with a much faster runtime than the CM due to the better asymptotic efficiency of the Viterbi algorithm.

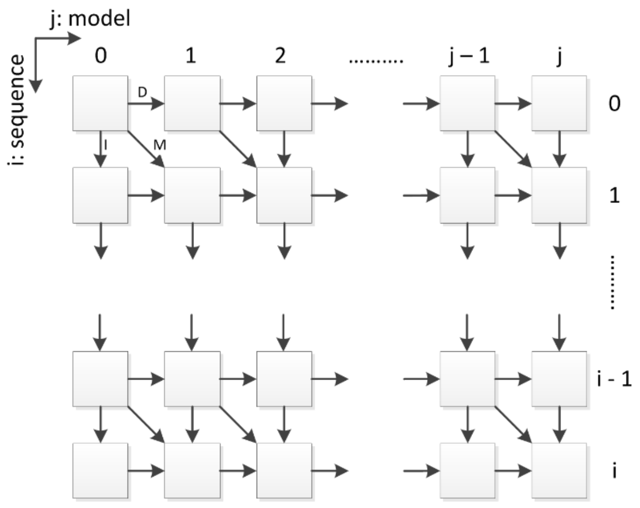


Figure 3.8 CP9 DP table data flow pattern in the sequence (vertical) and model (horizontal) directions

The Viterbi algorithm makes use of a two-dimensional dynamic programming table to store the log-probability scores computed for each state (Figure 3.8). The work for each node involves computing the log-probability of each transition into that node given the current sequence element and the scores of previous nodes. For example, to compute the Insert score for some node requires taking the sum of the Insert emission log-probability for that state, given the current base in the sequence, and the maximum of the scores of the three states that could transition into an Insert. Each of these scores must have the correct transition log-probability added to it before the max is computed. The final equation for an Insert score is as follows:

$$\text{Insert}(j,i) = \text{emit\_prob}(\text{Insert}_j, \text{Seq}_i) + \max\begin{cases} \text{Insert}(j, i-1) + \text{trans\_prob}(\text{Insert}_j \rightarrow \text{Insert}_j) \\ \text{Match}(j, i-1) + \text{trans\_prob}(\text{Match}_j \rightarrow \text{Insert}_j) \\ \text{Delete}(j, i-1) + \text{trans\_prob}(\text{Delete}_j \rightarrow \text{Insert}_j) \end{cases}$$

During the Viterbi run, the table cell j,i contains a score representing the log-probability of the sequence up to some element *i* matching the model up to some state *j*, and by the end of the run the entire table will be filled in this manner. The equations for the remaining transitions are very similar to those used in Plan 7 and can be found in the previous work [Oliver08]. The most significant exception is the additional Delete to Insert transition.

One dimension of the dynamic programing table is the model, so moving one cell to the right in Figure 3.8 is equivalent to moving to the next state in the model. The other dimension represents the sequence being scored, so moving down the table represents emitting one base. Each cell in Figure 3.8 represent an entire node, meaning that they each contain a Match, Insert and Delete state. The Delete state does not emit a base; instead it represents a state in the model that is not found in the sequence. For this reason, Delete is a transition from one node to its right hand neighbor in the table. Similarly, Insert emits a base that does not match any model state, so it moves down the table. The Match state is used when a base fits the model's expectations. In this case it both emits the base and moves on to the next model state, moving right and down one node in the table.

These data dependencies are such that the CP9 HMM has no backwards paths. This means that a computation can be performed along a wavefront starting with node (0, 0) (which depends on no other nodes) followed by computing (1, 0) and (0, 1) in parallel, then (2, 0), (1, 1) and (0, 2) also in parallel and so on. In other words, the lack of backwards data flow means that there is a huge amount of parallelism available in the computation. Note that communication is not all local from one node to the next. Not shown here are the Begin and End state transitions. These provide some (typically very low) probabilities to jump from the Begin state to any position in the model, and similarly from any state to End. Although these transitions are not local like the transitions in Figure 3.8, they are still entirely feedforward, so no parallelism is lost.

With these algorithm details established, we should consider the changes from Plan 7 (used in HMMER for Pfam, and the many HMMER accelerators) and CP9 (used in Infernal for Rfam, as well as our accelerator). The first key difference between Plan 7 and CP9 is which state transitions are available. In Plan 7, there are no transitions from Insert to Delete or vice versa. Supporting these edges requires slightly more complex PEs in CP9.

The fact that there are no feedback paths in CP9 HMMs is the second critical difference between CP9 and Plan 7 HMMs. The feedback path in Plan 7 allows for the model to match multiple copies of itself in succession [Takagi09]. This feedback requires special consideration for an FPGA implementation, and so the lack of it in CP9 gives us greater design freedom.

Among the changes from Plan 7, the third and most significant is the introduction of another state type. This End Local (EL) state only exists for some nodes along the model, and allows for large portions of the model to be bypassed (Figure 3.9). Because EL states also have a self-loop, like Inserts, they allow for a number of bases to be emitted while skipping over a potentially large portion of the model. More specifically, when the CP9 HMM is built from the CM, some nodes will contain an EL state and a probability to transition from Match to this state. The only transition out from the EL state is to some subsequent Match state. This transition is unique in CP9 in that it is the only transition other than from Match to End that can jump forward many states. It is also different from other states in that not every Match has an associated EL state. In addition, a single Match can have multiple incoming EL transitions. See Figure 3.9 for an example of some of these properties. As shown below, EL states add complexity to the FPGA implementation.

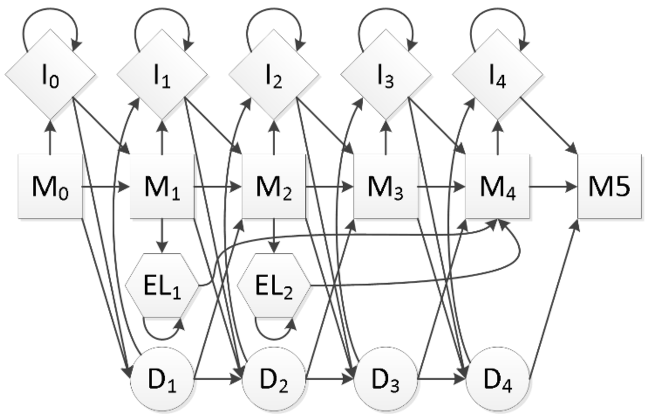


Figure 3.9 Example of a CP9 HMM featuring EL states

3.2.2 *CYK Algorithm*

To achieve high performance, Infernal's HMM filter stage removes the non-local dependencies in the ncRNA structure and simply looks at features detectable in a linear search. While this ignores important elements of the structure, the filter runs quickly and can eliminate most of the input data. The filter is tuned to provide a lower bound on the match likelihood score of matching sequencing. Thus, the HMM will accept many regions that do not actually include the target ncRNA. Further filtering requires looking at the secondary structure of the ncRNA model for which we are searching, and is performed by the Cocke-Younger-Kasami (CYK) algorithm [Younger67].

CYK is a dynamic programing algorithm, in some ways analogous to a three-dimensional version of Viterbi. Also like Viterbi, CYK is a general algorithm that can be used to parse any probabilistic context-free grammar in Chomsky normal form. CMs fit this description, as they are constructed using only a few rules of the appropriate form [Eddy94]. The CM directly represents the pairings and branchings found in the secondary structure of an ncRNA. This is done through

Match Pair and Bifurcate nodes respectively. As can be seen in Figure 3.10, Match Pair nodes represent pairings of potentially distant bases in the ncRNA sequence, something the HMM model cannot support. For example, in Hammerhead_3, the second and second-to-last bases are emitted by the same Match Pair node. Bifurcation nodes represent branches in the ncRNA secondary structure. Handling Bifurcations will require breaking the problem up into two smaller subproblems (one for the left branch and one for the right branch). Unfortunately, since each subproblem can use varying lengths of the target RNA, given that biological evolution can change the length of individual branches, we cannot know ahead of time where the split point occurs in the candidate sequence. This is solved by simply trying every possible split-point and picking the highest scoring match, a fairly time-consuming process. In Figure 3.10, Bifurcate nodes are found before the sets of Match Pairs that make up Hammerhead_3's two arms. This requires a Bifurcation state, because without one there would be no way to express two separate helices, or sets of paired bases.

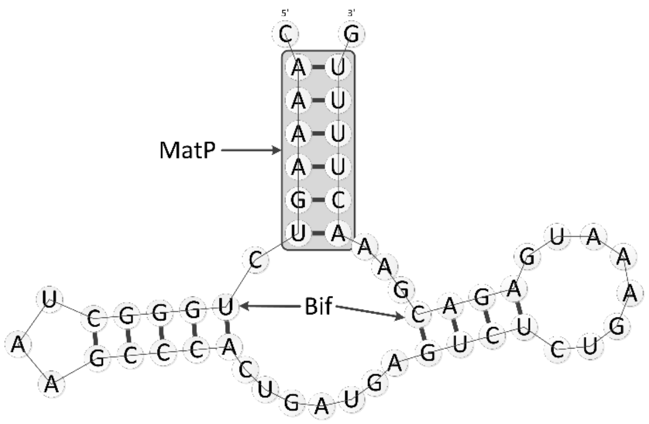


Figure 3.10 Secondary structure of a Hammerhead_3 ncRNA, where Bif indicates a bifurcation and the MatP region indicates required matched pairs

The CYK algorithm starts with a sequence of length N and a CM, and scores that sequence against the model, with a better match resulting in a higher score. It converts this into a three-dimensional dynamic programming problem by creating layers of triangular DP matrixes, one for each state in the CM as in Figure 3.11. These matrixes are triangular because position *j*, *i* in state S represents the best matching of states S…S_{End} to region *i*…*j* of the target sequence. It does not make sense to match to a region whose start is after its end, so entries *j*, *i* where $j < i$ are useless. Emissions of single bases are achieved through Match Left and Right states, where Left states move up one cell in the DP matrix by subtracting one from *i* as shown in Figure 3.11. Similarly, Right states move right one cell by adding one to *j*. The final dimension, *k*, refers to the state.

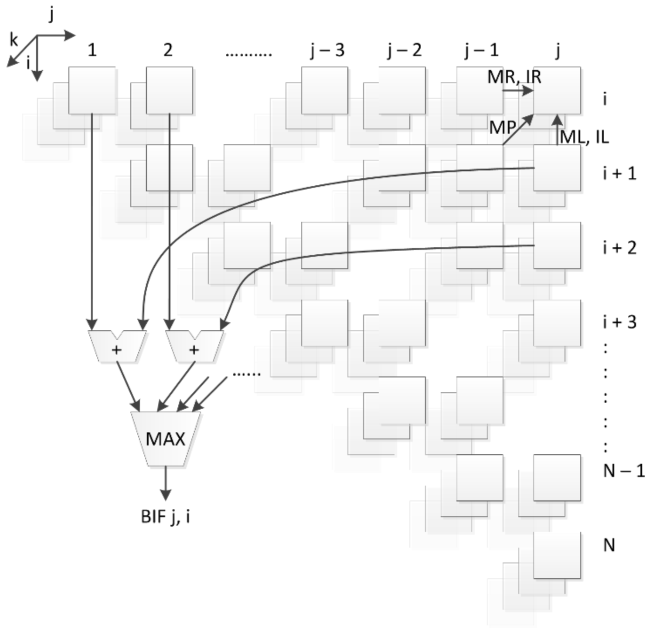


Figure 3.11 CM CYK three-dimensional DP table, featuring BIF (bifurcation), MR/IR (match/insert right), MP (match pair) and ML/IL (match/insert left) transitions

As the CYK algorithm proceeds, it works from the last CM state S_{End} back to the starting state S_{Root}, and processes from small sequences of length 1 (i.e. squares *i*, *i*+1) towards longer sequences. The probability score for each subsequence is stored in the DP table to be reused without being recomputed. This process handles most parts of the model (including Match Pair) very efficiently, with only local communication. The one exception is the Bifurcation state. In this state, processing a sequence from *i* to *j* requires finding the best split point, mid, where $i < mid < j$. Since we don't know where the bifurcation point is in the target sequence, CYK must compute the highest score for all possible mids by maximizing the score(*i*, *mid*, branch_1) + score(*mid*, *j*, branch_2) where branch_1 and branch_2 are the states containing the total scores for the subtrees. The first two cells of this calculation are shown in Figure 3.11 and the equation can be found in [Eddy94]. This computation is very similar to matrix multiplication and requires $O(n^3)$ arithmetic operations for each Bifurcation, resulting in a total CYK runtime on the order of $O(hn^2 + bn^3)$ where h is the number of non-bifurcation states, b is the number of bifurcations and n is the sequence length. Because of the expense of this operation, it can dominate the computation time for very large models (Figure 3.12). In practice, the impact of Bifurcation states is not always as extreme as this chart makes it appear because over half of all Rfam CMs are under 100 nodes long.

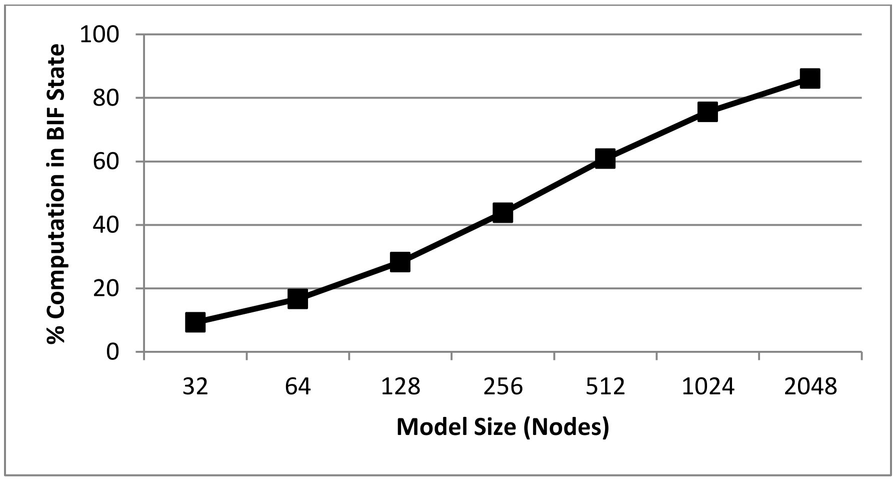


Figure 3.12 Computation in BIF state assuming 0.9% BIFs

3.2.3 *Forward and Inside Algorithms*

As mentioned previously Forward and Inside are more powerful, but also more computationally expensive, versions of Viterbi and CYK. Instead of identifying the single most likely (highest scoring) path through the model, these algorithms find the joint probability of all paths for a given input sequence; alternatively viewed as the probability that the model describes the input sequence. Because transmission and emission probabilities are stored as log likelihood ratio scores, and the max operation simply selects one of these probabilities, the implementation of Viterbi used in HMMER2, Infernal 1.0 and our work stores these as integer log-odds scores with a fixed unit size, as described in [Eddy08]. In other words, instead of multiplying the probabilities we add the logs. For Forward, this is not possible due to the requirement to sum the scores, which is an expensive operation on values stored as logarithms. As part of his work accelerating Forward for HMMER3, Eddy also identifies the danger of underflow for low probability paths when using a fixed representation [Eddy11]. Given that, the full Infernal 1.0 pipeline does include an Inside stage, this issue is discussed further in Section 3.4.3.

3.3 FPGA Implementation

The following sections describe the FPGA implementation of our Viterbi and CYK accelerators. Both make use of a linear array of PEs to compute large swaths of their respective DP tables in parallel. Our Viterbi accelerator fully supports EL states while the CYK accelerator is optimized to reduce the computational overhead of Bifurcations.

3.3.1 *Viterbi Implementation*

Figure 3.13 shows a block diagram of our CP9 Viterbi implementation. For all of the results presented in this and subsequent sections, our FPGA designs targeted a Pico Computing system featuring an EX-500 backplane equipped with an M-503 module. This module features a Xilinx Virtex-6 LX240T as well as 8GB of DRAM and 27MB of SRAM. The design operates on a streaming paradigm. This is appropriate because a single, relatively small HMM may be run against a very long sequence or set of sequences. The sequence is streamed in from off-chip and scores are streamed out on a similar channel. No other external memory or off-chip communication is used.

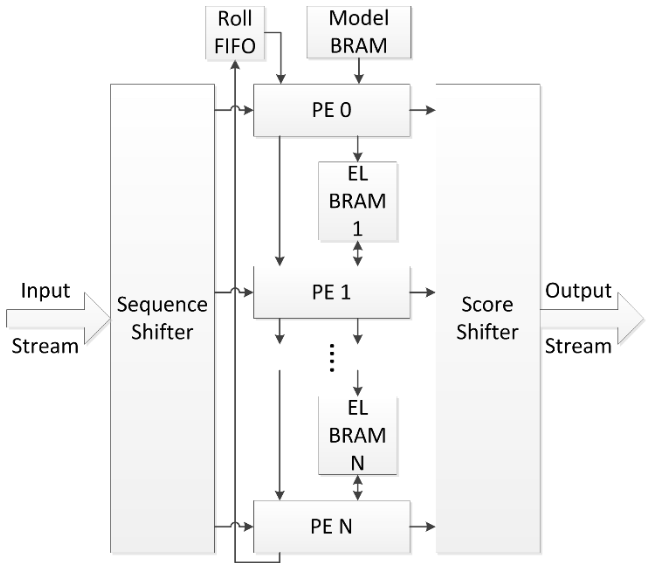


Figure 3.13 CP9 Viterbi FPGA Block Diagram

The CP9 Viterbi FPGA design is based on a linear array of PEs. Once the processor pipeline has filled, this allows all PEs to operate in parallel along the wavefront described previously. Given the dynamic program table in Figure 3.8, it makes the most sense for a single PE to either move down a column, computing scores for many bases from the sequence for a single state, or across a row, handling all states for a single base of the sequence. Either of these arrangements allows for neighbors to communicate model and score values. Because CP9 does not have a feedback path, but does have EL states, it is most efficient for a single PE to handle all states for a given base. The next PE handles the next base, and so on down the sequence as shown in Figure 3.14. Every PE will be computing at all times except for the beginning and end of the sequence. There are some cases that break this parallelism, for example, a very short model with fewer states than there are PEs. However, this situation can be handled by replicating the model multiple times and shifting in the next base after reaching the end of the first copy of the model. Solutions like this work because CP9 Viterbi scores can be local with respect to the sequence (and also the model, with the use of EL states) [Nawrocki09a].

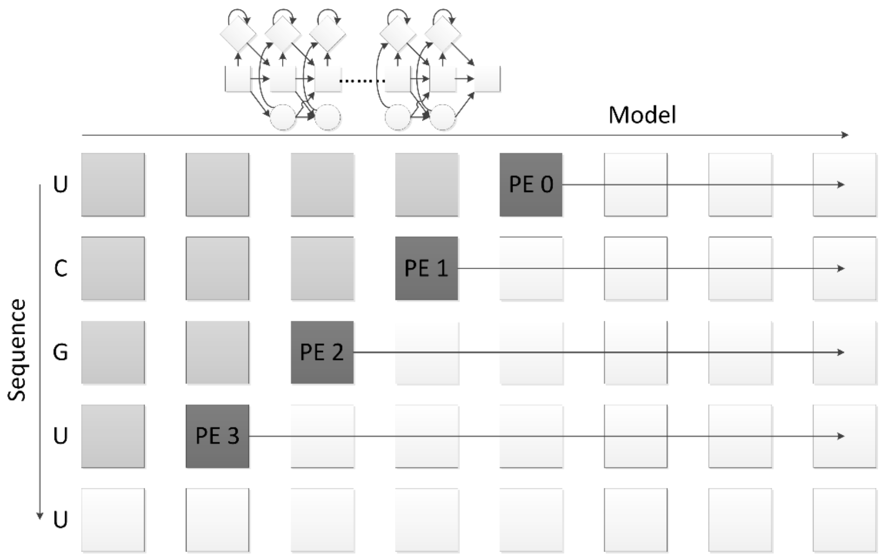


Figure 3.14 CP9 Viterbi PE allocation and wavefront, for a system with 4 PEs.

The Sequence Shifter itself is simply a set of shift registers. The first shifts bases in from the input stream and loads them into a second shift register when required. After a PE has completed work on a base, a new base is loaded from the PE's second shift register. The Score Shifter operates on similar principles and also filters output scores to avoid exceeding the available output bandwidth.

The Model BRAM stores the CP9 HMM parameters, which include emission scores for Insert and Match states and transition scores for all possible state transmissions. These scores are stored as the logarithm of an odds ratio measuring the likelihood of a given event, known as log-odds form. This form converts multiplications to addition of precomputed logs. Viterbi also requires maximum operations, which can use the same hardware in log and regular integer format. In addition, the precision of the calculations can be adjusted by changing the number of bits used to store the scores throughout the system. The amount of BRAM required to store the models depends on this width. Every state requires an emission probability for each base for Match and Insert, or 8 emission probabilities per state. There are transmission probabilities between each of the three states in the nodes, for a total of 9 different values. In addition, there are transition probabilities for Begin, End and EL for each state, bringing the total up to 20. Letting w_s represent the number of bits per score, this requires $20 \times w_s$ bits per state. Setting the maximum model length to 2k, which can handle all of the current Rfam models, and given that the Virtex-6 series features 36 Kb BRAMs, this means that the total model BRAM requirement is $2k \times 20 \times w_s / 36k = 1.25 \times w_s$. For our current system, a w_s value of around 18 bits is sufficient, requiring about 24 of our FPGA's 416 BRAMs for model tables, due to padding.

The Processing Elements themselves are fairly simple. The first PE receives model information from the BRAM. Since the computation wavefront, shown in Figure 3.14, is such that each PE is always one node in the HMM behind the previous PE, this model information can be passed from one PE to the next as well. Dark cells with PE labels represent current computation while the grey cells to their left have already been scored. The alternative approach, where each PE handles many sequence bases for the same state, requires that each PE have access to its own portions of the table, potentially reducing efficiency of BRAM use or introducing additional multiplexers.

Besides the transition and emission probability and sequence data discussed above, the PEs need some way to access and update the DP score table. Because all communication is either between adjacent bases or across many states but within the same base, there is no need to actually store the entire table in memory. Instead, the values that are required at any given time are only those at the computation wavefront. Values that do require longer communications, such as Begin and End scores (with an End score for each base being the ultimate output of the FPGA accelerator) can be stored in the local PE. The only exception to this local communication is the EL state, which we will discuss. Since the input sequence proceeds down the PEs in order, the next base after the final PE has to be handled by PE 0. At this point, all score information must be transferred back to PE 0. For models longer than the number of PEs, PE 0 will still be busy computing its current base, so this score information must be stored in the Roll FIFO shown in Figure 3.13. The total amount of score information required for each base is one score each from the Match, Insert, Delete and EL states, or $4 \times w_s$. There is also an EL BRAM (not shown) for PE 0 located after the FIFO.

The PEs themselves are simple pipelined arithmetic units. They perform the addition and max operations required to calculate the scores for the DP table cells they are processing before moving on to the next cell in the next cycle. It is worth noting that although the current version with a two stage pipeline can run at slightly over 100 MHz, it is challenging to add more pipeline stages. This is due to the various local dependencies for individual HMM states like EL and End. In addition to computing table values for the current cell, the PE stores the running best scores for the current base for long transitions. These include scores for EL, Begin and End.

3.3.1.1 EL State Combination

EL states are a special case for a number of reasons discussed earlier. To summarize, EL states introduce data dependencies between nonadjacent cells. EL states also have a self-loop with a score penalty that transitions to the next base similar to an Insert state. In order to handle these irregularities, we introduce the EL memory. This memory is necessary because unlike the End state, storage for many EL scores could be required simultaneously. See Figure 3.9 for an example of how this could happen. In this case, a PE computing M_4 requires data from both EL_1 and EL_2. More complex models require storage for more ELs, and there are some situations with much greater EL fan-in. Although these situations are handled with a simple loop over the EL input states when computing a Match score in the Infernal software, that luxury is not available for an FPGA implementation. Instead, our design handles this problem by combining ELs in a tree-like structure while running along the model. The shape of this combination tree is computed in software and stored with the model information prior to a Viterbi run. This is possible because EL states are typically generated from states that pair in the CM, and physical limitations on the molecule shape prevent overly complex EL patterns. In fact, for many CP9 HMMs a stack would be sufficient for EL storage. Our algorithm for EL combination is presented in Figure 3.15.

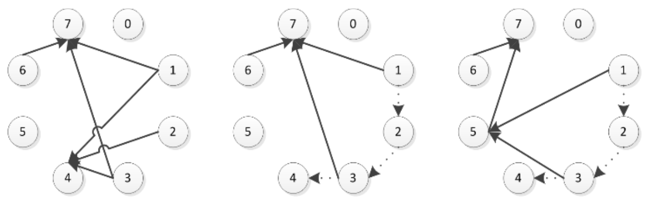
```
combineELs(ELMap)

Input: ELMap, containing list of all EL transitions
Output: ELCmd, containing EL memory and register commands for each CP9 HMM state

Convert ELMap to ELCmd equivalent (EL memory ops)
foreach link l in ELMap
    if l has only one destination state and
    if next l has only the same destination state
        remove l's EL memory ops from ELCmd
        add equivalent EL register op to ELCmd
foreach command c in ELCmd
    while c has reads from > 2 EL memory locations
        let wl be the origin state of data for reads in c
        sort wl by state
        foreach HMM state s starting at second state in wl
            if s has no EL memory ops (register ops are OK)
                add two reads (based on order in wl) to s
                remove those reads from ELCmd for c
                add write of combined data to ELCmd for s
                add a read of combined data to ELCmd for c
                terminate foreach
            if s = state of c
                insert dummy state after second state in wl
                terminate foreach
```

Figure 3.15 EL combination algorithm for CP9 HMMs

The EL combination algorithm above is capable of giving all states a maximum of one EL memory write and two EL memory reads, in addition to the use of an EL register inside the PE. In the FPGA, this translates to a pair of mirrored BRAMs for a total of one write port and two read ports. For an example of the algorithm's operation, see Figure 3.16. A represents the initial state after software EL states are converted into EL memory commands. States 4 and 7 both require three reads, so this is not a legal set of commands for our hardware. In B, after the EL register conversion has taken place, the extra reads on state 4 have been resolved. The register operations automatically combine EL scores and are represented with a dotted line. Finally, in part C, the first two EL reads in state 7 are reduced to a single read by doing an EL combination in state 5. Now no state has more than two reads and one write, resulting in a legal set of EL commands.



A. Initial Model

State	Cmd
1	W0
2	W1
3	W2
4	R 0, 1, 2
6	W3
7	R 0, 2, 3

B. After Register

State	Cmd
1	W0, Reg
2	Reg
3	W1, Reg
4	Read Reg
6	W2
7	R 0, 1, 2

C. After Combining

State	Cmd
1	W0, Reg
2	Reg
3	W1, Reg
4	Read Reg
5	R 0,1 W0
6	W1
7	R 0, 1

Figure 3.16 EL combination algorithm example

Currently, two BRAMs per PE is not a limiting factor, as can be seen in section 3.4.1, but if it becomes an issue in the future it would be possible to double-pump the BRAM ports and use a single BRAM for each PE. Finally, although it is possible to conceive of a scenario where the dummy states inserted by the combination algorithm could have a substantial impact on runtime, or even make an HMM too long to fit in the Model BRAM, we have not encountered any Rfam families that require even a single dummy state.

3.3.2 *CYK Implementation*

Our implementation of CYK for CMs, diagramed in Figure 3.17, is based around a linear array of PEs, similar to the one used for Viterbi. The most significant difference in the design is the use of off-chip memories. Because it has a three-dimensional DP table, CYK requires substantially more storage than Viterbi. By dividing the entire matrix into stripes and letting each PE handle a single column of the DP table as in Figure 3.18, most of this memory can be made local to the PE in the form of BRAM [Xia10].

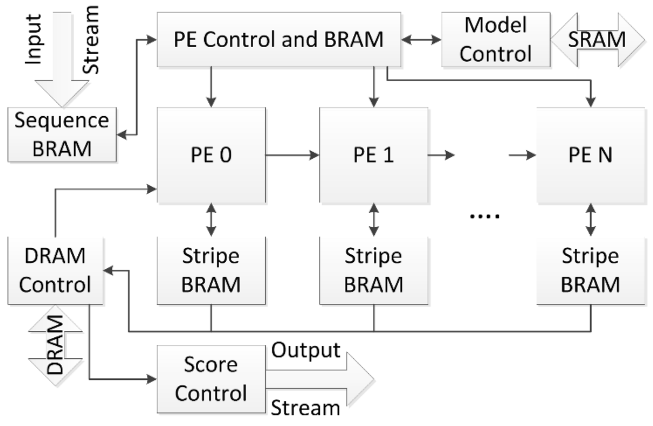


Figure 3.17 CM CYK FPGA implementation block diagram

This method of striping the model is possible because, much like Viterbi, there are no non-local or backwards data dependencies for non-Bifurcation states. The calculations for a given cell in a non-Bifurcation state only require data from the same cell and its immediate neighbors from layers in the previous node. This means the computation wavefront is similar to Viterbi, and a similar arraignment of PEs inside the stripes is most effective. After completing one state for a given stripe, the PEs can begin work on the next layer. After completing all layers, the PEs begin work on the next stripe until the computation is complete. Each PE contains many simple arithmetic units to update all states in the node simultaneously. This requires striping the state data across many local BRAMs.

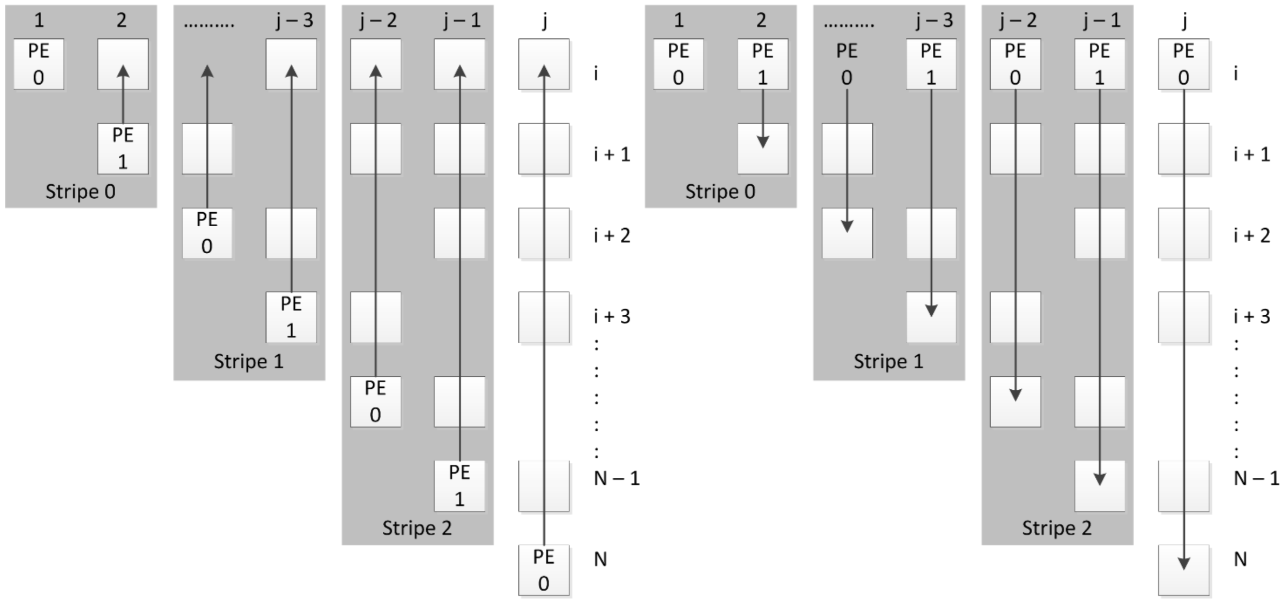


Figure 3.18 Striping for Normal (left) and Bifurcation (right) states

All of the DP cells for the current stripe can be stored in the local BRAM associated with each PE. Values from cells to the left can be passed from the previous PE. The values computed at the end of a stripe are written out to DRAM, to be reloaded when the same layer is reached during the next stripe. The only other time that DRAM storage is required is for one of the Begin layers prior to a Bifurcation state. Because the Bifurcation computation uses non-local data, it requires substantially more cells from DRAM. Figure 3.18 shows that PE operation is substantially different for Bifurcations and this requirement explains why. The row data used in the Bifurcation computation discussed in the previous section is streamed in from DRAM and passed from one PE to the next. Column data is loaded from BRAM as with other states. Every PE is equipped with multiple Bifurcation arithmetic units allowing for the use of multiple DRAM rows simultaneously, improving use of memory bandwidth for Bifurcation states. This can be seen in Figure 3.19.

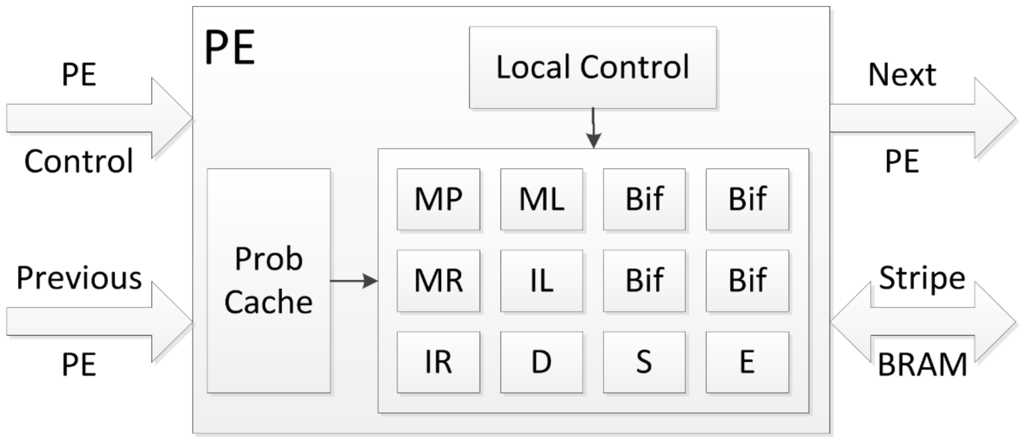


Figure 3.19 CM CYK PE block diagram

Although the striping pattern is similar to [Xia10], the novel aspect of this design is the parallel computation of the scores for an entire node. Eliminating the need of a layer in the DP table for each state (the standard implementation described in [Eddy02]), yields a substantial performance advantage for non-Bifurcation nodes. The parallel Bifurcation units in the PE confer a similar advantage for Bifurcation states. Increasing performance and memory operations, while keeping BRAM requirements constant, also results in better DRAM bandwidth utilization than [Xia10]. Although this requires a larger PE, FPGA logic is not the limiting resource, as explained below. Our design also precomputes all memory addresses and other information required to access the model and DP table data and stores this information in SRAM, resulting in an extremely simple control structure. The resulting savings help to offset the increased PE logic.

The CM model itself can be much larger than the Viterbi model, and BRAM is the limiting resource for this design, so the model is stored off-chip. BRAM becomes the limiting factor because it would be impossible to achieve high performance if the data required by all stripes was loaded from DRAM. Our M-503 platform offers two 64-bit DDR interfaces running at 400MHz for a total of 12.8 GB/s of DRAM bandwidth. Ignoring the bandwidth required to load the data along the first row, each PE requires as many as four DP table values each cycle for Match Pair, Match Left, Match Right and Delete states. This means that again assuming a score width of w_s the bandwidth required for a single PE is 4 × FPGA speed × w_s. Using w_s of 21 bits [Moscola10], slightly more than Viterbi, and a speed of 100 MHz, this gives a requirement of about 1 GB/s of bandwidth for each PE. This result would limit our design to 12 PEs on the M-503.

When using the striped approach with BRAM there is sufficient DRAM bandwidth available and BRAM capacity becomes the limiting factor. Assuming a maximum sequence length of 6144 bases, reasonable given the length of CMs in Rfam, the BRAM requirement for the largest stripe becomes $4 \times 4096 \times w_s$. Again using 21 bits, this yields a storage requirement of 504 Kb. Given the Virtex-6's 36 Kb BRAMs, this would require 14 BRAMs for each PE, allowing 24 PEs per FPGA after accounting for other BRAM usage. This solution is clearly superior, with twice as many PEs on-chip compared to a design that stores the entire table in DRAM. The ability to scale memory access bandwidth with faster PEs is another advantage to striping with BRAM. If all stripe data was loaded from DRAM, increasing PE clock speed would increase the required DRAM bandwidth of each PE proportionally thus allowing for fewer total PEs.

Another way the CYK design differs from Viterbi is in the input sequence handling. For Viterbi, the sequence can be streamed in, sent to the PEs and then discarded. Unfortunately, this is not the case for CYK. Because the use of stripes results in each stripe requiring the same parts of the sequence, the entire sequence must be stored for reuse after being streamed in. The sequence BRAM, shown on the block diagram, serves this purpose.

The last significant CYK module is the Controller. This module contains the Finite State Machine (FSM) responsible for the CYK algorithm operation. The Controller also contains BRAM that stores model-specific instructions. These consist of a number of fields generated by software and written to the FPGA prior to accepting sequence input. Entries in this table include DRAM and SRAM addresses and lengths for each stripe as well as state and other model information for the stripe. The CM itself can be large and is stored in off-chip SRAM. The FSM currently follows a simple schedule of loading the stripe information from the controller BRAM, loading appropriate DRAM values and then executing to completion while writing values back to DRAM. Future versions of the design could feature a more complex controller that performs some of these operations in parallel.

3.4 FPGA RESULTS

This section discusses the results of the individual accelerators as well as calculating estimated speedup for a full system.

3.4.1 *Viterbi Results*

Running at 100 MHz, the CP9 Viterbi implementation is limited by the available logic resources on our Virtex-6, as seen in Table 3.2. This is the expected result, given that each PE has many adders and max units. The current PEs make use of parallel maxes, a poison bit for overflow and other frequency optimizations that result in larger PEs. Determining the optimal tradeoff between frequency and quantity of PEs on an FPGA remains in the domain of future work.

Table 3.2 Post-map logic utilization of Viterbi design

PEs	Slices	LUTs	BRAMs
4	14%	9%	13%
16	27%	15%	19%
64	67%	43%	42%
128	97%	77%	73%

The performance of the FPGA-based solution is very good when compared with Infernal. Figure 3.20 shows the speedup vs. Infernal 1.0 software running on a single Intel Xeon E5520 core and compiled using GCC -O2. Adding PEs to the system results in nearly linear speedup, and this is to be expected given that there is no significant overhead. Speedup over Infernal 1.0 decreases as model size (shown in parentheses) increases, leveling out at around 230x. This is most likely due to inefficiencies in the software Viterbi implementation for very small models. Note that the median HMM model length in Rfam is under 100 states. This chart compares only the runtime of the Viterbi algorithm itself, not the entire FPGA and software systems.

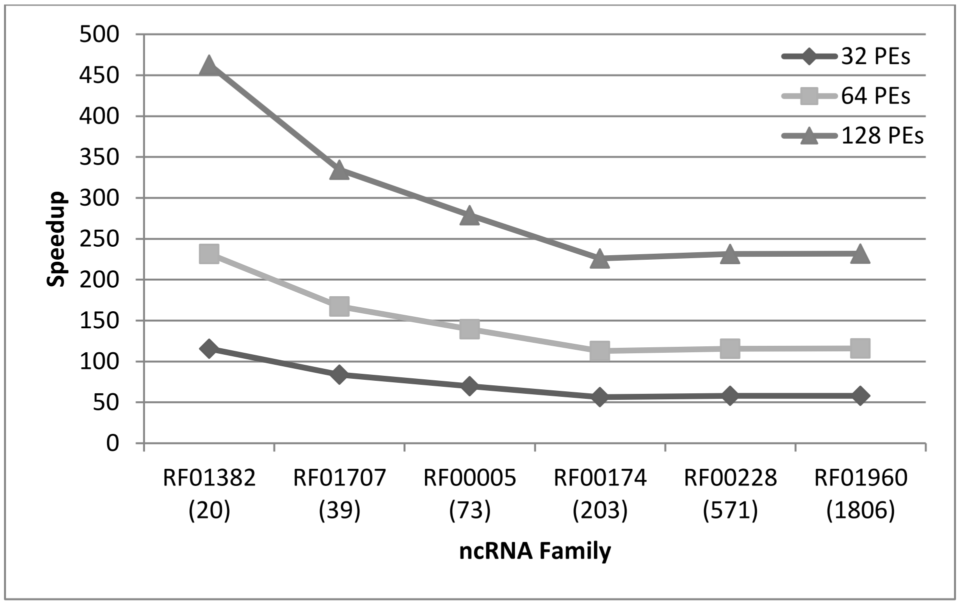


Figure 3.20 Speedup of FPGA Viterbi vs. single CPU Infernal

3.4.2 *CYK Results*

Running at 100 MHz, the FPGA CYK implementation is limited by BRAMs available on chip, as seen in Table 3.3. This is the expected result given the discussion of BRAM utilization in the previous section. Future designs may contain additional parallel logic in an attempt to achieve better performance with the same memory utilization.

Table 3.3 Post-map logic utilization of CYK design

PEs	Slices	LUTs	BRAMs
6	45%	29%	35%
12	54%	36%	56%
24	61%	42%	96%

CYK performance shows much more unpredictable variation between different CMs than Viterbi, as seen in Figure 3.21. This is because the CYK algorithm's runtime depends on a number of factors other than model size, including the prevalence of Bifurcation states. In addition, the speedup from adding PEs is sublinear for some models due to the overhead of switching between stripes and computing scores for cells in the stripe that are not actually part of the DP triangle. This effect is substantially more pronounced in very small models because the average height of a stripe is much less. Overall speedup is still good, with a geometric mean speedup of over 60x for 24 PEs.

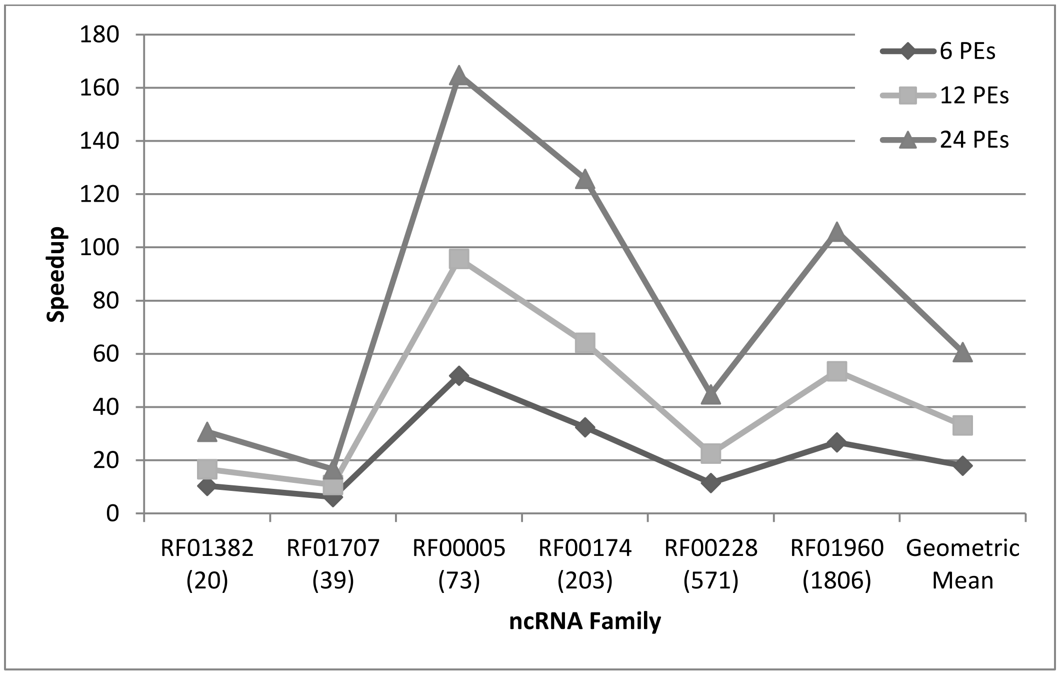


Figure 3.21 Speedup of FPGA CYK vs. single CPU Infernal

3.4.3 *Full System Estimates*

To get an estimate of the performance of an accelerator for Infernal's final search algorithm, Inside, it is necessary to determine the number of Inside PEs that would fit on our FPGA. The large increase in size from a CYK PE to an Inside PE is due to the use of floating point (FP) addition. As discussed in section 3.2.3, the CYK and Viterbi implementations avoid FP arithmetic and multiplication by using log-odds scores where those operations become integer additions. However, Inside requires addition of scores, instead of only multiplication, which is expensive in the logarithmic representation. For FPGAs, single precision FP addition requires approximately 20x more area than log multiplication (integer addition), which is still only half as expensive as log addition [Haselman05]. Based on these numbers, and our calculations showing that the CYK design becomes FPGA logic limited at around 80 PEs (logic, not BRAM, will limit Inside), a very conservative estimate for Inside would allow for 4 PEs and one sixth the speedup of CYK. This estimate is very likely excessively pessimistic because the Infernal 1.0 implementation of Inside is also slower than CYK.

Although the performance of the individual FPGA implementations of Viterbi and CYK is important, the ultimate goal of this work was to accelerate Infernal 1.0 in a way that is useful to biologists and others in the field. To measure the progress made towards this goal, it is valuable to look at the performance of Infernal 1.0 as a whole. Infernal performance using the FPGA runtimes for Viterbi and CYK are shown in Figure 3.22. Although no novel techniques for integrating the accelerators into an FPGA system are currently modeled, one advantage of a system that uses FPGA implementations to accelerate Viterbi and CYK is the possibility to reconfigure a single FPGA to run both filters if multiple FPGAs are not available. A multi-FPGA system could divide FPGAs between Viterbi and CYK as required given that some ncRNA families spend more time in one algorithm or the other, as seen in Figure 3.22. This is due to a number of factors, including the filtering fraction achieved by each filter and the asymptotically worse runtime of CYK, which begins to dominate for larger models. The ability to reconfigure the same hardware and allocate resources as required for a wide variety of genomics applications gives the FPGA implementation an advantage over potential ASIC designs like [Moscola10]. Finally, many bioinformatics laboratories already have FPGA platforms installed or make use of a cloud FPGA platform.

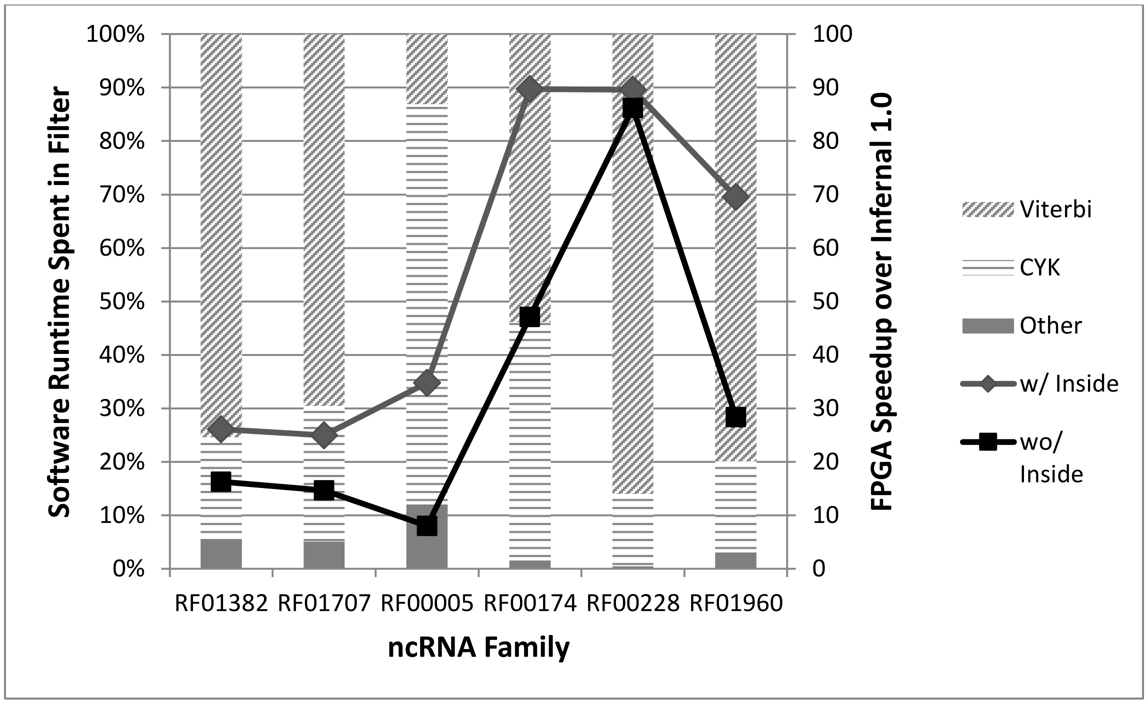


Figure 3.22 Estimated Infernal 1.0 FPGA system speedup

The speedup here is less than that of the individual accelerators - particularly without Inside acceleration - for two major reasons. First, without an accelerated version of the Inside algorithm sequences that make it past both the Viterbi and CM filters must still run in software. Although a very small percentage of the sequence passes both filters, the Inside algorithm is much slower than CYK, so a substantial amount of time is still spent running Inside. This time is less than 10% of the total prior to acceleration, but dominates afterwards. The other reason for the reduction in speedup is that these results include all software tasks performed by the system, not just Inside and the two filters. Although for large enough sequences the time spent in these tasks becomes very small, averaging less than 1% of the total runtime for a 60 million base sequence, they cannot be ignored. It is also worth noting that the ncRNA families selected for this experiment tend to be among the more frequently occurring, so speedup would likely be better for a more typical family because Inside would have fewer possible matches to score. With estimated Inside acceleration, the expected overall system speedup is much better in the worst case. For tRNA (RF00005), a particularly unusual ncRNA, speedup improves from 8x to 35x.

3.5 Conclusions and Future Work

This chapter presented an FPGA accelerator for the biologically important ncRNA homology search problem. The accelerator features FPGA implementations of the two filtering algorithms used by the Infernal 1.0 software package. The FPGA version of the first of these algorithms, Viterbi, gets a speedup of over 200x. The second, CYK, achieves a speedup of 60x over the software version. When combined into an Infernal-like system, we anticipate that these accelerators alone can run 25x faster than pure software. This speedup is limited by the fact that our current system has no accelerator for the Inside algorithm used as the final stage of Infernal's search pipeline. Given the results of our other accelerators, we anticipate that an Infernal 1.0 system featuring an FPGA implementation of the Inside algorithm could achieve a total system speedup of over 48x.

Any future iterations of this accelerator would need to implement Forward and Inside, because versions of Infernal after 1.0 rely almost exclusively on these algorithms. Some of the challenges for this approach are outlined above. Additionally, although it would result in a more complex accelerator, Eddy developed many improvements to profile HMMs and Viterbi/Forward, which could potentially be useful for accelerators such as multiple segment Viterbi and sparse rescaling [Eddy11]. Next, alternative ncRNA homology search strategies, such as those based on Hamming distance instead of covariance [Sun12], could be appropriate targets for accelerators. This area seems fertile, with almost all current ncRNA homology accelerators focusing on HMMs and CMs. Finally, there are other applications, such as stochastic context-free grammar based approach to RNA structure prediction found in PPfold [Sükösd11], that use very similar algorithms to CMs and could perhaps be targeted with a combined more broadly useful accelerator.

Chapter 4. K-MER COUNTING

To this point, we have discussed processing sequences when a prior model is available. In the case of short read alignment, the aligner compares the reads to an index based on a known reference or references such as mouse, human or E. coli. For ncRNA family identification, sequences are scored against a library of family models, such as Rfam. Although these models allow for fast and accurate search algorithms, there are also numerous situations where no model is available. For example, when the Human Genome Project built the first human reference sequence or when first constructing the Rfam database. This chapter focuses on an initial filter for one such problem. Parts of our approach to this filter was published in [McVicar17].

4.1 De Novo Assembly and K-mer Counting Background

De novo assembly is the problem of assembling an entire genome, or large contiguous fragments of a genome, from smaller fragments such as short and long sequence reads. In the case of short reads, this is much more computationally expensive than alignment [Zerbino08] because the short reads must be assembled in relation to each other which is fundamentally an all-to-all challenge. This can occur in many biologically important cases, such as a species which has not yet had its reference constructed (currently the vast majority of species on earth). It can also happen if the source of the reads is not known, for example in a metagenomic study of the life in seawater, dirt or the human gut [Peng12]. In these cases, the reads must be assembled into coherent sequences in much the same way as the first human reference during the Human Genome Project.

Although methods and computational power have improved significantly since 2003 when the HGP was completed, de novo assembly of short reads is still much more difficult and computationally expensive than aligning to a reference. One solution to this problem is assembling long reads with lengths of tens or hundreds of thousands of base pairs, from sequencing platforms such as those from Pacific Biosciences and Oxford Nanopore. Long read assembly can produce relatively accurate results quickly [Koren17], but there are still significant advantages to short reads, particularly in terms of cost and throughput [Jackman17]. Given the expense of de novo assembly of short reads, data reduction techniques are frequently used to improve performance. Of these techniques, K-mer counting is one of the most powerful and most popular.

4.2 K-mer Counting

As discussed in section 1.2.4.1, errors in short reads can be resolved through the use of many overlapping reads with higher coverage generally producing higher quality results at the cost of additional computation. For de novo assembly, average coverage of at least 10x is important for result quality, and due to biases in coverage a significantly higher average may be required to ensure that all regions have sufficient representation [Movahedi12]. This proliferation of reads greatly increases the processing time required to complete the assembly. In addition, the inclusion of more reads introduces more unique erroneous short sequences, due to read errors. If these reads are all included in the assembly without any attempt at error correction, the accuracy of the final result suffers as the assembler attempts to reconcile many slightly different versions of the same sequence.

K-mer counting is a data reduction and error removal step included in many sequencing flows to address these issues. The short reads are broken into overlapping subsequences called K-mers, typically of length 20 to 70 bases. Then, the occurrence rates of each K-mer are counted across all reads in the dataset. Since scientists try to ensure sufficient coverage, the reads will generally include 15 or 30 copies of the same data. A K-mer that only occurs once or twice in the dataset is thus very likely to include an error, and can be discarded [Chikhi13a].

In K-mer counting we break the data into K-mers, count the occurrence of each K-mer in the dataset, and eliminate all K-mers that appear fewer than some threshold count. For other K-mers, only one copy of the K-mer and an occurrence rate needs to be retained. This not only removes errors, but also reduces the size of the sequence graph that must be processed by the assembler. K-mer counting is a computationally expensive part of typical de novo assembly pipelines, sometimes consuming almost half of the run-time [Chikhi13b].

The following sections describe background necessary for understanding our K-mer counter implantation including minimizers and signatures, Bloom filters and additional details of the Hybrid Memory Cube (HMC) used in our hardware implementation.

4.3 MINIMIZERS

One obvious disadvantage to converting sequences, such as short reads, to K-mers is that (prior to counting) the amount of storage required for all K-mers of a string is much greater than the string itself. For example, a 100 bp read can be stored in 25 bytes without compression. Converting this read to 20-mers produces (length – K + 1) = 81 of them, resulting in a storage requirement of 405 bytes. Minimizers provide a solution to this problem, first proposed in [Roberts04]. For a given string, such as a short read or other DNA sequence, the minimizer is the lexicographically smallest M-mer in the string. As we will demonstrate, minimizers allow us to split a large collection of K-mers into manageable groups, as well as reduce the total storage requirement.

Consider the length 12 read CTTACGGTATTG, shown in Figure 4.1. Using the same arithmetic as before, this read requires 3 bytes of storage. Similarly, there are seven 6-mers, requiring 11 bytes. For this example and M = 3, the minimizer is the lexicographically smallest 3-mer of each K-mer. Observe that the first four K-mers CTTACG, TTACGG, TACGGT and ACGGTA all share the same minimizer (ACG), highlighted in orange in the top part Figure 4.1.

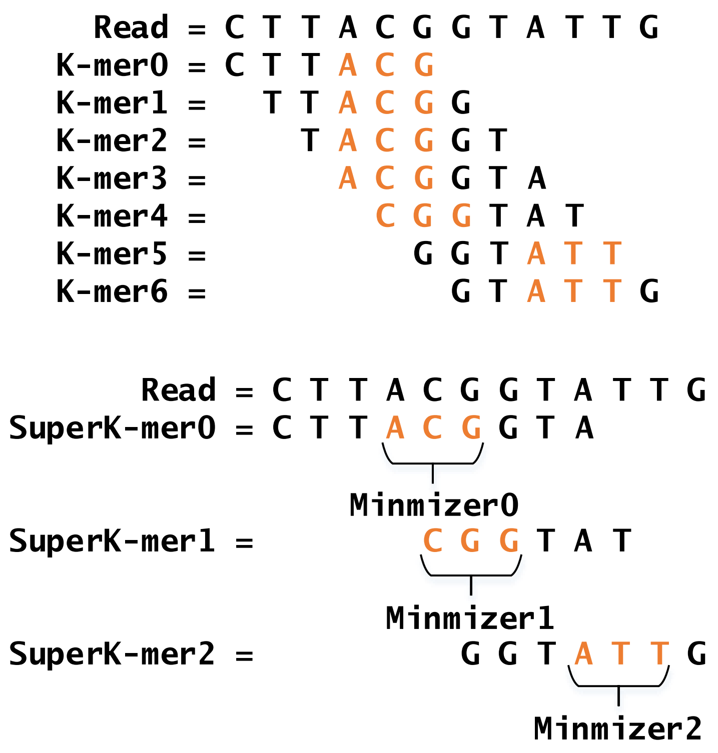


Figure 4.1 K-mers for a 12 bp read (top) and Super K-mers for the read (bottom), with minimizers shown in orange

Given that these K-mers share a minimizer, they can be grouped together into the nine nucleotide “super K-mer” CTTACGGTA. A super K-mer is a collection of neighboring K-mers that share the same minimizer. For the read in Figure 4.1, there are a total of 3 super K-mers (bottom). These have a total length of 9 + 6 + 7 = 22 bps and require 6 bytes of storage. This technique saves a great deal of space (and thus bandwidth), roughly ~2x over a naïve binary encoding of the K-mers for human short reads [Deorowicz15] and as much as 10x for other genomes [Li15]. However, the practical results are slightly worse because reverse complement minimizers must also be considered, since a K-mer is equivalent to its reverse complement and must be counted together.

Although compression is one valuable result of minimizers, it is not the most important for our purposes. Many K-mer counters partition K-mers into groups before counting, in order to reduce the total memory footprint of the counting data structure, and minimizers are perfect for assigning these partitions. For example, DSK [Rizk13] can count the 27-mers in the human genome in 4 GB, as opposed to the 56 GB required by BFCounter without partitioning [Melsted11]. Any counter that uses partitioning must guarantee that a given K-mer ends up in the same partition each time it appears in the read data. A naïve approach might be to partition based on the first few nucleotides of the K-mer, for example using an AAA partition, an AAC partition, and so on until the final TTT partition. However, this approach will result in large partitions of uneven sizes because of K-mers blowing up the size of the reads, as described above. Using minimizers, on the other hand, allows for many neighboring K-mers to be stored in the same partition.

4.3.1 *Signatures*

The KMC 2 K-mer counter, on which our work is based, modifies minimizers through a technique known as signature generation [Deorowicz15]. Signatures are based on two important observations about minimizers when used for K-mer counting. First, minimizers tend to lead to unevenly sized partition sizes, since minimizers with repeated bases, particularly As, are extremely common. Some tools attempt to deal with this problem by doing an initial survey of the minimizers present in the reads before assigning the partitions, but this introduces complexity and is not always reliable. For example consider a scenario where 512 partitions are used for 6 nucleotide minimizers, of which there are 4096 possibilities. In this case, it is entirely plausible that the minimizer AAAAAA will include more than eight times the "expected" number of K-mers, making uniform partitioning impossible. Table 6 of [Deorowicz15] demonstrates this effect, with the largest partition being roughly double the ideal size.

Strings of nucleotides, particularly As, introduce a second problem as well. Super K-mer length is often limited by this string as it reaches the end of the current K-mer. For example, for 6-base minimizers again and K = 9, consider the somewhat improbable read AAAAAATTTTTT. For the first K-mer the minimizer will be AAAAAA. For the second K-mer the minimizer will be AAAAAT. The third will have the minimizer AAAATT, and so on. This means that each K-mer will have a new minimizer and will thus become its own super K-mer, resulting in exactly the data explosion issue that minimizer were intended to avoid.

Signatures are intended to mitigate both of these issues, and KMC 2 does so using two rules. Until now, we have defined the minimizer of a string as the lexicographically smallest M-mer. However, this is not the only possible definition. In particular, given that the direction of short reads is typically unknown, applications often use the canonical minimizer, which finds the lexicographically smallest M-mer from both the string and its reverse complement. The signature for a K-mer is the canonical minimizer, except that it may not start with the sequences AAA or ACA, and may not contain the sequence AA except at the start. Although this method may appear simplistic, [Deorowicz15] compares it to other more complex techniques from the literature with favorable results. For cases where a K-mer has no legal signature, such as one made entirely of As, an illegal signature may be used, but note that the number of K-mers with this minimizer should be a much smaller percentage of the dataset.

4.4 Bloom Filters

A basic Bloom filter is a space-efficient mechanism for answering the question, “Have I seen this data item before?” [Bloom70]. A naïve method for handling this question is to have a bit in memory for each possible data item, then simply set that bit when the corresponding value is seen. However, for many questions the number of possible items is astronomical. Instead, we could hash each data item for lookup in a smaller set of bits. However, collisions in the hash function become a problem, and storing the actual data items in this hash table to resolve collisions may also require a huge memory.

Bloom filters maintain a small number of bits in a hash table, and do not keep the original items, so collision detection is impossible. Instead, the Bloom filter applies several independent hash functions to the incoming data item, and uses the result of each of those as a lookup in the bit table. Thus, if there are 3 hash functions, the item will map to 3 separate bits in the table. If any of those bits are a 0, we are guaranteed to have never seen that value before, and a result of “not seen before” is returned; those 3 bits are then set to 1. If during a lookup we find all the bits are true, we return the result of “has been seen before”. Note that this does mean there can be false positives: although a single different item is very unlikely to have set the same three bits to 1 previously since the hash functions are independent, it is possible that 3 previous items have each set one of the current value’s bits to true. Because of this, the number of hash functions and the size of the Bloom filter must be chosen to keep the false positive rate sufficiently low. The mathematical theory behind Bloom filters allows each parameter to be chosen correctly for a given application.

4.4.1 *Bloom Filter Variants*

Although the Bloom filter described above is extremely powerful, it also has many limitations. For example, items cannot be removed from the table, since there is no way to know if inserting another element also set the same bit. Additionally, Bloom filters although space efficient, are not optimally so [Pagh05]. Our work uses a variant called a counting Bloom filter.

A counting Bloom filter answers the question “How many times have I seen this data item before?” [Fan00]. Instead of a single bit per location, a counting Bloom filter stores a small counter value, for example 4 bits if we wish to handle counts of 0..15. When performing the lookup, the filter increments all counter values associated with the hashes (saturating at the maximum value), and returns the smallest value found. Note that it might be more correct to only increment the counters containing the smallest value, but this introduces a sequential dependency that can limit performance. In practice, an error due to this over-incrementing is equivalent to the false positives found in standard Bloom filters, and is generally considered insignificant. As with a standard Bloom filter, the number returned will never be smaller than the true result, though the value returned may be too high, which is again the equivalent of a false positive. Counting Bloom filters also allow elements to be deleted by decrementing the counter values, although this feature is not relevant for K-mer counting and is not used in our work

In subsequent sections we will discuss the hardware implementation of a counting Bloom filter for a K-mer counter. It is important to consider the steps required to do a lookup in the data structure: (1) Hash the incoming data with N independent hashes; (2) Read the current value at those N locations; (3) Increment the value at each location, subject to saturation. These steps are illustrated in Figure 4.2. In this example, in the initial state the K-mers CCTT and CTTC have already been inserted once each. This is a 2-hash filter and none of the hashes for these two K-mers overlap, so there are a total of four counts in the filter with a value of 1. In order to insert a single instance of the K-mer TTCC in the filter, it must be hashed twice. The first hash (the thin line) results in an unused position, but the second hash (thick line) collides with the second hash from CTTC. When this bucket is incremented the new value is now 2. Incremented values are returned from the filter, and the lowest is used as the count, correctly returning a single occurrence of TTCC from the first hash. If we were to insert TTCC in the filter again the count at the first hash would increment to 2 and the second count to 3. Upon returning these values the count would be correctly identified as the smallest value, 2.

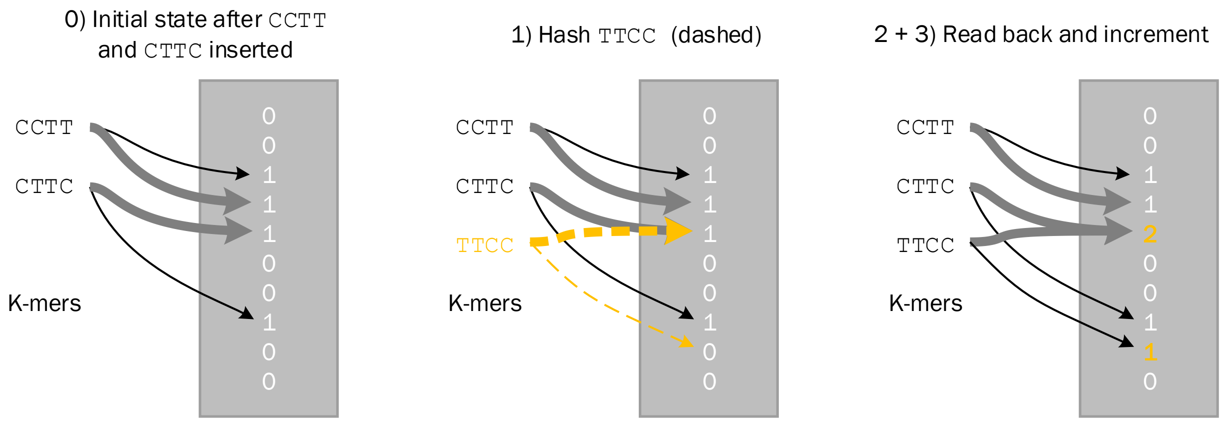


Figure 4.2 Counting Bloom filter insert operation for a 2-hash filter, where the thin line represents the first hash and the thick line the second

Note that if we allow concurrent accesses to improve throughput, this read-modify-write operation must essentially be made atomic. If two lookups to the same counter are in flight simultaneously, the resulting count may be corrupted. Correctly operating Bloom filters sometimes produce results that are too high, but cannot report a number too low (other than from saturation).

Although counting Bloom filters perform well for our purposes, they have since been superseded in many uses by more exotic data structures with similar properties. [Fan14] compares a number of Bloom filters in its Table 3. We believe that our work could be easily extended to one of these structures, resulting in noticeable performance improvement. However, this is outside the scope of what we address here.

4.4.2 *FPGA Bloom Filter Implementations*

FPGA-based Bloom filters have been in use for networking applications since at least the early 2000s [Dharmapurikar03][Lockwood03]. More recently other uses have emerged related to caching or otherwise filtering non-network traffic data [Becher15][Dobai15]. There are also more exotic use cases, such as extremely low power systems [Lyons09]. However, all of these applications have one thing in common: The Bloom filters are typically small (100s of kB to MBs), and are stored on chip. Off-chip Bloom filters in traditional DRAM are either not viable (too slow) or unnecessary (DRAMs are relatively large when considering typical non-HPC FPGA applications). However, HMCs and other stacked memories with very high random access rates may make off-chip Bloom filters viable for FPGAs.

4.4.3 *HMC Atomics and Bloom Filter Considerations*

One use the HMC makes of its logic layer is supporting a variety of atomic operations. Although the 1.1 specification [Hybrid Memory Cube Consortium14] provides the masked write required to treat each Bloom counter as independent of other counters for the read-modify-write increment operation, a single atomic operation to perform this update could reduce FPGA logic and increase performance by almost 50%. The more recent HMC 2.1 specification [Hybrid Memory Cube Consortium15] provides additional atomic operations that should be considered when designing an FPGA-based system using the HMC. The length of operations available in the 1.1 specification is not variable, since all of the operations provide a similar function of adding one or two immediate values to the data stored at a specified direct address. However, with the greater variety of operations available in the 2.1 specification, the length and other parameters can change.

The length of the message and response are measured in FLITs, which are serialized 128-bit flow control units that make up the HMC packets transmitted across the links. Posted operations are so-named because they do not require a response. For messages that do generate a response, WR_RS is a write response packet and RD_RS is a read response packet. Although posted write operations do not normally receive any write response packet, they can generate an error response packet in some cases.

Table 4.1. HMC Specification 1.1 atomic operations

Operation	Response	Operation FLITs	Response FLITs
Dual 8-byte add immediate	WR_RS	**2**	**1**
Single 16-byte add immediate	WR_RS	**2**	**1**
Posted dual 8-byte add immediate	N/A	**2**	**0**
Posted single 16-byte add immediate	N/A	**2**	**0**

This set is greatly expanded in the HMC 2.1 specification to include address request operations, additional arithmetic operations and bitwise and Boolean logical operations. Finally, it includes a number of comparison and compare and swap operations, which can be particularly useful for creating semaphores. These operations are shown in Table 4.2. Observe that due to the wide variety of operations available, different message lengths and responses are generated.

The add immediate and return operations are particularly powerful when used to perform the increment for a counting Bloom filter. Because the address buffer (see 4.6.3) prevents multiple write operations in flight to a single counter, only a single additional reserved value would have to be maintained to avoid overflow. In the case that the add immediate and return operation returns a saturated counter a posted write back to the original value (saturation – 1) would maintain correct operation with minimum bandwidth overhead. It would also allow for a smaller address buffer because slots could be freed without waiting for the HMC controller to accept the write back in cases where the counter isn’t saturated. However, this approach provides limited benefit in cases where many counters are saturated. It may be possible to achieve better performance by starting in an add immediate and return operating mode. After the percentage of saturated counters gets too high, the operating mode could switch to beginning with a regular read followed by an increment (a single FLIT operation) if required. This mode of operation would only require 1 TX FLITs in the case of saturation and 2 TX FLITs otherwise.

Table 4.2. HMC Specification 2.1 atomic operations

Operation	Response	Operation FLITs	Response FLITs
Dual 8-byte add immediate	WR_RS	**2**	**1**
Single 16-byte add immediate	WR_RS	**2**	**1**
Posted dual 8-byte add immediate	N/A	**2**	**0**
Posted single 16-byte add immediate	N/A	**2**	**0**
Dual 8-byte add immediate and return	RD_RS	**2**	**2**
Single 16-byte add immediate and return	RD_RS	**2**	**2**
8-byte increment	WR_RS	**1**	**1**
Posted 8-byte increment	N/A	**1**	**0**
16-byte XOR	RD_RS	**2**	**2**
16-byte OR	RD_RS	**2**	**2**
16-byte NOR	RD_RS	**2**	**2**
16-byte AND (also used as bitwise clear)	RD_RS	**2**	**2**
16-byte NAND	RD_RS	**2**	**2**
8-byte compare and swap if greater	RD_RS	**2**	**2**
16-byte compare and swap if greater	RD_RS	**2**	**2**
8-byte compare and swap if less	RD_RS	**2**	**2**
16-byte compare and swap if less	RD_RS	**2**	**2**
8-byte compare and swap if equal	RD_RS	**2**	**2**
16-byte compare and swap if zero	RD_RS	**2**	**2**
8-byte equal	WR_RS	**2**	**1**
16-byte equal	WR_RS	**2**	**1**
8-byte bit write	WR_RS	**2**	**1**
Posted 8-byte bit write	N/A	**2**	**0**
8-byte bit write with return	RD_RS	**2**	**2**
16-byte swap or exchange	RD_RS	**2**	**2**

4.5 K-MER COUNTERS

Now that we have considered the major background concepts (K-mer counting, Bloom filters and HMCs), a logical question is why is an FPGA/HMC based implementation of a Bloom filter appropriate for K-mer counting?

Bloom filters are a good fit for K-mer counting because of their space efficiency, independent of K. Alternatives, such as trees and hash tables can be prohibitive in size, requiring at least double the memory and growing with K. Jellyfish was one of the first fast parallel hash-based K-mer counters [Marçais11]. Shortly afterwards, BFCounter was released [Melsted11]. BFCounter was similar in design, but stored K-mers in a classical Bloom filter. This approach cut the memory requirement roughly in half when compared to Jellyfish, but also increased run time by a factor of 3x to 10x.

DSK introduced the concept of partitioning, discussed in section 4.3, where K-mers are divided into smaller partitions stored on disk, and counted separately using Bloom filters [Rizk13]. This approach reduces the memory required by a factor of 10 over BFCounter, as well as producing a substantial speedup. When paired with fast SSDs DSK is roughly the same speed as Jellyfish, while using 6% of the system memory and a similar amount of disk space. MSPKmerCounter further improved the partitioning scheme and moved away from Bloom filters, thereby sacrificing memory footprint for performance, resulting in a roughly 2x improvement over Jellyfish in some circumstances [Li15].

KMC 2, which our work is based on, improves on MSPKmerCounter through the use of advances compression techniques including signatures, discussed in 4.3.1 [Deorowicz15]. This results in performance better than JellyFish or MSPKmerCounter with a memory footprint similar to DSK. KMC 2 has a reduced disk bandwidth requirement, due to the larger super K-mers, and reduced CPU load and memory bandwidth due to the absence of a Bloom filter. Because partitioning K-mers to disk incurs a performance penalty, some solutions retain the in-memory approach of Jellyfish. For example, KMCBT adapts the compression of KMC 2, along with a fully in-memory tree-based architecture to give improved performance compared to KMC 2 at the cost of a large memory footprint that is not as amenable to FPGA work [Mamun16].

The reader might have observed that many more advanced K-mer counters, such as KMC 2, abandoned Bloom filters due to computational cost and memory bandwidth issues as well as the fact that a 4 GB footprint may be unnecessarily small in an era where inexpensive desktop machines have 16 GB of memory or more. However, FPGA-attached memories are traditionally smaller and our platform reintroduces mitigates the aforementioned concerns. The HMC provides significant memory bandwidth with a relatively small memory size, suggesting a good fit for a Bloom filter. Similarly FPGAs are well suited for the parallel computation introduced by resolving multiple hashes and count updates simultaneously, while the compression techniques used by KMC 2 and KMCBT require significant serial computation in their current form.

Another potential issues with Bloom filters is false-positives, resulting in an over-count of a K-mer for this application. These are not overly problematic because assemblers typically discard low occurrence K-mers as potential errors. If such a K-mer is retained, it merely represents a missed speedup where the data processed by the assembler is larger than otherwise, not a significant degradation of the final results (see [Melsted11] Figure 4). However, if we throw away good data (false negatives) we are potentially reducing the quality of the final assembly.

Given all of the above, a counter with a small memory footprint like DSK is ideal. Bloom filters are well suited to FPGA-based computation and for implementation using the HMC. Large volumes of small random accesses fit the HMC model well. Finally, this system generates many memory operations that are almost always independent, allowing for a highly parallel implementation in addition to efficient use of HMC bandwidth. Including signatures and other improvements from KMC 2, while retaining Bloom filters, results in a high performance low memory model for an FPGA-based K-mer counter.

4.6 FPGA COUNTER DESIGN

An overview of our K-mer counter can be seen in Figure 4.3. First, the reads are partitioned into small blocks of K-mers that share a signature. This is a technique used when the available working memory for counting is small, such as on the FPGA board. Next, the partitions of K-mers are transferred to the FPGA, one at a time or in blocks, where they are counted. The incoming K-mers are processed and counts are written to FPGA attached memory. Here, an associative structure is required where the key is the K-mer and the value is a count which is incremented every time that K-mer is seen on the input stream. Bloom filters are well suited to this purpose for the reasons discussed above.

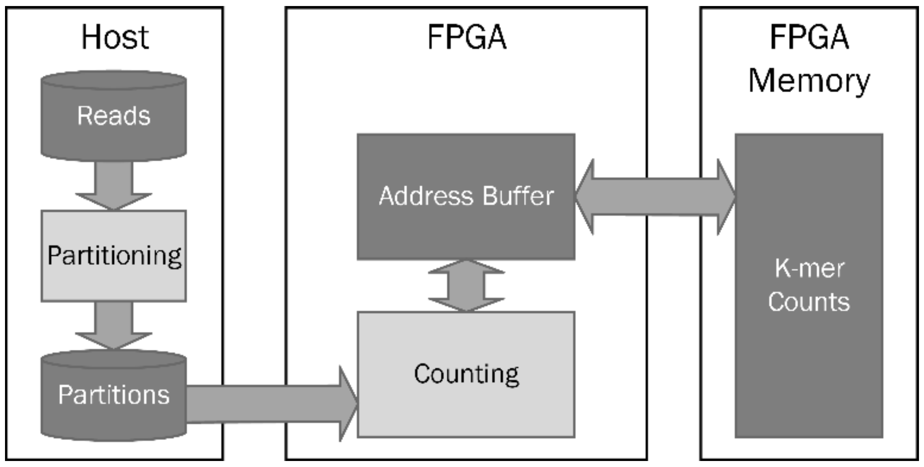


Figure 4.3 Generic FPGA-accelerated K-mer counter block diagram

The above design does not necessarily outperform software. Unpacking the partitioned K-mers requires hashing them to identify the correct memory address for each count, which is not a trivial amount of computation. Additionally, our host system can saturate its DRAM bandwidth when using all CPUs for these tasks, so the bottleneck for counting partitioned K-mers may be memory bandwidth and not computation. In order for our accelerator to achieve any significant speedup, its effective random access bandwidth must be greater than the host's. The HMC provides exactly this feature. Additionally, the representation must be compact since FPGA-attached memories are typically smaller than what is available to the host, and this compactness is provided by partitioning, minimizers and the more advanced techniques used in KMC 2.

4.6.1 *System Architecture*

Our implementation of the generic system from Figure 4.3 is built on a Micron SB-800 board featuring 4 Altera Stratix V FPGAs, and a 4GB HMC with a single high-speed serial link to each FPGA [Micron Technologies17]. In order to achieve the best performance, no link can accesses memory outside of its own four HMC vaults, so we decided to have each of the four FPGAs work independently on their own small Bloom filter. This approach fits well with K-mer counting where the large set of K-mers is already partitioned on disk so that the counting structure can fit into system memory. Because of this approach, there is very little downside to using multiple smaller Bloom filters in parallel instead of one larger one across all of the FPGAs. The overall system architecture is shown in Figure 4.4.

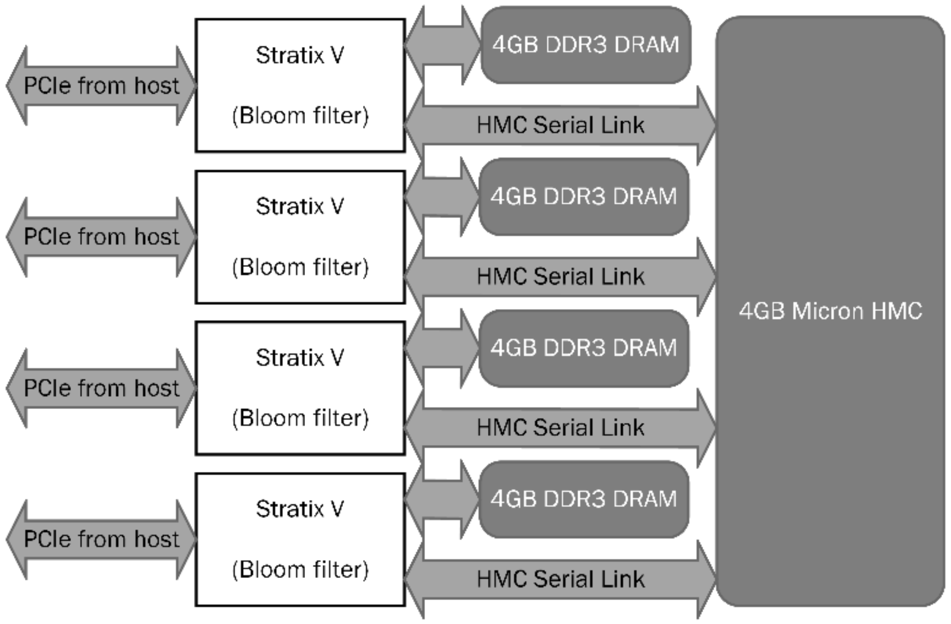


Figure 4.4 FPGA Bloom filter system architecture

The Bloom filter on each FPGA is designed to maximize memory utilization, since the entire system should ultimately be memory performance limited. K-mers arrive at the Bloom filter uncompressed, potentially having first traveled through a decompression unit, as seen at the top of Figure 4.5. This unit is not present in our evaluation. Next the K-mers are hashed, with the K-mer retained in a bypass FIFO for output.

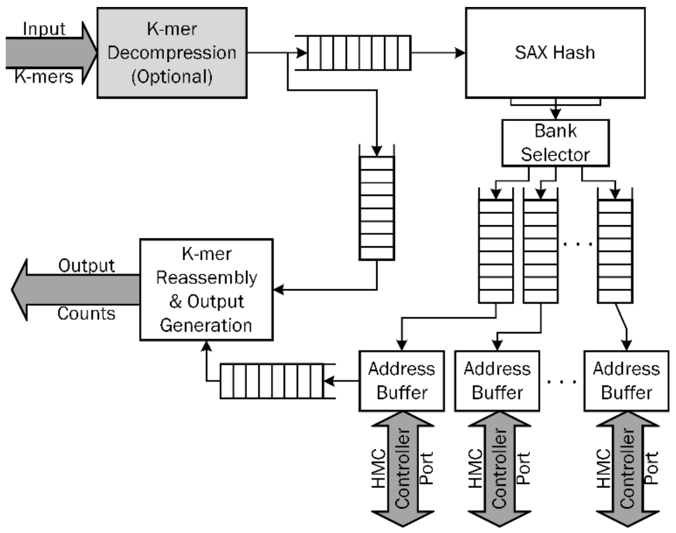


Figure 4.5 FPGA Bloom filter block diagram

After the K-mer has gone to two or more independent hash functions, the resulting hashed addresses are tagged to identify their origin K-mer. Tags are assigned sequentially, and are used to reintegrate data that flows down the independent datapaths in this design.

The Bloom filter then looks up each counter, increments it and finds the minimum value. Doing a single lookup at a time would be simple, but serializing HMC access is too slow. To saturate the HMC multiple operations must be in flight simultaneously, while maintaining atomic read-modify-write operation. This is particularly difficult because although the HMC will never reorder operations that access the same memory bank, the HMC memory controller located on the FPGA can perform just such a reordering. This controller, through which user logic communicates with the HMC, has 4 memory ports and may reorder operations arriving on different ports, even if they address the same bank.

To support efficient pipelining in the face of reordering, we assign banks to specific HMC controller ports, so that all accesses to a given bank use the same port, preventing accesses from being reordered. We also maintain a buffer of addresses in-flight, and will stall a subsequent access that targets the same Bloom filter counter to maintain atomicity. This is necessary because the read-modify-write introduced by the increment operation requires 2 memory accesses on this HMC revision. Because only data from the given counter is used, and writes are masked, reordering of operations that access different segments of the same memory word are not a problem.

When all reads for a K-mer complete, the filter finds the minimum count, and issues memory writes to update the counters that have not already saturated. These addresses are also removed from the buffer, so that any stalled accesses can proceed. The filter outputs the count along with the actual K-mer stored in the bypass FIFO for all K-mers or only new K-mers, depending on the configured output mode.

4.6.2 *Hash Functions*

Choosing a good hash function is very important for Bloom filter performance, and a great deal of work has been done on hash functions appropriate for ASIC and FPGA use [Yamaguchi13][Gosselin-Lavigne15]. We selected a two round shift-add-xor (SAX) hash function [Ramakrishna97]. This class of hash functions behaves very well for repetitive strings (like genomic sequences), avoids the use of multiplication, and easily allows for very deep pipelining, making it ideal for our purposes. The SAX function accepts 8 bits at a time. For length 63 K-mers, and 2-bit bases, we apply a 16 stage pipeline to compute the hash. Note that since our application is not latency sensitive, a deep hashing pipeline is both acceptable and necessary. A new K-mer can arrive each clock and requires multiple hashes. The hash pipeline is shown in Figure 4.6. Note that the pipeline actually employs twice as many stages as discussed above, where the second hashing round improves result quality.

The SAX hash produces a single 64-bit hash value, which could index 2^{64} different counters. However, since we only require 2^{30} different counters in our Bloom filter, this single 64-bit hash is broken into 2 32-bit hashes. Note that for designs that require more hash bits we replicate the SAX pipeline with a different seed. For example, using 4 hash functions for our Bloom filter, as in the evaluation below, required 2 copies of the entire SAX pipeline.

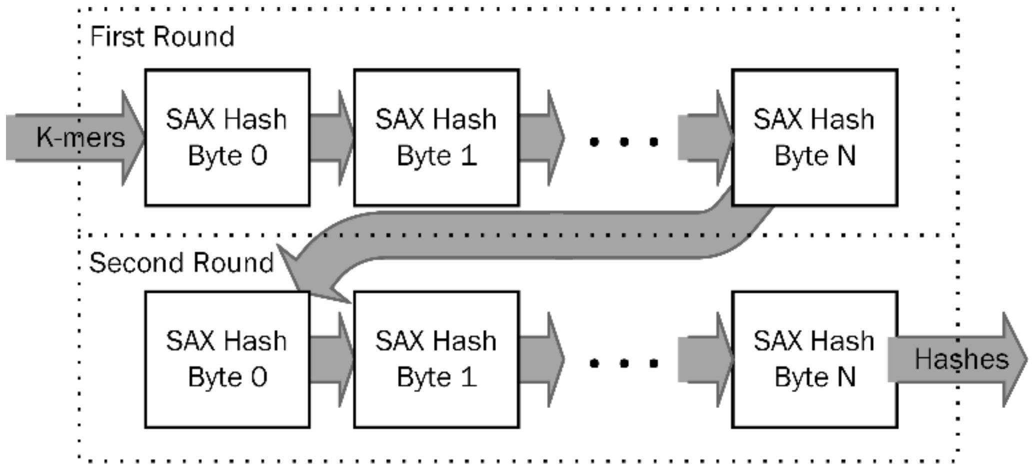


Figure 4.6 SAX hash pipeline

To demonstrate that SAX is an effective hash function for Bloom filters we compared the False Positive Probability (FPP) of a Bloom filter using SAX and SHA-256 to the expected FPP calculated using the approximation given in [Mitzenmacher05]. The results of this comparison can be seen in Figure 4.7.

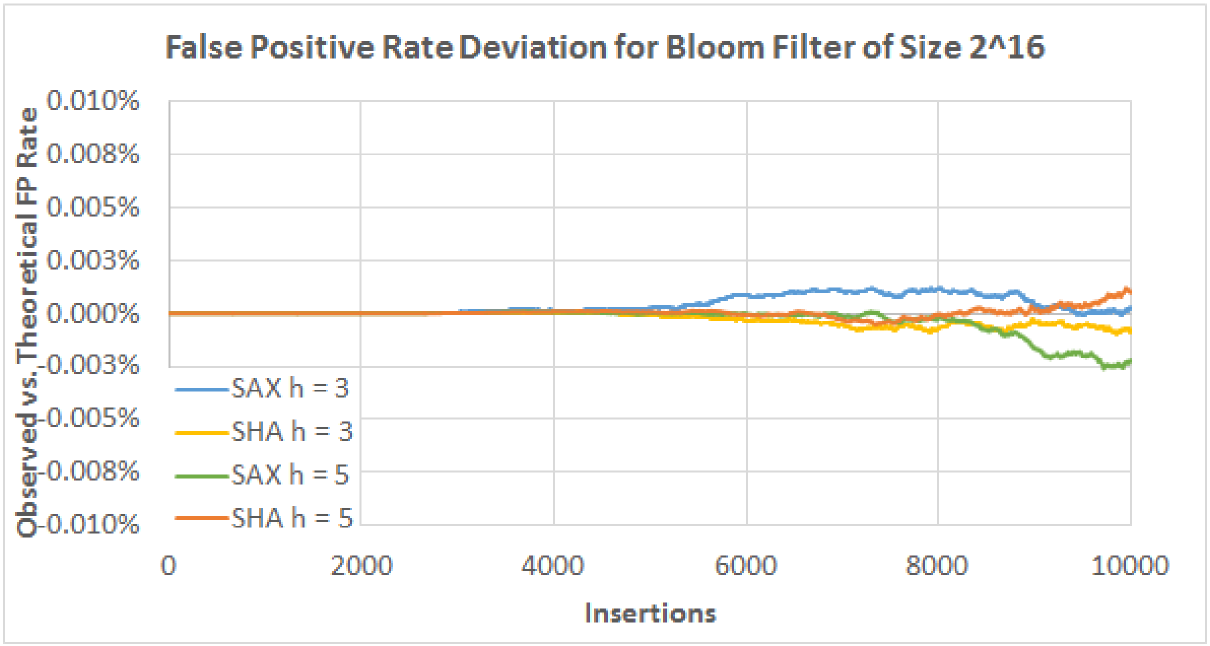


Figure 4.7 Bloom filter hash function FPP, where higher than theoretical FPP is positive

On the horizontal axis is the input size. This data was generated by averaging the FPP from many runs with a relatively small Bloom filter, so as the input size approaches 8,000 the filter becomes very full and the FPP rises significantly. SAX hash performs almost as well as the much more expensive SHA-256 in all situations, and both track the expected FPP very closely.

The difference between SAX and SHA-256 is at most 1/100th of 1%. However, note that this is for a Bloom filter with a large load factor (the fullness of the filter), such that the FPP is much too high (around 2% for N=5 and 4% for N=3, where N is the number of hash functions). In the normal operating range where FPP is safely well under 1%, SHA-256 and SAX perform extremely similarly. Given that SHA-256 is a very well behaved cryptographic hash function, this suggests that SAX is entirely sufficient for our Bloom filter.

4.6.3 *Address Buffer*

After being hashed, counter addresses from the K-mer enter a buffer (Figure 4.8). This module serves two purposes. First, it generates and remembers both the HMC tag (used to identify returning data from the HMC) and a K-mer identifying tag for each address for output generation. Second, the buffer provides collision detection. In particular, any K-mers that hash to one or more of the same counters will always be sent to the same address buffer because the bank selector distributes the hashes based on their hashed address. Therefore, collision detection at the address buffer is sufficient to avoid any races. In the case of a collision all addresses coming to that buffer are stalled until the aliasing K-mer's update is completed, at which point the entry is cleared and the new address can proceed. Stalling an entire set of banks, which can eventually put backpressure on incoming K-mers, is not the most efficient solution; however it allows for much simpler hardware, and collisions are extremely rare. The total impact on performance is much less than 0.1% in our tests.

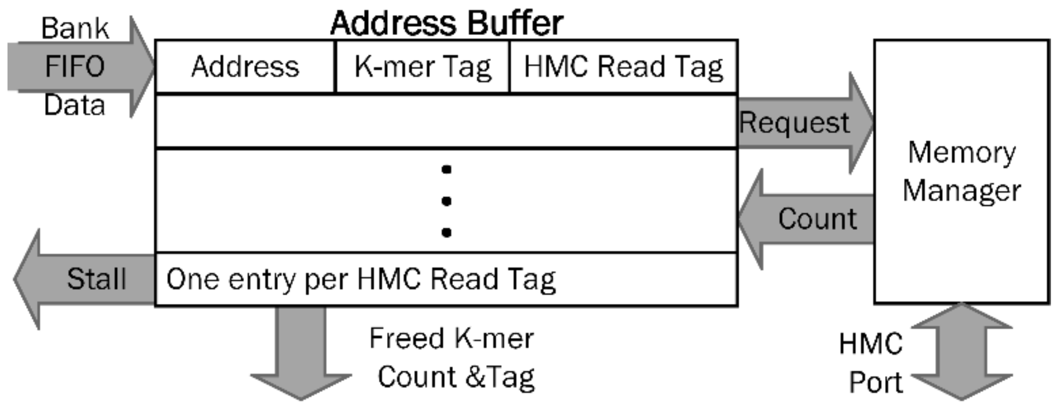


Figure 4.8 Address buffer and memory manager

The buffer also handles the reordering of read results returning from the HMC. This would not be necessary if there were a single buffer per bank, but there are many banks (for the HMC used in our work there are 4 vaults per link and 8 banks per vault for a total of 32 banks per link, see Figure 1.4) and each buffer would be inefficiently small if there were truly one per bank. We instead have per-HMC controller port buffers, with reordering resolution logic.

When read data returns, the counter must be incremented, and that value must be written back using a bit-mask to avoid impacting any other counters that may have been updated after the read. This allows the buffer to stall only when there is an exact counter address collision, not every time two addresses share the same word in memory. Note that counts saturate at the full counter value. For example, once a count reaches 15 when using 4-bit counters that counter will no longer be written back. This saves a minor amount of memory bandwidth. Unsaturated counters are always incremented and written back as described.

After any required write-backs are issued by the memory manager, the buffer entry is invalidated and dependent stalls are released. The count returned by the memory is sent to the output module along with the K-mer tag. The output module must collect the value of each counter that a K-mer hashed to and return the minimum of those values as the count for that K-mer.

4.6.4 *Signature Generator and Partitioning*

While our hardware implementation does not perform signature generation or partitioning, we believe that doing so on the FPGA could provide a performance advantage, so our proposed design is worth including here. It is important to understand that the work presented in this chapter focuses on an FPGA accelerated Bloom filter using the HMC and the application of that filter for K-mer counting. Partitioning on the FPGA could increase total speedup for K-mer counting, but it is not an essential part of the issues explored in this chapter.

Our FPGA signature generator design identifies the signatures for each K-mer in the read and combines them into the corresponding super K-mers (Figure 4.9). These could be transferred back to the host and written to the appropriate disk partitions as those partitions' write buffers fill up. For each input read, the minimizer tree identifies the signature for every K-mer. In this case, the signature is the minimizer, excluding minimizers that would violate the rules described in section 4.3.1. The output of the minimizer tree goes to the super K-mer construction module, which compacts K-mers with the same signature into super K-mers. These are buffered and written back to the host. Note that read information is no longer relevant at this point because super K-mers are typically stored in partitions without any pointers back to the read they originated in.

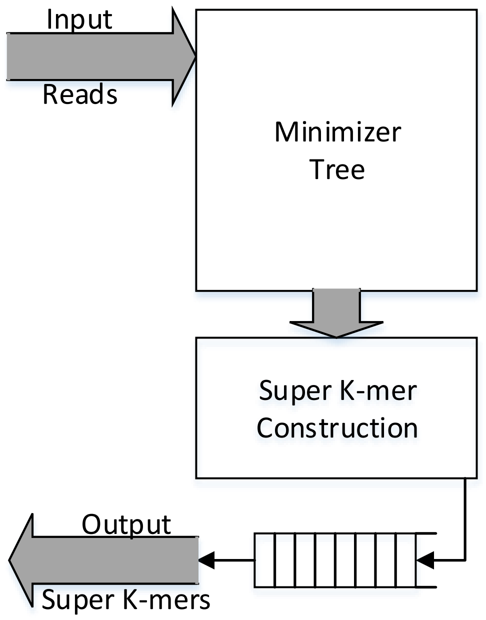


Figure 4.9 FPGA practitioner block diagram

The minimizer tree uses a series of comparisons to find the correct signature for every K-mer. The minimizer beginning at the nucleotide at each position in the read (except for those fewer than M bases from the end) starts owning its own position in the read at the top of the structure and then a series of comparisons with an exponentially increasing range replace the signature owning each K-mer with the correct one (Figure 4.10). In particular lexicographically smaller minimizers always replace larger ones, unless they are prohibited by rule. Because the structure operates in the same ordered manner across all comparisons ties are resolved consistently regardless of where they occur. At the end, the results of the final comparison include the correct signature for each K-mer, and the decreasing width of the tree has appropriately accounted for the number of K-mers in the read. K will always be significantly larger than M, and the first row of comparators is the number of minimizers in the read, which is fewer than the number of bases.

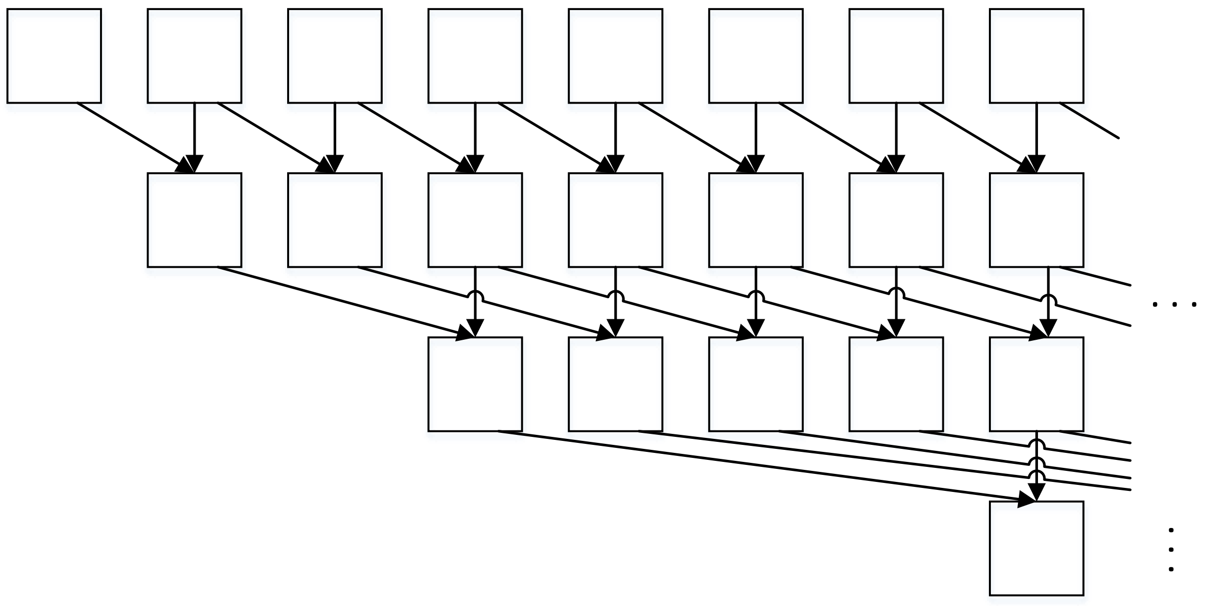


Figure 4.10 Minimizer tree architecture block diagram for the first three stages of the earliest K-mers, with 2-input comparators

Although it is possible for this tree to function with any minimizer and K-mer size, it can only support specific sizes for M and K without significant additional tagging information (such as the initial position of the minimizer in the read) at each position. For example, 2-input comparators require $\log_2(K – M + 1)$ stages and can only correctly identify the signature for every K-mer without additional information if K – M + 1 is a power of 2. With 3-input comparators at each node $\log_2((K – M) / 2 + 1)$ stages are required. The most efficient combination of input widths and K-mer lengths is an open question, as are the potential advantages of more complex tree structures.

After the minimizer tree identifies the signature of each K-mer, the super K-mer construction module combines neighboring K-mers with matching signatures into super K-mers for output. Once the boundaries between the signatures have been identified (trivially with a series of comparators), priority encoders can identify the super K-mers for the read. At this point different encodes are possible, including storing the super K-mers themselves for output or maintaining their positions and the input read.

Finally, the output buffer packs the completed super K-mers. This is not unimportant because output bandwidth is the most likely bottleneck for partitioning (aside from disks on the host). For 100 bp reads, the average number of super K-mers ranges from 2 to 6 depending on K-mer length [Deorowicz15]. If we assume a worst case scenario of a relatively short K ≈ 31 and 5 super K-mers per read, there are roughly 45 bps per super K-mer. At this length each super K-mer requires 12 bytes of storage and the total for the read would be 60 bytes. If we assume 2 GB / s of FPGA-to-host bandwidth, this results in a total of 36 M super K-mers / sec per FPGA or over 7 M reads / sec per FPGA before bandwidth becomes the bottleneck. For comparison, KMC 2 can partition roughly 1 to 2 million reads per second under the same conditions, using all CPUs, depending on the host system. If these assumptions hold, our FPGA-based partitioner could achieve a 4x speedup over KMC 2 in the worst case and significantly more with longer (more realistic) K-mers.

4.7 FPGA Results

This section presents the evaluation of our FPGA-based K-mer counter, and particularly the counting component of the problem. This was done with care to avoid the disk bottleneck and other limitations of our evaluation platform, which we do not feel has any bearing on the fundamentals of our design, aside from the fact that any disk partition based K-mer counter will require fast disks to avoid the bottleneck at this point. Less constrained versions of the system are discussed in section 4.8.

As discussed above, our evaluation was performed on a Micron SB-800 board. This HMC board plugs into a PCIe slot and uses 4 Altera Stratix V FPGAs to support all of the external links of a single HMC device (smaller more recent boards, such as the AC-510 and AC-520, couple an

HMC with a single Xilinx or Intel FPGA across multiple links). Thus, our system is composed of 1 HMC, 4 Stratix Vs with 4 DDR3 DRAM memories, one per FPGA, and 1 host computer with 6 cores. In the results that follow, we will compare HMC, DRAM, and CPU implementations in various configurations.

The obvious question that arises is, what resources form a fair comparison? If we view this work as an evaluation of computation resources, then perhaps 1 FPGA vs. 1 CPU is the right comparison. Alternatively, if this is a comparison of memory systems, then perhaps 1 HMC (requiring 4 FPGAs), 1 FPGA-attached DRAM, and 1 host memory subsystem (which means all 6 cores) is the better target. Given that our Bloom filter logic consumes only roughly 15% of the logic resources on each FPGA, our entire design could fit on a single FPGA connected to all HMC ports on a board better suited to it. Our conclusion is that no one comparison is the right one. Thus, we will present the results of each of the different interesting configurations.

First we will compare the HMC to DRAM, followed by our reference implementation of the same algorithm running on the host, and finally a fully optimized software K-mer counter.

4.7.1 *HMC vs. DRAM Performance*

One feature of the SB-800 board is that each FPGA is connected to the HMC and DRAM through a similar interface. This provides an opportunity to compare the performance of the HMC and the DRAM at the same task. The HMC's design gives it a significant advantage over traditional FPGA-attached DRAM for random access heavy workloads, such as a Bloom filter. This results in the same 4 hash function design achieving a significant speedup when using the HMC instead of traditional DRAM on the same FPGA, as seen in Table 4.3. All tests were run on Illumina short read data from a Yoruba man and available as experiment SRX016231 [Bentley08]. This is a standard dataset, containing 93 Gbases of read data collected across 37 sequencer runs; it is frequently used for genomics performance evaluations.

Table 4.3 HMC and DRAM Bloom filter system performance for various K-mer lengths with 4 hashes (M K-mers / sec)

	K = 31	**K = 47**	**K = 63**
HMC 1 FPGA	55.8	57.4	57.5
HMC 4 FPGAs	221	226	227
DRAM 1 FPGA	4.44	4.49	4.45
HMC vs. DRAM 1 FPGA Speedup	12.6x	12.8x	12.9x

The HMC-based Bloom filter achieves approximately a 13x speedup as compared to the DRAM based system. This might be surprising because, on our board, the aggregate HMC bandwidth is only about 4x greater than the DRAM bandwidth. However, DRAM is optimized for large block transfers, and the small Bloom filter accesses required fit traditional DRAM poorly. In comparison, the HMC is able to support small accesses much more efficiently, outperforming its relative bandwidth advantage by 3.3x.

The value of K is irrelevant in these tests because the FPGA-based system simply utilizes more resources to calculate hashes in a longer pipeline. Note that this does increase latency slightly, but the impact is minor and is not important for this application. However, if host to FPGA bandwidth was a bottleneck then the K-mer length could become important.

Although the 4 FPGAs share the HMC, each only uses the memory in the 4 vaults local to its link. This means that there is almost no performance overhead to adding more FPGAs (for K = 47 four FPGAs are 3.94x faster than a single FPGA).

4.7.2 *HMC vs Host Memory Performance*

Our system actually has three types of memory that could be used for K-mer counting: HMC, FPGA-attached DRAM, and the host's memory system itself. Given that CPU DDR3 memory controllers and architecture (wider busses, faster DRAM timings and dual-channel support) typically provide significantly more random access bandwidth than FPGA-attached DRAM one might expect the host system to outperform the FPGA DRAM solution. In Table 4.4 we compare the FPGA implementations (DRAM and HMC) to a software based implementation, including a single-threaded implementation (1 CPU), and a fully parallel version using multiple threads on all the CPUs in the system (6 CPU). The CPU K-mer counters are running on an Intel Core i7-5930K. The number of hashes is not particularly relevant for this comparison because all three solutions are impacted identically, although 4 hashes are still used.

Table 4.4 HMC and DRAM Bloom filter system performance for various K-mer lengths with 4 hashes (M K-mers / sec)

	K = 31	**K = 47**	**K = 63**
HMC 1 FPGA	55.8	57.4	57.5
HMC 4 FPGAs	221	226	227
DRAM 1 FPGA	4.44	4.49	4.45
1 CPU 1 Thread	6.76	4.27	3.28
6 CPUs Multiple Threads	41.2	28.8	22.8
1 FPGA HMC vs. 1 CPU Speedup	8.25x	13.4x	17.5x
1 FPGA HMC vs. 6 CPUs Speedup	1.35x	1.99x	2.52x
4 FPGA HMC vs. 6 CPUs Speedup	5.36x	7.85x	9.96x

The first thing to notice is that, unlike the FPGA-based versions, the CPU implementations are sensitive to the K-mer length. This makes sense since, while the FPGA can handle hashing and memory operations in parallel, the CPU must spend extra cycles on these tasks. Our CPU implementation is optimized, but doesn’t make explicit use of SSE instructions or other potentially beneficial techniques that a fully optimized software package like KMC 2 would.

The single-threaded software performance is roughly comparable to the FPGA DRAM performance. Although the host DRAM has better support for random access operations, the limited parallelism of the single threaded implementation dominates. However, the multi-threaded software version is significantly faster than the DRAM FPGA version.

Finally, the 1 FPGA HMC implementation is approximately twice as fast as the host CPUs running a fully parallelized version of the software. While both the FPGA and the multi-threaded software implementations can overlap computation with memory accesses (the FPGA through spatial parallelism, the CPU via multi-threading), the random access support of the HMC results in a significant speedup. In fact, if we use the entire HMC capacity, requiring us to use all 4 of the FPGAs on the board, we achieve a 5.36x to 9.96x speedup compared to the fully parallelized host version, with speedup depending on K-mer length.

4.7.3 *HMC vs. Software K-mer Counter Performance*

In the previous section we compared Bloom filter based K-mer counters with both FPGA and CPU implementations. However, there are alternative strategies for performing K-mer counting. To give an idea of the current state of the art, we’ve compared our FPGA/HMC K-mer counter to KMC 2 [Deorowicz15] running on the same Intel Core i7-5930K with 6 physical CPU cores and 32 GB of system memory.

Note that this comparison is only K-mer counting, so time spent on partitioning and other processing is not included. PCIe transfer is also ignored for the FPGA version because this is not a bottleneck for the reasonable Ks chosen and any lead in time to achieve the steady state is roughly comparable to software startup time for human genome size runs. A more complete end to end comparison would require partitioning which is not fully implemented on our current FPGA system. Finally, in a Bloom filter where the number of elements inserted and the size of the filter are fixed the false positive rate can be determined using the number of hashes [Starobinski03]. Through tuning, we found that we could achieve the same quality as KMC 2 using a 2 hash Bloom filter and smaller partitions. In order to understand why 2 hashes is sufficient, consider SRX016231 again. For K = 47, this data set contains 6.4 billion unique K-mers or roughly 12.5 million K-mers per partition with 512 partitions. It is possible to calculate the false positive rate using the equation mentioned above with values of 2 hashes (h), 12.5 million K-mers (n) and a filter size of 512 MB * 8 bits / byte / 4 bits / count = 1024 Mcounts (s).

$$fpp \approx \left(1 - e^{-\frac{hn}{s}}\right)^{h} = \left(1 - e^{-2\times\frac{12.5}{1024}}\right)^{2} = 0.000582$$

This false positive probability is only 0.0582% once the filter is entirely full, so for most of the insertions it will be smaller. Given that this value is small and that false positives are not extremely significant in our application, the following comparison uses 2 hashes.

Table 4.5 HMC (with 2 hashes) and KMC 2 comparison (M K-mers / sec)

	K = 31	**K = 47**	**K = 63**
HMC 1 FPGA	124	124	123
HMC 4 FPGAs	484	485	484
KMC 2 1 CPU (1 thread)	19.7	8.53	6.93
KMC 2 6 CPUs (12 threads)	52.0	41.1	27.5
1 FPGA HMC vs. 1 CPU KMC 2 Speedup	6.29x	14.5x	17.7x
1 FPGA HMC vs. 6 CPU KMC 2 Speedup	2.38x	3.02x	4.47x
4 FPGA HMC vs 6 CPUs KMC 2 Speedup	9.31x	11.8x	17.6x

As seen in Table 4.5 the FPGA-based counter compares well to a highly optimized software based solution. Like the software version of our Bloom filter, KMC 2 also slows down as the K-mer length increases. Overall, our 1 FPGA system provides a 2.4x – 4.4x speedup when compared to a fully parallelized state-of-the-art K-mer counter, and a 9.31x – 17.6x speedup when using the entire HMC with all 4 FPGAs. It's also worth noting that the FPGA performance roughly doubles going from 4 hashes to 2, suggesting the FPGA counter is almost entirely memory bandwidth limited in these circumstances. This is as designed, since sufficient FPGA resources were provided for hashing and other parallelizable tasks and the Bloom filter based counter was designed to utilize parallel FPGA resources efficiently.

4.8 CONCLUSIONS AND FUTURE WORK

In this chapter we developed a Hybrid Memory Cube based K-mer counter, which demonstrates very high performance. In many ways HMCs can be considered as a DRAM replacement, offering higher bandwidth and nearly comparable capacity. HMCs also offer better random access performance when compared to DRAM, especially for smaller transfer sizes. When we exploit the full bandwidth of an HMC, we can handle 484 M K-mers/second. In comparison a single DRAM based system can achieve only 9.3 M K-mers/second, and even using all four DRAMs the system would still be 13x slower. This 13x speedup represents an approximately 2x advantage in bandwidth, and a 3.3x advantage in efficiency for small random accesses, for the HMC compared to DRAM.

We also compared our FPGA & HMC implementation to a fully parallelized software version, which exploits multiple threads on each of 6 CPU cores in the host. The HMC based system was able to achieve a 5.36x – 9.96x speedup compared to the software version with 4 hash functions. When we compare our HMC implementation to a state-of-the-art, non-Bloom filter K-mer counter, our system is 9.31x – 17.6x faster when using 2 hash functions.

Looking at these speedup numbers and the DRAM performance, we see that an FPGA K-mer counter using DRAM would perform roughly the same or slightly worse than software. This suggests that memory style is very important and we believe the same result would appear for many other genomics problems, including the remainder of the de novo assembly pipeline.

Although this evaluation is promising for the HMC, it includes the K-mer counter only. A full system evaluation would require an FPGA implementation of the partitioning algorithm, as well as performance evaluations including disk bottlenecks on the same platform as software. Storing a list of new K-mers in the unused FPGA DRAM would allow for all counts to be transferred back to software at the end of the counting process, which is currently not possible without an additional software request because Bloom filters are not traversable. However, given the advantages of KMC 2 compression and access to extremely high bandwidth “gumstick” SSDs, it is likely that an all FPGA K-mer counting solution could easily outperform multithreaded software solutions that are not currently close to saturating that disk bandwidth.

Further improvements could come from using more recent specification HMCs with additional atomics, as discussed in 4.4.3. Our current system requires a read of each counter position and a subsequent write back of all non-saturated counters, for a total of two HMC memory operations and one return value. With atomics for saturating counters, or logic to clean up overflow if only a masked increment is available, only a single write and a single return value would be required in most cases, increasing performance by 50 to 100% depending on the details of the memory bottleneck.

Finally, the promise shown by HMC Bloom filters could be leveraged for a fully FPGA based de novo assembler, based on the memory efficient techniques used in Minia [Chikhi13b]. Minia stores the de Bruijn graph in a Bloom filter and removes false positives using a small hash table that could also be stored on the HMC or in DRAM. Subsequent version of Minia that use cascading Bloom filters could perform an entire human assembly in a 4GB HMC [Salikhov13].

Chapter 5. CONCLUSIONS

This dissertation presents three FPGA-based accelerators for bioinformatics applications. Additionally, it discusses the advantages of non-traditional DRAM for one of these applications, in the form of the HMC. The first application is short read alignment, where a hybrid software/FPGA system produces an extremely flexible and fast short read aligner. The power and near software configurability of this aligner enables a short read classification scheme that would be extremely difficult to match using a fully FPGA-based aligner. The second application also uses dynamic programming, this time to find the likelihood that a sequence belongs to a specific ncRNA family. The final application performs K-mer counting, an important filter for de novo sequence assembly, quickly using FPGA parallel compute resources coupled with an HMC for greater random access memory bandwidth.

5.1 Summary of Results

In Chapter 2 we presented a combined FPGA and software implementation of a short read aligner. This aligner is 5x faster than similar software aligners with better quality of results, and a single FPGA is over 400x faster than a single CPU when producing results of a similar quality. It is also significantly more configurable, both in terms of available options and ease of achieving these configurations, than more fully FPGA-based designs. This result is achieved primarily through the use of the combined host and FPGA DRAM bandwidth.

The flexibility of our alignment platform allows for a classification scheme for RNA-Seq reads that provides high classification quality when compared to software aligner based classification solutions. Being able to perform Smith-Waterman scoring on many more candidate locations (vs. BWA-SW) due to the high parallelism of FPGAs, is a significant factor in the improved result quality. Specifically, these results show that the Smith-Waterman alignment score itself is a very good classifier for mouse and human retinal RNA reads. Furthermore, we believe S-W scoring of paired-end reads could allow for more advanced classifiers and other down-pipeline processing in the future.

Chapter 3 presented an FPGA accelerator for ncRNA homology search. In this application a sequence of nucleotides is scored against one or more ncRNA models, known as families, in order to determine the likelihood that each sequence is a member of each family. Our accelerator includes FPGA implementations of two of the key filtering algorithms used by the Infernal 1.0 software package. These filters greatly reduce the space of candidate families before a final, and much more expensive, algorithm ultimately scores the remaining candidates. The Viterbi algorithm is the first of these filters, and our implementation achieves a speedup of over 200x versus a single CPU. CYK is the second algorithm, and the speedup is 60x. Our estimate for a system similar to Infernal 1.0, using these FPGA accelerators on a single FPGA, suggest a 25x speedup for the entire pipeline on a single core or a 3x speedup for our entire system. The greatest limitation of such a system is our lack of an accelerator for the final scoring stage, which uses the Inside algorithm. This algorithm is similar to a floating-point version of CYK, much like Forward is similar to a floating-point Viterbi. We estimate that our system, with an Inside accelerator, could achieve a speedup of 48x compared to Infernal 1.0.

Chapter 4 describes the design of a Bloom filter based K-mer counter implemented on an FPGA with an attached Hybrid Memory Cube. This counter achieves very good performance and shows that for some applications the HMC can be treated as a DRAM replacement with significantly higher bandwidth at the expense of reduced capacity and slightly increased latency. In the case of K-mer counting latency is not relevant since the application is entirely streaming, and dividing the K-mers into more partitions can reduce the memory requirements to fit in an HMC without significant performance costs. For our particular accelerator the huge increase in random access rate for small operations was most impactful, allowing for counting of 484 M K-mers / second on a single HMC. A single FPGA equipped with comparable DRAM can only achieve 9.3M K-mers / second (a 13x slowdown), and 4 FPGAs with attached DRAM only achieve 37.2 M K-mers / second. Further evaluations demonstrate that this speedup is due to an advantage of over 2x in total memory bandwidth going from DRAM to the HMC and a 3.3x advantage in efficiency for small random accesses.

In addition to comparing FPGA implementations using traditional DRAM and the HMC we evaluated a parallelized software implementation using the 6 CPU cores available on our host machine. The FPGA-based system was roughly 5 to 10x faster than software when using 4 hash functions, depending on the K-mer length. The FPGA suffers no slow-down from hashing longer K-mers, while the software-based solution does. Furthermore, due to the use of a Bloom filter, K-mer length has no effect on the memory footprint of counting. Finally, we compared our FPGA-based counter to KMC 2, a state-of-the-art hash based counter. We achieved similar quality of results with a 9.31x to 17.6x performance advantage when using 2 hash functions for the Bloom filter, again depending on K-mer length. One interesting property of Bloom filters is the small number of hash functions often required to achieve a low false positive rate when table occupancy is reasonable.

Combining the speedup from these evaluations gives a more dramatic result. An FPGA “accelerated” K-mer counter that uses only DRAM would perform roughly the same as software, due to limited DRAM bandwidth. Not surprisingly, memory performance is critical to the viability of some accelerators. We believe that this result holds for other problems in the field of bioinformatics, particularly other areas of de novo assembly. This is not to say that CPUs equipped with non-traditional memories, such as the HBM2, will always be able to achieve similar results. In the case of K-mer counting the increased performance was possible specifically because of the HMC's high random accesses bandwidth combined with the parallel K-mer decompressing and hashing capabilities of the FPGA.

These specific results show that FPGA-based bioinformatics accelerators perform very well across a variety of applications. Some approaches, such as dynamic programing appear in multiple areas, allowing larger problems to fit in local FPGA memory to avoid the off-chip bottleneck. In other cases, higher random access rate memory (such as the HMC) can be used to avoid this bottleneck. As with many big data system, the greatest challenge when processing large quantities of short reads or other sequence data is often the various bandwidth bottlenecks, including disk, host memory, host to FPGA transfers and FPGA-attached memory. Properly managing these limitations is essential for any FPGA bioinformatics accelerator.

5.2 FPGA-ACCELERATED BIOINFORMATICS

After completing the work described above, we have come to the conclusion that the most difficult part of FPGA-based bioinformatics is not the design and development of accelerators. Instead it is achieving adoption into the pipelines used by biologists and others performing large amounts of processing on genomic data. These are a number of somewhat obvious reasons for these difficulties. FPGAs are expensive and a particularly hard sell if they can only accelerate a limited number of components in a long genomics pipeline. FPGA-based solutions are not as well established and may produce slightly different output, creating doubt to the quality of results. Furthermore, genomics software packages generally offer a tremendous number of options and are often employed slightly differently in each pipeline, making FPGA integration a challenging and labor intensive task. Introducing a new piece of hardware into large computer systems is also a big ask for IT that may already be stretched thin or reluctant to offer support for an unfamiliar system.

The difficulty of finding partners for FPGA-based bioinformatics accelerators, particularly in moving past the early trial phase, is not a new observation. However, we feel it is worth emphasizing as one of the most striking observations resulting from this work. To the best of our knowledge, success in achieving wider adoption of FPGAs for bioinformatics during the period this work was ongoing was achieved primarily by installing and heavily supporting FPGA systems in large numbers of bioinformatics labs, particularly by Convey Computer prior to their acquisition by Micron in 2015 [Chrysos12][Chrysos14].

The advent of public cloud computing services that include FPGAs for acceleration presents an intriguing solution to many of these problems. The cloud would make it much easier for biology groups around the globe to share in the upfront costs of pipeline integration for FPGA accelerators without committing to a large upfront purchase. Similarly, a library of accelerators could be developed by disparate sources with the ultimate goal of improving the performance of entire pipelines. The fact that these goals still seem distant, despite significant efforts from small companies as well as backing from much larger players such as Amazon, suggests there are still challenges in this area as well.

5.2.1 *Cloud Based Bioinformatics*

Public cloud services are a large and growing industry, with the total cloud market estimated at $175 billion for 2015, $204 billion in 2016 and growth continuing at a similar rate [Gartner, Inc.15]. One way that this growth is sustained is the movement of entire classes of computing from on-site or wholly owned servers to cloud based services, and one way to facilitate this shift is more heterogeneous clouds. As part of this trend, many cloud services offer GPU instances to their customers. More recently, the two largest providers of cloud infrastructure services, Amazon and Microsoft (who command a combined 42% of the market [Day16]), have announced products which will allow internal or external clients to use FPGAs as part of their cloud-based application [Caulfield16][Amazon Web Services, Inc.16].

For applications widely deployed on cloud-based FPGAs, the economic advantages of speedups that might otherwise be considered modest in the FPGA community can be significant. For example, during the early evaluation of their Catapult data center FPGA platform, Microsoft found that for an increase in server total cost of ownership (TCO) of under 30% they could achieve a 95% improvement in per sever Bing search ranking throughput [Putnam14]. For a large problem, such as Bing, this could lead to tremendous cost savings, despite the lack of multiple orders of magnitude speedup. As demonstrated in work analyzing ASIC cloud deployments, the non-recurring engineering cost, which is often higher for FPGA-based applications than pure software, can quickly become irrelevant as problems approach the size of Bing or other large cloud-scale applications [Magaki16].

One area with many applications of sufficient computational intensity and broad user base is genomics. Although processing NGS data seems like an ideal area for cloud computing, there has been a great deal of skepticism due to the specifics of genomic data, including privacy concerns, communication to computation ratio and difficulty parallelizing common bioinformatics algorithms [Schatz10]. Given the scale of the problem and the obvious economic incentives, there has been significant research into resolving these problems [Reid14]. We believe that FPGA-based genomics accelerators such as ours offer solutions to some of these problems.

5.3 Future Work

There are two distinct directions for future work in bioinformatics accelerators. First, the accelerators presented here can be extended in a number of ways. Second, and perhaps more promisingly, FPGA-based accelerators with specific considerations given to cloud deployment could greatly increase adoption and usefulness. We will now address the first of these directions.

There are multiple viable paths to improve our accelerated short read aligner. The use of a higher bandwidth memory, like the HMC, could allow for more of the alignment processes to take place on the FPGA, essentially returning to the V1 system with greatly improved performance. This could allow for alignment to run entirely on the FPGA with very high performance, while other parts of the user's pipeline run on the host simultaneously. If this could be done in an entirely streaming manner the total amount of access to the network or disk could be reduced, greatly increasing overall performance. Such a system could also scale well beyond our current system, which saturates host memory well before it uses all the FPGAs that could be attached. Such a system would also have to support accelerated CIGAR string generation, as these are often required by genomics pipelines.

Another area of future work is a version that supports spliced alignment. This would be very useful for RNA-Seq data, allowing discovery of new isoforms. Currently spliced aligners are very slow, even compared to other software aligners, making acceleration particularly desirable. Spliced alignment may also greatly increase classification accuracy for cases where the list of known isoforms are less complete.

Improvements for acceleration of ncRNA homology search are much more straightforward. Implementation of Forward and Inside are an absolute must, particularly because more recent versions of Infernal use these algorithms for all stages of filtering, almost entirely abandoning

Viterbi and CYK. Infernal, and particularly HMMER, also makes use of significantly optimized versions of profile HMMs and the Viterbi/Forward algorithm that achieve an order of magnitude speedup, or more, in software. Although it would increase complexity, any future version of an ncRNA homology accelerator would need to consider these developments.

Another approach would be to look at entirely different methods of modeling ncRNA families. These may be more amenable to FPGA acceleration given that current ncRNA family models are heavily based on floating point arithmetic. For example, one alternative to covariance is Hamming distance based models. Moving away from ncRNA homology, there are other uses for stochastic context-free grammars in bioinformatics. PPfold uses models much like CMs for RNA secondary structure prediction. There may be a place, particularly in the cloud, for a more general bioinformatics CM accelerator with support for a variety of applications.

For K-mer counting, future work would have to include a complete FPGA-based counter. This system could then improve performance through the use of atomic operations to reduce the HMC bandwidth required for each Bloom filter update. Alternatively, a high performance version of the system using more advanced hash tables, such as those found in KMC3, may achieve better performance overall. More advanced Bloom filters would also provide performance and memory footprint benefits, allowing for fewer partitions. Either of these approaches could be used as the basis for a full de novo short read assembler implemented on FPGAs. Finally, given industry movement towards longer reads for novel assemblies, it would be valuable to look at what portions of the approach used in this work might be applicable to other FPGA-based assemblers, if any.

5.3.1 *Future Work in the Cloud*

Given the size of the de novo assembly problem, it is one of the many areas where bioinformatics accelerators might need to migrate to the cloud in order to be viable. It is also an area where our work might be most directly applicable. Current work on de novo assembly for FPGAs lags behind other bioinformatics applications [Schmidt17] particularly because the size of the problem, and the difficulty inherent in graph partitioning, makes development of solutions on local FPGA platforms challenging.

Although de novo assembly is a difficult problem for FPGAs, cloud-based accelerators are already being deployed in other areas. Amazon, and their partners DNAnexus and Edico Genome offer a solution for some of the trickier parts of cloud based bioinformatics such as compliance and integration into genomics pipelines [Friedman17]. One advantage of this framework is the ability to integrate other FPGA accelerators, such as ours, easing the transition for biologists who already run one or more applications on Amazon EC2 F1 instances. As an additional benefit, in a field where shipping hard drives can be a cost effective means of transmitting trivially parallel short read data, the increased computation density of servers equipped with FPGAs can greatly reduce network traffic in the cloud provider's datacenter.

The large amount of DRAM attached to each FPGA (64GB for Amazon F1) also opens up possibilities for some applications, such as assembly, but the bandwidth may still not be sufficient for aligners like the one presented above with multiple tables. However, Alibaba's rumored F3 instance, and presumably future EC2 instances and those from other providers as well, will include FPGAs such as the Virtex UltraScale+ VU31P with integrated HBM DRAM. Cloud adoption of the HMC, HBM or other non-traditional memories will greatly increase the variety of practical bioinformatics accelerators, as demonstrated with our K-mer counter.

Given all of these developments, cloud-based FPGAs seem to finally be poised to fulfil the promise of wide adoption of FPGA accelerated bioinformatics applications. Easy pipeline integration and boards with support for high memory bandwidth can bring a wide array of powerful

FPGA-based genomics solutions, such as ours, to biologists, other scientists and even patient care facilities.

BIBLIOGRAPHY

[Abbas10] N. Abbas, S. Derrien, S. Rajopadhye, P. Quinton, and others, "Accelerating HMMER on FPGA using Parallel Prefixes and Reductions," 2010.

[Ahdesmäki16] M. J. Ahdesmäki, S. R. Gray, J. H. Johnson, and Z. Lai, "Disambiguate: An open-source application for disambiguating two species in next generation sequencing data from grafted samples," *F1000Research*, vol. 5, 2016.

[Alkan09] C. Alkan et al., "Personalized copy-number and segmental duplication maps using next-generation sequencing," *Nature Genetics*, vol. 41, no. 10, pp. 1061–1067, 2009.

[Aluru14] S. Aluru and N. Jammula, "A Review of Hardware Acceleration for Computational Genomics," 2014.

[Amazon Web Services, Inc.16] Amazon Web Services, Inc., "Amazon EC2 F1 Instances (Preview)," 2016. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/. [Accessed: 17-Jan-2017].

[Arram16] J. Arram, T. Kaplan, W. Luk, and P. Jiang, "Leveraging FPGAs for Accelerating Short Read Alignment," *IEEE/ACM transactions on computational biology and bioinformatics*, 2016.

[Arram13] J. Arram, K. H. Tsoi, W. Luk, and P. Jiang, "Reconfigurable acceleration of short read mapping," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, 2013, pp. 210–217.

[Bakos10] J. D. Bakos, "High-performance heterogeneous computing with the Convey HC-1," *Computing in Science \& Engineering*, vol. 12, no. 6, pp. 80–87, 2010.

[Bakos07] J. D. Bakos, P. E. Elenis, and J. Tang, "FPGA acceleration of phylogeny reconstruction for whole genome data," in *Bioinformatics and Bioengineering, 2007. BIBE 2007. Proceedings of the 7th IEEE International Conference on*, 2007, pp. 888–895.

[Becher15] A. Becher, D. Ziener, K. Meyer-Wegener, and J. Teich, "A co-design approach for accelerated SQL query processing via FPGA-based data filtering," in *Field Programmable Technology (FPT), 2015 International Conference on*, 2015, pp. 192–195.

[Behringer07] R. R. Behringer, "Human-Animal Chimeras in Biomedical Research," *Cell Stem Cell*, vol. 1, no. 3, pp. 259–262, 2007.

[Bentley08] D. R. Bentley et al., "Accurate whole human genome sequencing using reversible terminator chemistry," *Nature*, vol. 456, no. 7218, pp. 53–59, 2008.

[Bloom70] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[Bourque04] G. Bourque, P. A. Pevzner, and G. Tesler, “Reconstructing the genomic architecture of ancestral mammals: lessons from human, mouse, and rat genomes,” *Genome research*, vol. 14, no. 4, pp. 507–516, 2004.

[Bradford13] J. R. Bradford et al., “RNA-Seq differentiates tumour and host mRNA expression changes induced by treatment of human tumour xenografts with the VEGFR tyrosine kinase inhibitor Cediranib,” *PloS One*, vol. 8, no. 6, p. e66003, 2013.

[Burge12] S. W. Burge et al., “Rfam 11.0: 10 years of RNA families,” *Nucleic Acids Research*, 2012.

[Caulfield16] A. M. Caulfield et al., “A cloud-scale acceleration architecture,” in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, 2016, pp. 1–13.

[Chamberlain07] R. D. Chamberlain et al., “Application development on hybrid systems,” in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007, p. 50.

[Chao17] J. R. Chao et al., “Transplantation of human embryonic stem cell-derived retinal cells into the subretinal space of a non-human primate,” *Translational vision science \& technology*, vol. 6, no. 3, pp. 4–4, 2017.

[Chen12] Y. Chen, B. Schmidt, and D. L. Maksell, “An FPGA aligner for short read mapping,” in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, 2012, pp. 511–514.

[Chikhi13a] R. Chikhi and P. Medvedev, “Informed and automated k-mer size selection for genome assembly,” *Bioinformatics*, pp. 31–37, 2013.

[Chikhi13b] R. Chikhi and G. Rizk, “Space-efficient and exact de Bruijn graph representation based on a Bloom filter,” *Algorithms for Molecular Biology*, vol. 8, no. 1, p. 22, 2013.

[Chow91] E. Chow, T. Hunkapiller, J. Peterson, and M. Waterman, “Biological information signal processor,” in *Application Specific Array Processors, 1991. Proceedings of the International Conference on*, 1991, pp. 144–160.

[Chrysos12] G. Chrysos et al., “Opportunities from the use of FPGAs as platforms for bioinformatics algorithms,” in *Bioinformatics \& Bioengineering (BIBE), 2012 IEEE 12th International Conference on*, 2012, pp. 559–565.

[Chrysos14] G. Chrysos et al., “Reconfiguring the bioinformatics computational spectrum: Challenges and opportunities of fpga-based bioinformatics acceleration platforms,” *IEEE*

Design \& Test, vol. 31, no. 1, pp. 62–73, 2014.

[Conway12] T. Conway et al., “Xenome—a tool for classifying reads from xenograft samples,” *Bioinformatics*, vol. 28, no. 12, pp. i172–i178, 2012.

[Dai18] W. Dai, J. Liu, Q. Li, W. Liu, Y.-X. Li, and Y.-Y. Li, “A comparison of next-generation sequencing analysis methods for cancer xenograft samples,” *Journal of Genetics and Genomics*, 2018.

[Day16] M. Day, “How Seattle became ‘Cloud City’: Amazon and Microsoft are leading a tech revolution,” *The Seattle Times*, vol. Dec., no. 9th, p. A1, 2016.

[Deorowicz15] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, “KMC 2: Fast and resource-frugal k-mer counting,” *Bioinformatics*, vol. 31, no. 10, pp. 1569–1576, 2015.

[Derrien07] S. Derrien and P. Quinton, “Parallelizing HMMER for hardware acceleration on FPGAs,” in *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, 2007, pp. 10–17.

[Derrien10] S. Derrien and P. Quinton, “Hardware Acceleration of HMMER on FPGAs,” *Journal of Signal Processing Systems*, vol. 58, no. 1, pp. 53–67, 2010.

[Dharmapurikar03] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, “Deep packet inspection using parallel bloom filters,” in *High Performance Interconnects, 11th Symposium on*, 2003, pp. 44–51.

[Dobai15] R. Dobai and J. Korenek, “Evolution of Non-Cryptographic Hash Function Pairs for FPGA-Based Network Applications,” in *Computational Intelligence, 2015 IEEE Symposium Series on*, 2015, pp. 1214–1219.

[Eddy02] S. Eddy, “A memory-efficient dynamic programming algorithm for optimal alignment of a sequence to an RNA secondary structure,” *BMC Bioinformatics*, vol. 3, no. 1, p. 18, 2002.

[Eddy98] S. R. Eddy, “Profile hidden Markov models.,” *Bioinformatics*, vol. 14, no. 9, p. 755, 1998.

[Eddy11] S. R. Eddy, “Accelerated profile HMM searches,” *PLoS Computational Biology*, vol. 7, no. 10, p. e1002195, 2011.

[Eddy94] S. R. Eddy and R. Durbin, “RNA sequence analysis using covariance models,” *Nucleic Acids Research*, vol. 22, no. 11, pp. 2079–2088, 1994.

[Eddy08] S. R. Eddy, “A probabilistic model of local sequence alignment that simplifies statistical significance estimation.,” *PLoS computational biology*, vol. 4, no. 5, p. e1000069, 2008.

[Eusse Giraldo10] J. F. Eusse Giraldo, N. Moreano, R. P. Jacobi, and A. C. M. A. de Melo, "A HMMER hardware accelerator using divergences," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2010, pp. 405–410.

[Fan14] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies*, 2014, pp. 75–88.

[Fan00] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000.

[Finn16] R. D. Finn et al., "The Pfam protein families database: towards a more sustainable future," *Nucleic acids research*, vol. 44, no. D1, pp. D279–D285, 2016.

[Forney Jr73] G. D. Forney Jr, "The Viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[Friedman17] A. Friedman, "How DNAnexus and Edico Genome are Powering Precision Medicine on Amazon Web Services (AWS)," 2017. [Online]. Available: https://aws.amazon.com/blogs/apn/how-dnanexus-and-edico-genome-are-powering-precision-medicine-on-amazon-web-services-aws/. [Accessed: Jul-2018].

[Gartner, Inc.15] Gartner, Inc., "Forecast: Public Cloud Services, Worldwide, 2013-2019, 4Q15 Update," 2015. [Online]. Available: http://www.gartner.com/newsroom/id/3188817. [Accessed: 17-Jan-2016].

[Gokhale91] M. Gokhale et al., "Building and using a highly parallel programmable logic array," *Computer*, vol. 24, no. 1, pp. 81–89, 1991.

[Gokhale15] M. Gokhale, S. Lloyd, and C. Macaraeg, "Hybrid memory cube performance characterization on data-centric workloads," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, 2015, p. 7.

[Gosselin-Lavigne15] M. A. Gosselin-Lavigne, H. Gonzalez, N. Stakhanova, and A. A. Ghorbani, "A Performance Evaluation of Hash Functions for IP Reputation Lookup Using Bloom Filters," in *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, 2015, pp. 516–521.

[Hach10] F. Hach et al., "mrsFAST: a cache-oblivious algorithm for short-read mapping," *Nature Methods*, vol. 7, no. 8, pp. 576–577, 2010.

[Haselman05] M. Haselman, M. Beauchamp, A. Wood, S. Hauck, K. Underwood, and K. S. Hemmert, "A comparison of floating point and logarithmic number systems for FPGAs," in *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE*

Symposium on, 2005, pp. 181–190.

[Hatem13] A. Hatem, D. Bozdağ, A. E. Toland, and Ü. V. Çatalyürek, “Benchmarking short sequence mapping tools,” *BMC Bioinformatics*, vol. 14, no. 1, p. 184, 2013.

[Hauck08] S. Hauck and A. DeHon, *Reconfigurable computing*. Burlington, Massachusetts, USA: Morgan Kauffman, 2008.

[Hoang92] D. T. Hoang and D. P. Lopresti, “FPGA implementation of systolic sequence alignment,” in *International Workshop on Field Programmable Logic and Applications*, 1992, pp. 183–191.

[Homer09] 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, 2009.

[Hybrid Memory Cube Consortium14] Hybrid Memory Cube Consortium, “Hybrid Memory Cube Specification 1.1,” 2014.

[Hybrid Memory Cube Consortium15] Hybrid Memory Cube Consortium, “Hybrid Memory Cube Specification 2.1,” 2015.

[Jablin10] T. B. Jablin, Y. Zhang, J. A. Jablin, J. Huang, H. Kim, and D. I. August, “Liberty queues for epic architectures,” in *Proceedings of the Eigth Workshop on Explicitly Parallel Instruction Computer Architectures and Compiler Technology (EPIC)*, 2010.

[Jackman17] S. D. Jackman et al., “ABySS 2.0: resource-efficient assembly of large genomes using a Bloom filter,” *Genome research*, vol. 27, no. 5, pp. 768–777, 2017.

[Kalvari17] I. Kalvari et al., “Rfam 13.0: shifting to a genome-centric resource for non-coding RNA families,” *Nucleic Acids Research*, vol. 46, no. D1, pp. D335–D342, 2017.

[Karl08] M. O. Karl, S. Hayes, B. R. Nelson, K. Tan, B. Buckingham, and T. A. Reh, “Stimulation of neural regeneration in the mouse retina,” *Proceedings of the National Academy of Sciences*, vol. 105, no. 49, pp. 19508–19513, 2008.

[Kellis14] M. Kellis et al., “Defining functional DNA elements in the human genome.,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 111, no. 17, pp. 6131–8, 2014.

[Khandelwal17] G. Khandelwal et al., “Next-Generation Sequencing Analysis and Algorithms for PDX and CDX Models,” *Molecular Cancer Research*, 2017.

[Kim15] D. Kim, B. Langmead, and S. L. Salzberg, “HISAT: a fast spliced aligner with low memory requirements,” *Nature methods*, vol. 12, no. 4, p. 357, 2015.

[Kim11] M. Kim, “Accelerating Next Generation Genome Reassembly in FPGAs: Alignment Using Dynamic Programming Algorithms,” 2011.

[Knodel11] O. Knodel, T. B. Preußer, and R. G. Spallek, “Next-generation massively parallel short-read mapping on FPGAs,” in *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, 2011, pp. 195–201.

[Koren17] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, N. H. Bergman, and A. M. Phillippy, “Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation,” *Genome research*, vol. 27, no. 5, pp. 722–736, 2017.

[Kuon07] I. Kuon and J. Rose, “Measuring the gap between FPGAs and ASICs,” *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 26, no. 2, pp. 203–215, 2007.

[Lamba06] D. A. Lamba, M. O. Karl, C. B. Ware, and T. A. Reh, “Efficient generation of retinal progenitor cells from human embryonic stem cells,” *Proceedings of the National Academy of Sciences*, vol. 103, no. 34, pp. 12769–12774, 2006.

[Lander01] E. S. Lander et al., “Initial sequencing and analysis of the human genome,” *Nature*, vol. 409, no. 6822, pp. 860–921, 2001.

[Langmead12] B. Langmead and S. L. Salzberg, “Fast gapped-read alignment with Bowtie 2,” *Nature Methods*, vol. 9, no. 4, pp. 357–359, 2012.

[Lemoine95] E. Lemoine and D. Merceron, “Run time reconfiguration of FPGA for scanning genomic databases,” in *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*, 1995, pp. 90–98.

[Li09a] H. Li and R. Durbin, “Fast and accurate short read alignment with Burrows-Wheeler transform,” *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[Li09b] H. Li et al., “The sequence alignment/map format and SAMtools,” *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[Li09c] H. Li et al., “The Sequence Alignment/Map format and SAMtools.,” *Bioinformatics (Oxford, England)*, vol. 25, no. 16, pp. 2078–9, 2009.

[Li13] H. Li, “Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM,” *arXiv preprint arXiv:1303.3997*, 2013.

[Li10] H. Li and R. Durbin, “Fast and accurate long-read alignment with Burrows-Wheeler transform,” *Bioinformatics*, vol. 26, no. 5, pp. 589–595, 2010.

[Li09d] R. Li et al., “SOAP2: an improved ultrafast tool for short read alignment,” *Bioinformatics*, vol. 25, no. 15, pp. 1966–1967, 2009.

[Li15] Y. Li and others, “MSPKmerCounter: a fast and memory efficient approach for k-mer counting,” *arXiv preprint arXiv:1505.06550*, 2015.

[Lockwood03] J. W. Lockwood, J. Moscola, M. Kulig, D. Reddick, and T. Brooks, “Internet worm and virus protection in dynamically reconfigurable hardware,” in *Proceedings of the Military and Aerospace Programmable Logic Device Conference*, 2003.

[Lyons09] M. J. Lyons and D. Brooks, “The design of a Bloom filter hardware accelerator for ultra low power systems,” in *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*, 2009, pp. 371–376.

[Magaki16] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor, “ASIC clouds: specializing the datacenter,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016, pp. 178–190.

[Magistri17] M. Magistri and D. Velmeshev, “Identification of Long Noncoding RNAs Associated to Human Disease Susceptibility,” *Promoter Associated RNA: Methods and Protocols*, pp. 197–208, 2017.

[Mamun16] A.-A. Mamun, S. Pal, and S. Rajasekaran, “KCMBT: A k-mer Counter based on Multiple Burst Trees,” *Bioinformatics*, p. btw345, 2016.

[Manners10] D. Manners, “FPGA Market Soaring To $4bn In 2010, says Gavrielov,” 2010. [Online]. Available: https://www.electronicsweekly.com/news/products/fpga-news/fpga-market-soaring-to-4bn-in-2010-says-gavrielov-2010-05/. [Accessed: 22-Dec-2017].

[Marçais11] G. Marçais and C. Kingsford, “A fast, lock-free approach for efficient parallel counting of occurrences of k-mers,” *Bioinformatics*, vol. 27, no. 6, pp. 764–770, 2011.

[McVicar13] N. McVicar, W. L. Ruzzo, and S. Hauck, “Accelerating ncRNA Homology Search with FPGAs,” *to appear in Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, 2013.

[McVicar18] N. McVicar, A. Hoshino, A. La Torre, T. A. Reh, W. L. Ruzzo, and S. Hauck, “FPGA Acceleration of Short Read Alignment,” *arXiv preprint arXiv:1805.00106*, 2018.

[McVicar17] N. McVicar, C.-C. Lin, and S. Hauck, “K-mer counting using Bloom filters with an FPGA-attached HMC,” in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, 2017, pp. 203–210.

[Melsted11] P. Melsted and J. K. Pritchard, “Efficient counting of k-mers in DNA sequences using a bloom filter.,” *BMC Bioinformatics*, vol. 12, p. 333, 2011.

[Micron Technologies17] I. Micron Technologies, “SB-800,” 2017. [Online]. Available: http://picocomputing.com/ex-800-blade-server/. [Accessed: 27-Dec-2017].

[Micron Technology15] I. Micron Technology, “HMC Controller IP,” 2015. [Online]. Available: http://picocomputing.com/wp-content/uploads/2015/06/HMC-Controller-IP-Overview.pdf. [Accessed: 09-Sep-2015].

[Milo13] R. Milo, “What is the total number of protein molecules per cell volume? A call to rethink some published values.,” *BioEssays : news and reviews in molecular, cellular and developmental biology*, vol. 35, no. 12, pp. 1050–5, 2013.

[Mitzenmacher05] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[Moscola10] J. Moscola, R. K. Cytron, and Y. H. Cho, “Hardware-accelerated RNA secondary-structure alignment,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 3, no. 3, pp. 1–44, 2010.

[Movahedi12] N. S. Movahedi, E. Forouzmand, and H. Chitsaz, “De novo co-assembly of bacterial genomes from multiple single cells,” in *Bioinformatics and Biomedicine (BIBM), 2012 IEEE International Conference on*, 2012, pp. 1–5.

[Nawrocki09a] E. P. Nawrocki and S. R. Adviser-Eddy, “Structural RNA homology search and alignment using covariance models,” 2009.

[Nawrocki09b] E. P. Nawrocki and S. R. Eddy, “Computational identification of functional RNA homologs in metagenomic data,” *Manuscript submitted*, 2009.

[Nawrocki09c] E. P. Nawrocki, D. L. Kolbe, and S. R. Eddy, “Infernal 1.0: inference of RNA alignments,” *Bioinformatics*, vol. 25, no. 10, pp. 1335–1337, 2009.

[NHGRI10] NHGRI, “The Human Genome Project Completion: Frequently Asked Questions,” 2010. [Online]. Available: http://www.genome.gov/11006943. [Accessed: 13-Oct-2014].

[Oliver08] T. Oliver, L. Y. Yeow, and B. Schmidt, “Integrating FPGA acceleration into HMMer,” *Parallel Computing*, vol. 34, no. 11, pp. 681–691, 2008.

[Olson11] C. Olson, “An FPGA Acceleration of Short Read Human Genome Mapping,” *Department of Electrical Engineering, University of Washington. Master’s Thesis*, 2011.

[Olson12] C. B. Olson et al., “Hardware Acceleration of Short Read Mapping,” in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, 2012, pp. 161–168.

[Pagh05] A. Pagh, R. Pagh, and S. S. Rao, “An optimal Bloom filter replacement,” in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, 2005, pp.

823–829.

[Pan08] Q. Pan, O. Shai, L. J. Lee, B. J. Frey, and B. J. Blencowe, “Deep surveying of alternative splicing complexity in the human transcriptome by high-throughput sequencing,” *Nature Genetics*, vol. 40, no. 12, pp. 1413–1415, 2008.

[Pawlowski11] J. T. Pawlowski, “Hybrid memory cube (HMC),” in *Hot Chips*, 2011, vol. 23.

[Peng12] Y. Peng, H. C. M. Leung, S. M. Yiu, and F. Y. L. Chin, “IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth.,” *Bioinformatics (Oxford, England)*, vol. 28, no. 11, pp. 1420–8, 2012.

[Pico Computing, Inc.14] Pico Computing, Inc., “Xilinx and Altera FPGA-based HPC and embedded modules; M-503,” 2014. [Online]. Available: http://picocomputing.com/products/hpc-modules/m-503/. [Accessed: 13-Oct-2014].

[Poller17] W. Poller et al., “Non-coding RNAs in cardiovascular diseases: diagnostic and therapeutic perspectives,” *European Heart Journal*, p. ehx165, 2017.

[Putnam14] A. Putnam et al., “A reconfigurable fabric for accelerating large-scale datacenter services,” in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 13–24.

[Quail12] M. A. Quail et al., “A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers,” *BMC Genomics*, vol. 13, no. 1, p. 1, 2012.

[Ramakrishna97] M. Ramakrishna and J. Zobel, “Performance in Practice of String Hashing Functions.,” in *DASFAA*, 1997, pp. 215–224.

[Ramalingam16] S. Ramalingam, “HBM package integration: Technology trends, challenges and applications,” in *Hot Chips 28 Symposium (HCS), 2016 IEEE*, 2016, pp. 1–17.

[Reid14] J. G. Reid et al., “Launching genomics into the cloud: deployment of Mercury, a next generation sequence analysis pipeline,” *BMC Bioinformatics*, vol. 15, no. 1, p. 1, 2014.

[Research16] G. V. Research, “Field Programmable Gate Array (FPGA) Market Analysis By Technology (SRAM, EEPROM, Antifuse, Flash), By Application (Consumer Electronics, Automotive, Industrial, Data Processing, Military & Aerospace, Telecom), And Segment Forecasts, 2014 - 2024,” 2016. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/fpga-market. [Accessed: 22-Dec-2017].

[Research11] Research and Markets, “Research and Markets: Global Field-Programmable Gate Array (FPGA) Market 2010-2014 - Interest in High Bandwidth Devices is Driving Demand for Field Programmable Gate Array,” 2011. [Online]. Available:

https://www.businesswire.com/news/home/20110401005410/en/Research-Markets-Global-Field-Programmable-Gate-Array-FPGA. [Accessed: 23-Dec-2017].

[Rizk13] G. Rizk, D. Lavenier, and R. Chikhi, "DSK: k-mer counting with very low memory usage," *Bioinformatics*, pp. 652–3, 2013.

[Roberts04] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004.

[Salikhov13] K. Salikhov, G. Sacomoto, G. Kucherov, and others, "Using cascading Bloom filters to improve the memory usage for de Brujin graphs.," in *WABI*, 2013, pp. 364–376.

[Schatz10] M. C. Schatz, B. Langmead, and S. L. Salzberg, "Cloud computing and the DNA data race," *Nature Biotechnology*, vol. 28, no. 7, p. 691, 2010.

[Schmidt17] B. Schmidt and A. Hildebrandt, "Next-generation sequencing: big data meets high performance computing," *Drug discovery today*, vol. 22, no. 4, pp. 712–717, 2017.

[Schwarzer17] A. Schwarzer et al., "The non-coding RNA landscape of human hematopoiesis and leukemia," *Nature communications*, vol. 8, no. 1, p. 218, 2017.

[Shang14] J. Shang, F. Zhu, W. Vongsangnak, Y. Tang, W. Zhang, and B. Shen, "Evaluation and comparison of multiple aligners for next-generation sequencing data analysis," *BioMed research international*, vol. 2014, 2014.

[Shaw14] D. E. Shaw et al., "Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2014, pp. 41–53.

[Sogabe17] Y. Sogabe and T. Maruyama, "A fast and accurate FPGA system for short read mapping based on parallel comparison on hash table," *IEICE Transactions on Information and Systems*, vol. 100, no. 5, pp. 1016–1025, 2017.

[Starobinski03] D. Starobinski, A. Trachtenberg, and S. Agarwal, "Efficient PDA synchronization," *IEEE Transactions on Mobile Computing*, vol. 2, no. 1, pp. 40–51, 2003.

[Storaasli07] O. Storaasli, W. Yu, D. Strenski, and J. Maltby, "Performance evaluation of FPGA-based biological applications," *Cray User Group*, 2007.

[Sükösd11] Z. Sükösd, B. Knudsen, M. Vaerum, J. Kjems, and E. Andersen, "Multithreaded comparative RNA secondary structure prediction using stochastic context-free grammars," *BMC bioinformatics*, vol. 12, no. 1, p. 103, 2011.

[Sun12] Y. Sun, O. Aljawad, J. Lei, and A. Liu, “Genome-scale NCRNA homology search using a Hamming distance-based filtration strategy,” *BMC Bioinformatics*, vol. 13, no. Suppl 3, p. S12, 2012.

[Sun09] Y. Sun, P. Li, G. Gu, Y. Wen, Y. Liu, and D. Liu, “Accelerating HMMer on FPGAs using systolic array based architecture,” in *Parallel \& Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009, pp. 1–8.

[Taber14] K. A. J. Taber, B. D. Dickinson, and M. Wilson, “The promise and challenges of next-generation genome sequencing for clinical care,” *JAMA internal medicine*, vol. 174, no. 2, pp. 275–280, 2014.

[Takagi09] T. Takagi and T. Maruyama, “Accelerating HMMER search using FPGA,” in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, 2009, pp. 332–337.

[Tso14] K.-Y. Tso, S. D. Lee, K.-W. Lo, and K. Y. Yip, “Are special read alignment strategies necessary and cost-effective when handling sequencing reads from patient-derived tumor xenografts?,” *BMC genomics*, vol. 15, no. 1, p. 1172, 2014.

[Venter01] J. C. Venter et al., “The sequence of the human genome,” *Science*, vol. 291, no. 5507, pp. 1304–1351, 2001.

[Waidyasooriya16] H. M. Waidyasooriya and M. Hariyama, “Hardware-acceleration of short-read alignment based on the Burrows-Wheeler transform,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1358–1372, 2016.

[Walter17] M. C. Walter et al., “MinION as part of a biomedical rapidly deployable laboratory,” *Journal of biotechnology*, vol. 250, pp. 16–22, 2017.

[Weinberg04] Z. Weinberg and W. L. Ruzzo, “Exploiting conserved structure for faster annotation of non-coding RNAs without loss of accuracy,” *Bioinformatics*, vol. 20, no. suppl 1, p. i334, 2004.

[Weinberg06] Z. Weinberg and W. L. Ruzzo, “Sequence-based heuristics for faster annotation of non-coding RNA families,” *Bioinformatics*, vol. 22, no. 1, pp. 35–39, 2006.

[Wetterstrand14] K. Wetterstrand, “DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP),” 2014. [Online]. Available: http://www.genome.gov/sequencingcosts. [Accessed: 13-Oct-2014].

[Wilken16] M. S. Wilken and T. A. Reh, “Retinal regeneration in birds and mice,” *Current opinion in genetics \& development*, vol. 40, pp. 57–64, 2016.

[Wu17] J. Wu et al., “Interspecies Chimerism with Mammalian Pluripotent Stem Cells,” *Cell*, vol. 168, no. 3, pp. 473–486, 2017.

[Wu10] T. D. Wu and S. Nacu, “Fast and SNP-tolerant detection of complex variants and splicing in short reads,” *Bioinformatics*, vol. 26, no. 7, pp. 873–881, 2010.

[Xia10] F. Xia, Y. Dou, D. Zhou, and X. Li, “Fine-grained parallel RNA secondary structure prediction using SCFGs on FPGA,” *Parallel Computing*, vol. 36, no. 9, pp. 516–530, 2010.

[Xilinx12] I. Xilinx, “FPGA and ASIC Technology Comparison,” 2012. [Online]. Available: http://www.xilinx.com/training/downloads/what-is-the-difference-between-an-fpga-and-an-asic.pptx. [Accessed: Oct-2017].

[Yamaguchi13] F. Yamaguchi and H. Nishi, “Hardware-based hash functions for network applications,” in *Networks (ICON), 2013 19th IEEE International Conference on*, 2013, pp. 1–6.

[Younger67] D. H. Younger, “Recognition and parsing of context-free languages in time n^3,” *Information and control*, vol. 10, no. 2, pp. 189–208, 1967.

[Zerbino08] D. R. Zerbino and E. Birney, “Velvet: algorithms for de novo short read assembly using de Bruijn graphs,” *Genome research*, vol. 18, no. 5, pp. 821–829, 2008.

[Zhang07] P. Zhang, G. Tan, and G. 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*, 2007, pp. 39–48.

VITA

Nathaniel McVicar received his Bachelor of Science in Electrical and Computer Engineering from Washington University in St. Louis in 2006, after which he developed FPGA accelerated market data platforms at Exegy, Inc. He received a Master of Science in Electrical Engineering from the University of Washington in 2011, where he was an NSF Graduate Research Fellow. His research included a VLIW execution model for CGRAs, as well as FPGA-based bioinformatics systems. During his time in graduate school he was a contractor at Microsoft Research, a consultant and an instructor in Vietnam. Nathaniel completed his Doctor of Philosophy in Electrical Engineering in 2018.