FPGA ACCELERATION OF SHORT READ ALIGNMENT WITH HIGH-LEVEL SYNTHESIS

BY

DANIEL E. CHEN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor Deming Chen

ABSTRACT

With the introduction of next-generation sequencing (NGS) technologies, DNA sequencing is becoming an increasingly widespread process. When performed on human patients, it can allow for the prediction and prevention of diseases. An essential part of this bioinformatics pipeline is *short read alignment*, which refers to aligning short fragments of DNA to the large and expansive reference genome. This can be a very time-consuming process with much room for improvement. This thesis improves on Bowtie 2, an aligner that is already very popular and high-performing. Through the use of OpenCL, it is possible to parallelize this application for both GPU and FPGA by using the same code. Several different levels of parallelism are implemented in order to achieve speedup on Bowtie 2.

ACKNOWLEDGMENTS

I would like to thank Professor Deming Chen, my adviser, for welcoming me to his research group and providing academic guidance during my time as a master's student. I would also like to thank Yan Yan, who has contributed tremendously to this project. Additionally, I want to thank Sitao Huang, who has always been eager to help. Lastly, I want to thank my friends and family for their support throughout the years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Overview

Humans are an extremely complex and varied species, with many different expressions for the same underlying features. Deoxyribonucleic acid, commonly known as DNA, serves as the blueprint for each individual human. This means that it is not only responsible for those characteristics that make us human, but also the variations that make each of us unique. The majority of our genetic code is encoded in DNA using an alphabet with four unique elements. Furthermore, the human DNA is encoded in a double helix strand, so each element in the code has a complementary element. Together they are known as base pairs (bp). Knowledge of a person's DNA sequence has great implications. For example, the onset of some diseases can be predicted through the presence of mutations at certain key locations in the genome. By identifying this mutation in specific people, we are able to take preventative measures that can prolong a person's life or even completely mitigate these diseases.

1.2 Sequencing

In order to properly interpret a person's genetic code, his/her DNA must be sequenced. With improving DNA sequencing technology, it has become cheaper and more accessible for a person to have his/her DNA sequenced. As a result, DNA sequencing is becoming an increasingly popular process that has the potential for great utility in the medical field. The most modern developments in sequencing are known as next-generation sequencing (NGS), which refers to the latest high-throughput technologies, including Illumina and Roche sequencing. These processes take the physical strands of DNA from human beings as the input and seek to convert them into a digital format, so that they may be analyzed. The outputs of these sequencers are short fragments known as reads, which can vary in length from under 50 bp to several hundred bp.

Before much sense can be made of these reads, they must be aligned to the reference human genome, which is approximately 3.2 billion bp long. There are many different software tools developed to achieve this task. Examples of popular short read aligners include Bowtie 2 and Burrows Wheeler Aligner (BWA). Due to the independent nature of aligning each read as well as the seeds within each read, there are massive opportunities for parallelism in these algorithms. With the rapidly emerging paradigm of parallel computing through the use of hardware such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs), it is only natural that these short read aligners should reap the benefits of these hardware platforms. This thesis focuses on the parallelization of the Bowtie 2 aligner through the use of FPGA.

CHAPTER 2

BACKGROUND

2.1 DNA

Described in the overview, DNA encoding consists of four elements, known as nucleobases or bases. These four bases are guanine (G), cytosine (C), adenine (A), and thymine (T). DNA is structured as a double helix, which means that each DNA molecule consists of two complementary strands. Therefore, with knowledge of the sequence of one strand, it is also possible to deduce the contents of the other. The mapping between the bases on one strand and the complementary strand are one-to-one and symmetrical. G is always matched with C, and A is always matched with T. The length of DNA varies greatly between different species. Primarily, we will deal with the human genome, which is approximately 3.2 billion bp long. By comparing a sequenced human genome to the reference human genome, it is possible to check for mutations at key positions, leading to important insights about one's health.

2.2 Sequencing Technologies

Before DNA can be analyzed on a digital basis, it must go through a sequencing method. One important consequence of utilizing these sequencing technologies is that the sequencers will fragment the DNA into reads, rather than recovering the entire sequence altogether. The read length varies between technologies. There are a few common technologies that are used for next-generation sequencing. They are as follows:

1. Illumina (Solexa) Sequencing

2. SOLiD Sequencing

3. 454 Pyrosequencing
4. Ion Torrent Semiconductor Sequencing

Each technology has its unique accuracy, read length, speed, and cost trade-offs. A rigorous discussion of the different sequencing technologies is omitted, since the technical details of the sequencing technology are not of particular concern for the performance of the short read aligner. Some particularly important factors directly related to the sequencing technology, when considering a short read aligner's performance, are the read length, whether the reads are single-end or paired-end, and the sequencing error rate.

2.3 Alignment Overview

After the reads are generated, they need to be *aligned* to the reference genome. The alignment process for a read seeks to determine which specific location with the reference genome the read corresponds to. Due to the nature of biology, often there can be multiple compelling locations that a read might align to. The aligner must account for this possibility and determine the best possible alignment location for a read. It is this process which makes the reads useful, giving them a meaningful context and basis for comparison. Without this context, one would be challenged to make much sense from the sequence contained within a read. Once the read is aligned, one is able see what insertions, deletions, or substitutions, if any, exist, by comparing the aligned sequence to the reference sequence. The following shows a general algorithm for a short read aligner operating on a single read:

1. Generate all seeds for a read.
2. Find all exact matches for the seed in the reference genome.
3. Extend each seed hit in the reference genome to reflect the full length of the original read.
4. Determine the quality of alignment between each read and candidate alignment location.
5. Report the best alignment within all the candidate alignments.

2.3.1 Indexing the Reference Genome

A reference genome is often billions of base pairs long, depending on the species. It can be quite costly to search for a string within such a reference genome without any indexing. There are two different schemes for indexing reference genomes that are seen in the most popular aligners today.

Each reference genome must go through a relatively time-consuming indexing process, but only once. Afterwards, the same index can be reused for each read that is being aligned to the same reference genome. So, this is a one-time cost and generally not a significant factor when considering the performance of alignment.

Burrows Wheeler Transform

The most commonly used method for indexing the reference genome utilizes what is known as the Burrows Wheeler Transform (BWT) [1], which is closely related with the FM-Index. This indexing method works in great synergy with read alignment. It allows for a highly efficient method to obtain the location of all exact matches of a specific search sequence within the reference genome.

The Burrows Wheeler Transform and FM-index search relies on several different structures to make it work:

1. Wavelet Tree for rank queries
2. Suffix Array to allow for traversing the resultant ranges
3. Burrows Wheeler Transform to allow for backwards search
4. Character Occurrence Table to allow for character count queries

Figure 2.1 is a demonstration of the BWT being generated. The example string will is “ILLINOIS”. It is typical to use a symbol for marking the end of the string. In this example, “$” is used. The first step is to create a matrix that shows every possible circular shift of the characters in the string. Then, the rows are sorted alphabetically according to the first character in each row, as shown in Figure 2.2

I	L	L	I	N	O	I	S	$
$	I	L	L	I	N	O	I	S
S	$	I	L	L	I	N	O	I
I	S	$	I	L	L	I	N	O
O	I	S	$	I	L	L	I	N
N	O	I	S	$	I	L	L	I
I	N	O	I	S	$	I	L	L
L	I	N	O	I	S	$	I	L

Figure 2.1: Burrows-Wheeler Transform Circular Shifts

I	L	L	I	N	O	I	S	$
I	N	O	I	S	$	I	L	L
I	S	$	I	L	L	I	N	O
L	I	N	O	I	S	$	I	L
N	O	I	S	$	I	L	L	I
O	I	S	$	I	L	L	I	N
S	$	I	L	L	I	N	O	I
$	I	L	L	I	N	O	I	S

Figure 2.2: Sorted Burrows-Wheeler Transform Circular Shifts

2.3.2 Inexact vs. Exact Match

Searching for an exact match of a string within a reference genome represents a much less computationally complex problem than inexact matching. One example of a technique implemented to perform inexact matching is called backtracking [2]. This is used by the Bowtie and Bowtie 2 aligners. If an exact match is not found on the original query sequence, the program backtracks and tries to perform another exact match, except with a substitution/deletion/insertion applied to the original search query. If still nothing is found, the program continues to make further changes to the sequence until some sort of limit is reached. This algorithm can very quickly multiply the number of computations being performed in search of a match. This makes backtracking a computationally expensive process to perform.

2.3.3 Seed-and-Extend

Performing an exact search on a very long search sequence decreases the likelihood of a match being found. On the other hand, using inexact search can quickly become very time-consuming. Helping to minimize computation time and maximize search hits, aligners typically adopt a strategy that is known as seed-and-extend [2].

Utilizing the FM-Index and Burrows Wheeler Transform, one can very quickly perform an exact match on a substring of the read, finding all locations within the reference genome. Afterwards, with the potential locations already determined, the exact matched substring can be extended to match the full-length read and then compared. Examples of alignment programs that use this scheme are Bowtie 2, BWA, and SNAP [3], [4], [5]. The program performs an exact match on short substrings of the original search sequence. This does not require any inexact matching to be done and has a relatively high probability of obtaining a hit compared to the original full length search queries. These matched substrings represent what are known as *seeds*.

The algorithm then takes the seed and its location on the reference genome and extends the sequence around its seed in order to account for those nucleobases that were cut off during this process. The extended sequence will represent a greater length, which corresponds to the length of the read. This process is known as the extension portion of the seed-and-extend method. After extension, there may be mismatches when comparing the search sequence to the extended hit on the reference genome. Note that through this method, it was possible to essentially obtain an inexact match without performing the costly backtracking procedure.

Some programs combine the seed-and-extend techniques with backtracking, allowing the user to configure a small number of mismatches on the seed, which will be handled using the backtracking technique. Using a small number of mismatches (i.e., 1-2) provides a compromise between the complexity of backtracking and results quality.

2.3.4 Scoring

After the seed-and-extend process is completed, the aligner must determine which candidate alignment is the best. In other words, the quality of each candidate must be evaluated. Naturally, this utilizes a score value, which is calculated based on the reported quality of each nucleobase and whether or not it matches the reference genome.

Further increasing the complexity, the quality calculation must also consider possible insertions/deletions, which may shift the alignment by a number of nucleobases. This inclusion of insertions/deletions makes the scoring process no longer a trivial calculation. The most popular method used for finding the highest scoring alignment between two strings is known as the Smith-Waterman algorithm.

Smith-Waterman Algorithm

The Smith-Waterman algorithm is used to perform local alignment [6]. That is, it will take two sequences and figure out the most efficient way that they align to each other. It is considered a dynamic programming algorithm. The algorithm is commonly represented in the form of a matrix. There is also a corresponding scoring scheme which assigns positive or negative point values for the occurrences of a match, mismatch, insertion, or deletion.

Each element in the matrix is defined as follows:

$$H(i,j) = \max \left\{ \begin{matrix} 0 & \\ H(i-1,j-1) + \ s(a_i,b_j) & \text{Match/Mismatch} \\ \max_{k\geq 1}\{H(i-k,j) + \ W_k\} & \text{Deletion} \\ \max_{l\geq 1}\{H(i,j-l) + \ W_l\} & \text{Insertion} \end{matrix} \right\}$$

$$1 \leq i \leq m, 1 \leq j \leq n$$

The first row and column start with the initial values of zero. The first row and first column do not correspond to any part of either sequence. As described in the equation above, the representation starts in the second row and column of the matrix.

Accordingly, the matrix can be generated in $O(mn)$ time, where m and n correspond to the lengths of the read and the substring of the reference genome that is being compared to.

2.4 Field-Programmable Gate Array (FPGA)

A stark contrast to the CPU that is often used for general computing, the FPGA represents a piece of hardware whose circuit can be configured according to a custom design [7]. When using a CPU, the developer only has control over what software is being run on the hardware. The hardware elements of the CPU are already designed and implemented as a permanent solution and cannot be reconfigured. On the other hand, when utilizing FPGA, the developer may have full control over the hardware, configuring a specially designed circuit in order to meet the unique demands of each application. This FPGA design is most often represented and developed using a hardware description language (HDL). The two most prominent HDLs being used today are Verilog and VHDL.

As one can imagine, designing and implementing hardware can be an extremely tedious and time-consuming task. The nature of hardware design presents many challenges for verification. A typical way of testing a hardware design is using a testbench for simulation. The entire design is encapsulated within a testbench module. Then, the tester decides what vectors should be applied to the circuit in order to achieve a certain behavior. One can then use assertions in order to verify that the behavior is as expected.

FPGAs consist of many programmable logic blocks, among which are programmable interconnects. This arrangement allows the hardware to be routed in a massive number of configurations. Programmable logic blocks often contain components such as lookup tables, multiplexers, and flip-flops. This general architecture is shown in Figure 2.3 [8].

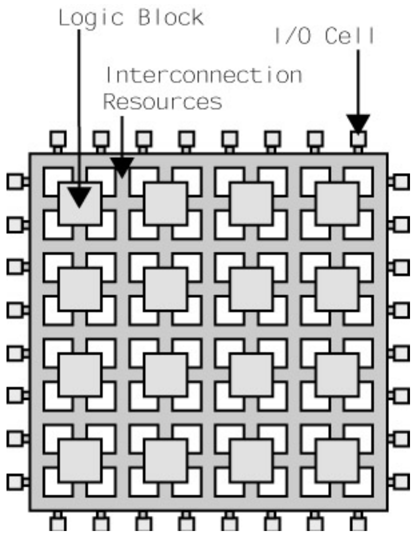

Figure 2.3: FPGA Architecture

After an HDL is written for the desired circuit, it is then synthesized. The process of synthesis converts the relatively high-level HDL into a gate-level representation of the circuit. At this point, simulation can be performed in order to verify the correctness of the circuit. However, it is not yet ready to be programmed to an FPGA.

Following the synthesis process is a step known as place-and-route. This is usually done automatically by utilizing a place-and-route tool. The synthesized HDL outputs a netlist. The place-and-route tool determines how the circuit will be actually configured on a specific FPGA in order to represent the desired circuit. Placing refers to assigning a physical circuit element on the FPGA to represent a portion of the circuit. Routing refers to configuring the interconnects in order to represent the synthesized netlist.

2.5 Heterogeneous Parallel Computing

In order to achieve the maximum performance gain with an FPGA, the heterogeneous parallel computing system methodology is implemented. This relies on utilizing different platforms in order to perform well-suited computations. In this project, the desired usage for this project is to use the CPU for general purpose computations, but then algorithms with high amounts of parallelism can be computed on the target device.

In a typical setup, the *host* system will be running the bulk of the code, including the code that invokes the parallel computing kernel, which is run on the *device*. It is important to consider that data must be deliberately transferred between the host and device, an overhead that is not incurred in typical programs where the entire application is run on what would be the host system. Figure 2.4 shows the memory hierarchy for CUDA applications running on GPU [9]. In order for the GPU to use any data, it must first be transmitted to the global memory on the GPU. Once the data is in global memory, it must be optimized by being moved to faster memory as appropriate for the specific application. This is the only way to achieve maximum speedup. In summary, careful consideration of memory management is essential when accelerating an application using heterogeneous computing.

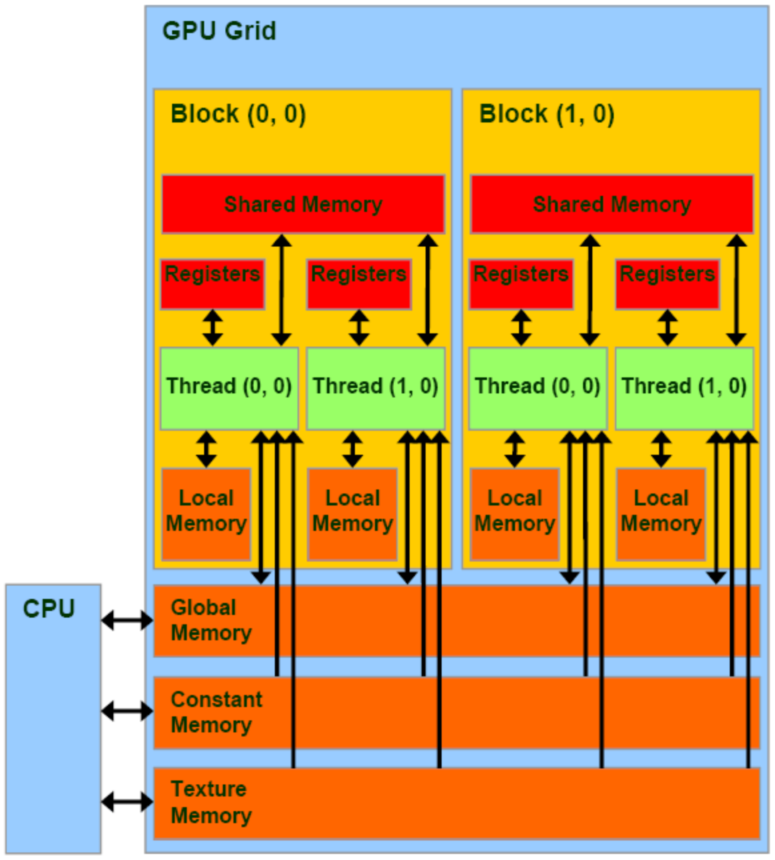

Figure 2.4: CUDA Memory Model

2.6 High-Level Synthesis

HLS is a technology that allows for the conversion of high-level code into a bitstream that may be directly utilized on the FPGA. HLS, if desired, can be performed directly on C code in conjunction with some optimization pragmas.

For this project, HLS is performed on OpenCL, a language that is based on an open standard for parallel computing. In contrast to another popular parallel computing language called CUDA, which only works natively with Nvidia GPUs, OpenCL is compatible with many different platforms. In particular, OpenCL can be run on GPU much like CUDA. However, the same code can also be synthesized to run on FPGA. This synthesis is typically done by using HLS tools provided by Altera or Xilinx.

The use of high-level code in order to describe hardware designs that are typically implemented at the circuit level is an emerging technology. Such solutions are very desirable because they greatly improve the productivity of hardware design, allowing efficient solutions to be deployed much faster than if the technology were being implemented at the hardware description language (HDL) level. The difference in productivity is clear by looking at Figure 2.5 [10].

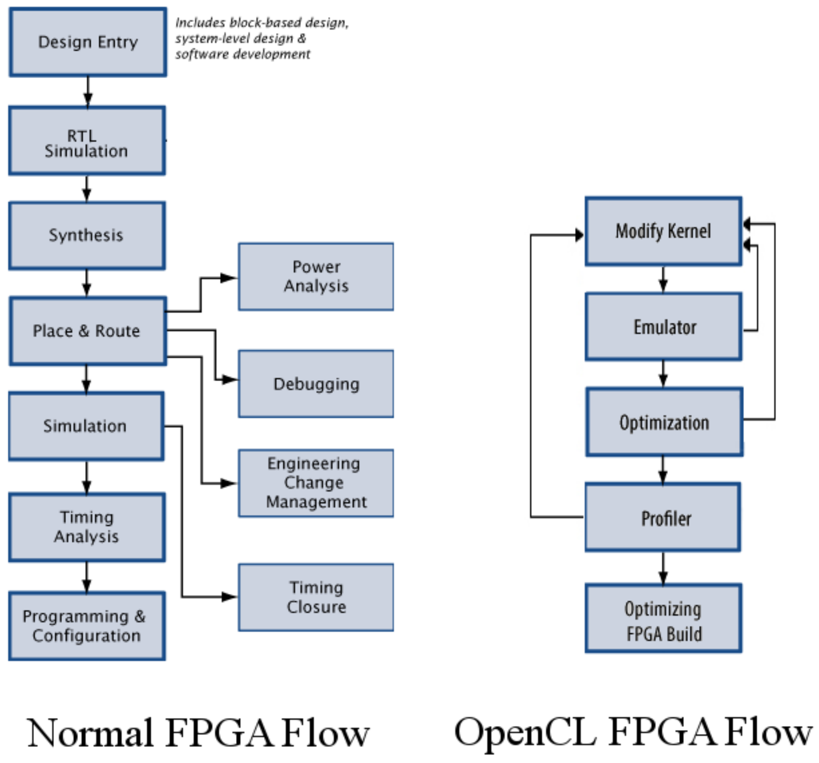

Figure 2.5: Normal vs. HLS Design Flow

By writing code meant for parallel processing in OpenCL, the parallel structure of the code is more directly apparent during synthesis. If synthesizing directly from normal C or C++ code, there is no guarantee that the synthesis tool is able to see the parallel structure that is so apparent to the developer and can be more efficiently described by utilizing OpenCL.

2.6.1 CPU vs. GPU/FPGA

The CPU has its advantage as a low latency device. On the other hand, it has relatively low throughput. It works the best when operating on computations that are necessarily serial—that is, the next step cannot proceed without some result from the first step. For programs that follow this flow, it is advantageous to perform each individual computation as fast possible. Much of the CPU was designed in order to excel at this type of processing. The classic structure of the CPU involves a pipeline that is focused on getting a single instruction through the pipeline as fast as possible.

On the other hand, parallel computing hardware is advantageous in situations that require high throughput: a parallel processor, can perform many, many computations at the same time. However, parallel computing hardware generally has much higher latency than its CPU counterpart. The implication of this is that there will be a huge performance hit if there is not enough parallelism to combat the high latency.

To summarize, the CPU will shine during computations that require low throughput, while parallel computing will excel when there are many elements that can be processed at the same time. For example, if we want to perform a single addition, the CPU has lower latency and will be able to finish this computation faster than the parallel computing hardware. However, if we want to perform 10,000 additions, the CPU does not have the throughput to process all of this at once, whereas the parallel computing hardware could process all 10,000 additions in parallel, resulting in a large performance gain over the CPU despite its higher latency on a single computation.

2.7 Related Works

There are several existing FPGA-based short read aligners. Although some may be high performing, it was sometimes found that capabilities were greatly reduced in order to assist FPGA implementation.

One example of an FPGA-based aligner is Shepard [11]. It is about 60 times faster than GPU implementations. However, it can only perform exact matches. This significantly reduces the computational complexity of the application. When performing exact match, the Smith-Waterman algorithm is not necessary. The Smith-Waterman algorithm represents a significant portion of the runtime in most aligners.

VelociMapper is a commercial FPGA aligner [12]. There are not many details to its implementation, but it claims to be faster than aligners such as BWA and Bowtie 2, while allowing for a high number of mismatches.

An FPGA aligner implemented by Arram et al. [13] implements FM-Index and backtracking on 8 Altera Stratix-V FPGAs. They achieve 28x speedup over Bowtie 2.

CHAPTER 3

PROFILING

The main objective of this project is to obtain speedup on a short read aligner by utilizing the FPGA for processing strategic portions of the program. Instead of manually developing Verilog, high-level synthesis (HLS) is utilized in order to achieve the desired results.

3.1 Selecting an Aligner

The project goal is to obtain speedup on a pre-existing aligner rather than write a new one wholly from scratch. This will help in demonstrating the utility of OpenCL and HLS as a viable and efficient method to accelerate a pre-existing program. However, special care must be taken to select a suitable program for use with heterogeneous programming. Since the processing is no longer happening solely on the CPU, data must be transferred onto an external chip before computation can be performed. With this data transfer, there are new potential constraints and bottlenecks to be considered. The first consideration is the amount of memory available on the FPGA/GPU, which is generally less than that available to the CPU. Therefore, there may be major performance implications if the selected program is particularly memory-intensive. Secondly, there is the possibility of an IO bottleneck when performing data transfers. Even if the algorithm on the FPGA/GPU is extremely fast, its performance enhancements may be covered by the speed of transferring data back and forth. Therefore, it is important to consider how much data transfer a specific program will utilize when working as a heterogeneous application.

3.1.1 Dataset

Reference Genome

When gathering data, we utilize the *Anopheles gambiae* (anoGam1) reference genomes.

anoGam1 is selected because it is unnecessarily time-consuming to run alignments on the relatively long human genome when the goal is to verify the correctness of an implementation. Correctness is not dependent on the size or length of the dataset. A constant reference genome is also useful for determining speedup due to changes in program implementation.

Reads

There are two classes of reads that we can utilize. First, there are the reads that are from an actual organism and sequenced using technologies such as Illumina HiSeq [14]. This type of dataset has several advantages:

1. Realistic nucleotide sequences

2. Quality data for each nucleobase

Second, using a real sequenced dataset provides some challenges that may hinder the ability to collect results. For example, it may be difficult to find a dataset with exactly the desired read length. More importantly, there is generally no golden model for the alignment results of these datasets. Therefore, it would be very difficult to establish the accuracy of an alignment. If the accuracy of an alignment cannot be established, then its speed may not be meaningful. It is primarily for this reason that the second class of reads, called simulated reads, is used in experiments analyzing the performance. For this experiment, the `wgsim` application is used in order to simulate the desired reads. There are several distinct advantages of using simulated reads:

1. Ability to evaluate the quality of alignment

2. Full control over read length

3. Ability to specify error rate of read sequences

However, the most overwhelming disadvantage of read simulators is that `wgsim` does not simulate the quality of the each nucleobase. Each nucleotide in the sequence is hard-coded with the same quality score.

3.1.2 Profiling Speed and Accuracy

It is desired to select a pre-existing aligner that was already high performing. This will make the acceleration efforts more meaningful by pushing the cutting edge of alignment performance rather than accelerating an average application. Some surveying was performed in order to select a variety of aligners that are popular. The selected aligners are listed in Table 3.1.

Table 3.1: Selected Aligners for Profiling

Aligner	**Version**
Bowtie 2	bowtie2-2.2.5
BWA	bwa-0.7.12
SNAP	snap-0.15.4
SOAP2	soap2.20

In order to evaluate these aligners, a dataset was generated using `wgsim`. The command used to generate these reads was:

```
./wgsim -N100000 -1200 -d0 -S11 -e0 -r0.02 \
../../ref/anoGam1.fa ../anoGam-200-02.fq /dev/null
```

This generated dataset has the specifications listed in Table 3.2

Table 3.2: Performance Measuring Dataset

Genome	**Read length (bp)**	**Number of reads**	**Error rate**
anoGam1	200	100,000	0.02

3.1.3 Profiling Memory Usage

In addition to raw speed, it was also important to consider the memory usage of these programs. When performing heterogeneous parallel programming, data needs to be moved back and forth from the target device. For both

GPU and FPGA, this memory transfer is done over the PCI-Express bus. This issue is not as significant when we are dealing with processing that is occurring purely on CPU. As such, it is important to measure the peak memory usage of each aligner.

The memory transfer between host and device is an extremely important metric of suitability for heterogeneous programming because alignment is a relatively IO intensive process. This means that even with the most efficient parallel algorithms, the performance may be dictated by the memory transfer speeds to the GPU or FPGA.

The peak memory usage of the program was measured using:

```
/usr/bin/time -v
```

This built-in Linux utility is generally used to measure the runtime of a certain process, but also contains an option which allows the user to measure the peak memory usage of the process during that specific run. This benchmark is performed using the CPU-only system shown in Table 5.1

3.1.4 Profiling Results and Discussion

After collecting the results shown in Table 3.3, a holistic comparison was made in order to determine the highest performing aligner.

Table 3.3: Performance Results (anoGam1, 200bp, 0.02 Error Rate)

Aligner	**Time (s)**	**Aligned (%)**	**Peak Memory (MB)**
Bowtie 2	56.2	99.99	327
BWA	109.3	93.39	586
SNAP	118.9	99.98	4681
SOAP2	23.5	82.32	1087

Bowtie 2 has the lowest memory usage by far. Without considering any other performance metrics, the SOAP2 and SNAP aligners are largely out of contention due to their high memory usage, especially SNAP. Bowtie 2 also has the highest alignment rate, closely followed by SNAP. However, as discussed before, SNAP's memory usage is simply far too high for a GPU/FPGA application. SOAP2 is faster than Bowtie 2, but the significance of this speed is limited considering the relatively low alignment rate. Based on these results, Bowtie 2 prevails as the overall highest performing aligner.

3.2 Accelerating Bowtie 2

3.2.1 Identifying Target Functions

Heterogeneous parallel programming does not seek to port the entire application onto FPGA. It is common to encounter certain “problem” functions within a program that compose a significant portion of the runtime. Therefore, a minority of the code may be responsible for majority of the runtime. If this minority of code is suitable for parallelization, then it would be extremely appropriate for FPGA acceleration. Therefore, the goal of this profiling is to find those “problem” functions for FPGA acceleration.

Based on previous knowledge of how short read aligners work, it is expected that the algorithm will have extremely high levels of parallelization on multiple levels. In order to verify these expectations and assist with identifying key functions, profiling is performed to obtain a call graph for the application. This call graph shows not only the flow of the application, but also the number of calls and percentage of time spent in each function.

This profiling is performed using a program known as `gprof`, a GNU utility. This application provides the user with a text-based call graph for a specific run of an executable. Using an application called `gprof2dot`, it can be converted into a graphical call graph which is seen in Figure 3.1. This graphical view of the call graph simplifies the matter of understanding the program’s flow.

Call Graph Analysis

The complete call graph for Bowtie 2 consists mostly of minor functions which represent insignificant amounts of runtime. There are only a few key functions which represent over 90% of the runtime. This specific region in the program flow is shown in Figure 3.1.

The call graph allows us to see which functions correspond to which portions of the Bowtie 2 algorithm. This in-depth algorithm is shown in Figure 3.2 [3].

A higher-level pseudocode of this algorithm is shown in Figure 3.3

In the call graph, the most ideal function for acceleration is contained within `SwDriver:extendSeeds` which in turn contains `SwAligner::align`.

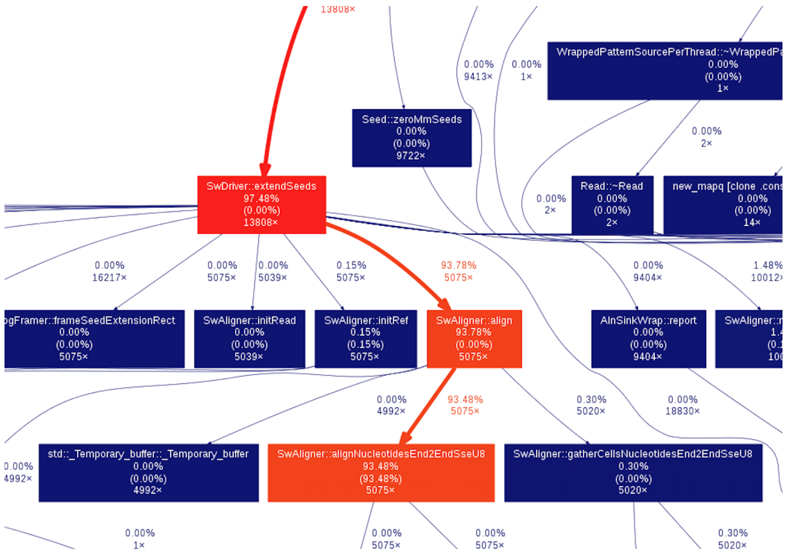
SwDriver::extendSeeds 97.48% (0.00%) 13808×
SwAligner::align 93.78% (0.00%) 5075×
SwAligner::alignNucleotidesEnd2EndSseU8 93.48% (93.48%) 5075×
SwAligner::gatherCellsNucleotidesEnd2EndSseU8 0.30% (0.00%) 5020×
std::_Temporary_buffer::_Temporary_buffer 0.00% (0.00%) 4992×
Seed::zeroMmSeeds 0.00% (0.00%) 9722×
SwAligner::initRead 0.00% (0.00%) 5039×
SwAligner::initRef 0.15% (0.15%) 5075×
AlnSinkWrap::report 0.00% (0.00%) 9404×
Read::~Read 0.00% (0.00%) 2×

Figure 3.1: Key Region of Bowtie 2 Call Graph

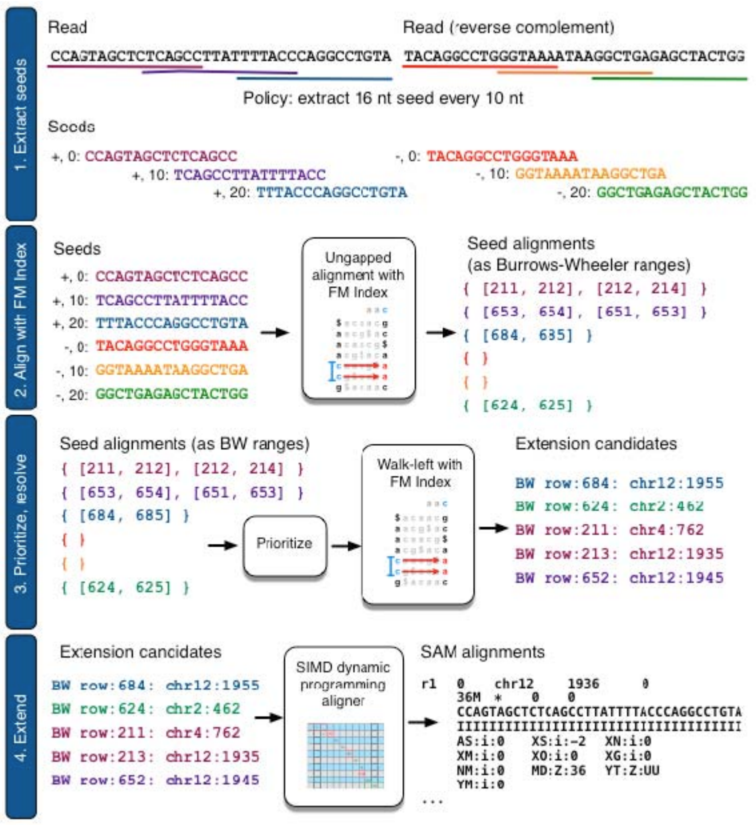
1. Extract seeds
Read
Read (reverse complement)
Policy: extract 16 nt seed every 10 nt
Seeds
2. Align with FM Index
Ungapped alignment with FM Index
Seed alignments (as Burrows-Wheeler ranges)
3. Prioritize, resolve
Prioritize
Walk-left with FM Index
Extension candidates
4. Extend
SIMD dynamic programming aligner
SAM alignments

Figure 3.2: Bowtie 2 Program Flow

The target function is called `SwAligner:alignNucleotidesEnd2EndSseU8`. This function represents the calculation of the Smith-Waterman matrix. This

```
for read in reads:
        seeds = generate_seeds(read)
        for seed in seeds:
                if(exactMatch(seed) > 0):
                        hits.add(exactMatch(seed))
        for hit in hits:
                scores.add(smithWaterman(hit))
        best_hit = max(scores)
        alignment = backtrace(best_hit)
```

Figure 3.3: Pseudocode of Bowtie 2 Program Flow

is also referred to as `SIMD dynamic programming aligner` under step 4 in Figure 2.5. The Smith-Waterman scoring calculation represents about 93-94% of the runtime. This function is selected as the target function for parallel acceleration because it represents the function which occupies the largest amount of runtime without including any specific function within it that represents the significant majority of that runtime.

3.2.2 Striped Smith-Waterman

Bowtie 2 utilizes an accelerated version known as the Striped Smith-Waterman algorithm [15]. This algorithm utilizes SIMD instructions that operate on 128 bit registers in order to introduce instruction level parallelism to Smith-Waterman. The 128 bit SIMD registers generally correspond to 16 elements of 8 bits each. In some cases, higher precision is required and instead the 128 bits are divided into 8 elements of 16 bits each. The SIMD is processed by utilizing Intel's SSE2 instruction set. Farrar's algorithm also generates a query profile, used in determining each cell's score in a striped memory access fashion, creating greater efficiency. This implementation is able to achieve 2-8x speedup over pre-existing SIMD implementations [15].

CHAPTER 4

KERNEL DESIGN

The plan is to accelerate the application by adapting the Striped Smith-Waterman functionality for OpenCL execution. From there, various levels of parallelism can be attained, resulting in an overall acceleration of the application. The different types of parallelism that are achieved will be described in detail in this chapter. When the Smith-Waterman kernel is written using OpenCL, it is possible to both run it on GPU and synthesize for FPGA with the same code. It is expected that running the kernel on GPU will be more straightforward and have a shorter development time than needed to get it running on FPGA. This is because setting up the data transfers is relatively simple and well-documented for GPUs, while one usually encounters some additional levels of complexity when performing data transfers onto FPGA. Since the same code can be used for both, it is a natural stepping stone to verify functionality and fine-tune the code by running the kernel on GPU, first.

4.1 Striped Smith-Waterman Adaptations

The Smith-Waterman function already incorporates instruction-level parallelism, which is replicated in the OpenCL kernel. However, OpenCL does not have a 128-bit datatype like the SSE2 instruction set does. The solution was to basically re-write all of the used SSE2 instructions in vectorized form. Instead of a primitive datatype, an equivalent representation is implemented in the form of an array of 8 bit chars or 16 bit shorts, depending on the granularity needed by the SIMD instruction. The OpenCL compiler will automatically vectorize code that is written in an appropriate format. In this way, high performance, utilizing instruction level parallelism, was achieved on the code.

4.2 Parallelism Overview

The application of read alignment has many potential levels of parallelism. Each read is aligned completely independently of all other reads. Within each read, the candidate seeds can be processed independently of the other seeds within a read. Already, there are two levels of parallelism. For each seed, the Smith-Waterman matrix must be generated. Within an individual Smith-Waterman computation, there exists some opportunity for data parallelism. This data parallelism within the kernel is largely based on the Striped Smith-Waterman optimization by Farrar [15].

4.3 Read-level Parallelism

In the unmodified Bowtie 2 code, each read is put through the entire alignment pipeline before moving onto the next read. While it makes sense to parallelize the entire pipeline, this represents a large amount of complexity for a device like the GPU and FPGA. Additionally, a majority of the runtime is occupied by a relatively minor portion of the pipeline, the Smith-Waterman scoring algorithm. As such, the aim is to parallelize the minimum portion of the codebase in order to allow for this level of parallelization. Performing any additional parallelization will result in much greater memory usage, while not increasing runtime much at all. In fact, due to constraints on memory transfer time and scarcity of device memory available, parallelizing these non-intensive portions of the pipeline may result in overall slowdown.

Due to the serial nature of the pre-existing code, restructuring of the program flow is required in order to obtain the desired parallelism. The comparison of the very high-level pseudocode in Figures 4.1 and 4.2 illustrate the change in program flow.

```
for read in reads:
        generate_seeds
        match_and_extend_seeds
        score_hits
        return_best_alignment
```

Figure 4.1: Bowtie 2 Unmodified Alignment Pipeline

With this type of program flow, it is impossible to get read-level parallelism unless the entire pipeline is parallelized. As discussed before, this is not optimal. Figure 4.2 shows the optimal flow for maximum parallelism.

```
for read in reads:
        generate_seeds
for read in reads:
        match_and_extend_seeds
for read in reads:
        score_hits
for read in reads:
        return_best_alignment
```

Figure 4.2: Bowtie 2 Restructured Alignment Pipeline

This program flow restructuring is achieved by using the `pthreads` library, which allows the application to run multiple threads at a time. For each of n threads, one read will be processed. Once all threads reach the desired *score_hits* stage, the data from the reads of all the different threads are batched together and processed in parallel on the target device. In this way we can achieve n-way parallelism.

When using `pthreads` to achieve this parallelism, it is not necessary to explicitly parallelize all steps in the alignment process. A *barrier* is used right before the *score_hits* portion is reached so that all threads will sync up at that place, allowing the parallel kernel to execute. However, the concurrence and order of execution of the alignment steps before and after this parallel kernel execution are not of particular importance.

4.4 Seed-level Parallelism

Similar to the read-level parallelism, the alignments of all the seed candidates within a read are totally independent of each other and can be processed in parallel. Much like the issue with the program flow for the reads, Bowtie 2 processes the seeds in a non-parallel friendly manner. That is, each seed is fully processed in serial before moving onto the next seed. However, the sole desired function of the parallel kernel is to perform the Smith-Waterman scoring. At the seed-level, it is feasible to directly modify the code in order to achieve the desired program flow. As such, there is no need to rely on the `pthreads` method in order to restructure the application.

4.5 High-Level Synthesis Optimizations

In order to fully exploit the parallel nature of the code, pragmas are used in the OpenCL code in order to explicitly denote where parallelism exists. For this kernel, loop unrolling and pipelining will in theory provide some benefit, allowing certain parts within each parallel thread to execute concurrently.

4.5.1 Vectorization

The pre-existing code was already vectorized to exploit high levels of parallelism. Instead of calculation being performed on one element, it was being performed on eight elements at a time. As an artifact of this manual parallelization, loop iterations were no longer independent of each other. There is only one main loop in the entire kernel. As a result, using the loop unrolling pragma presented the identical circuit as the non-unrolled version.

```
uchar16 _uc_cmpeq_epi16(uchar16 a, uchar16 b) {
    short8 * a_short8 = &a;
    short8 * b_short8 = &b;
    short8 four_f = (short8)0xffff;
    short8 four_0 = (short8)0x0000;
    (*a_short8) = (*a_short8) == (*b_short8) ?
                four_f : four_0;

    return a;
}
```

Figure 4.3: Vectorized Comparison Function

Figure 4.3 shows one example of the vectorized function that is built into the kernel, exploiting parallelism at a low level. This function performs the *equals* operator across 8 elements at once.

Figure 4.4 shows an application of the unrolling pragma. If the function were simple enough and did not have dependencies within loop iterations,

```
// For each character in the reference text:
size_t j;
#pragma unroll
for(j = 0; j < iter; j++) {
    // Load cells from E, calculated previously
    ve = _uc_load_si128(pvELoad);

    pvELoad += ROWSTRIDE;

    // Store cells in F, calculated previously
    // veto some ref gap extensions
    vf = _uc_subs_epu8(vf, pvScore[1]);
...
...
...
    // Save E values
    _uc_store_si128(pvEStore, ve);
    pvEStore += ROWSTRIDE;

    // Update vf value
    vtmp = _uc_subs_epu8(vtmp, rfgapo);
    vf = _uc_subs_epu8(vf, rfgape);

    vf = _uc_max_epu8(vf, vtmp);

    pvScore += 2;
}
```

Figure 4.4: Unrolling Pragma on the Kernel

extra hardware would be utilized allowing sequential iterations to run concurrently, instead.

4.5.2 Pipelining

Pipelining allows different stages of the program execution to overlap each other. It was found that this had a tangible benefit. When high amounts of data were being processed, the kernel execution time would decrease by a factor of up to 5x. The pragma for pipelining is applied on a kernel basis, as shown in Figure 4.5

```

__kernel void my_alignNucleotidesEnd2EndSseU8(
    __global aln_data * restrict input,
    __global uchar16 * restrict profbuf_,
    myTAlScore                 minsc_,
    int
readGapOpen_val,
    int
refGapOpen_val,
    int
readGapExtend_val,
    int
refGapExtend_val                ) {

#pragma HLS PIPELINE
...
...
...
}
```

Figure 4.5: Pipelining Pragma on the Kernel

CHAPTER 5

RESULTS

The same system was used for both CPU and GPU benchmarks. It is important that the same CPU is used in both benchmarks in order to hold all factors constant except for the introduction of the GPU kernel. The detailed configuration is shown in Table 5.1.

Table 5.1: System Specifications

Processor	Intel Xeon E5-2603
RAM	32 GB
Graphics Card	Nvidia GTX 770
VRAM	4096 MB

5.1 GPU Performance

5.1.1 Kernel Memory Performance

Initially, the kernel was IO bound, attributing the majority of its runtime to memory transfer. The majority of the data being transferred back and forth between the host and device belonged to the Smith-Waterman matrix. It turns out that the content of this matrix is used solely for the backtracing procedure. Since the matrix is being generated within the kernel already, it makes sense to incorporate the backtracing procedure as part of the kernel as well. This way, the computation can be directly performed on the data instead of having to transfer it back to CPU.

With the backtrace operation being performed on the kernel, the significant majority of data being transferred from device to host was eliminated. This resulted in a massive amount of speedup for the overall kernel. The results shown in Figure 5.1 illustrate the vast difference in the kernel's performance

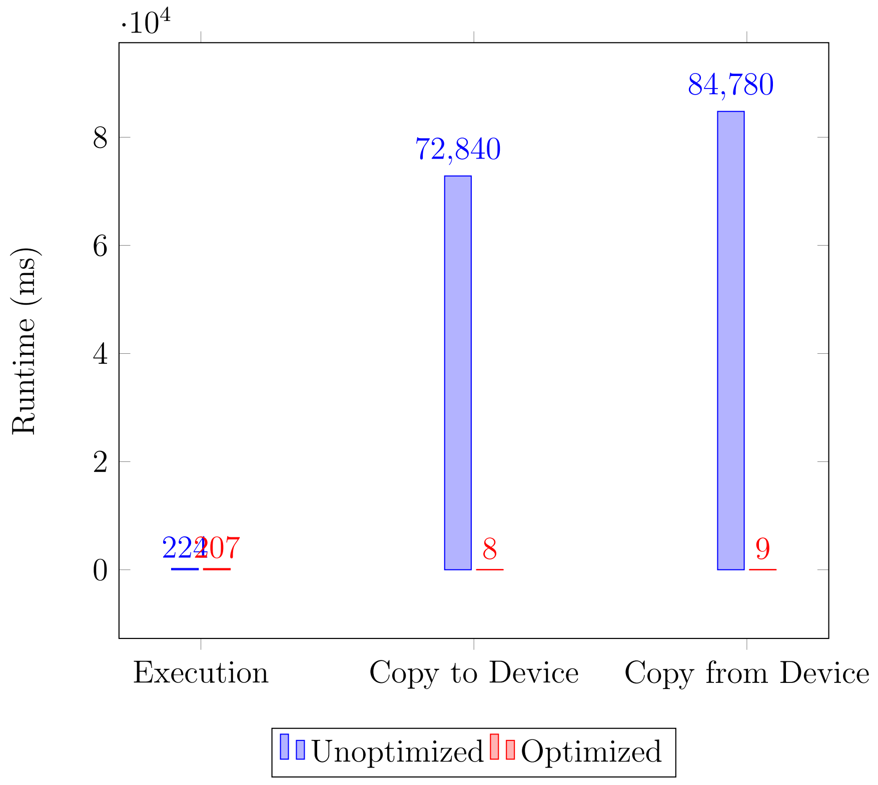

Figure 5.1: GPU Kernel Memory Performance

with and without this optimization. It is also apparent that the performance of this kernel is IO bound rather than compute bound.

5.1.2 Kernel Execution Performance

The results in Figure 5.2 show the difference in the Smith-Waterman performance of the Bowtie 2 CPU vs. GPU version. The total kernel runtime is measured as the time including memory transfer from the host to the kernel, executing the kernel, and memory transfer from the kernel to the host. The CPU runtime measures the total time that the Smith-Waterman computation is being executed. This is performed on the `anoGam1` genome with varying read lengths. With the `pthreads` optimization, the application is able to achieve a level of parallelism equivalent to the number of threads that are executed. For this experiment, 512 threads are used. Ideally, the more threads, the better. However, there are some diminishing returns when executing this many different threads on CPU due to the need to switch contexts when changing between threads. Eventually, the amount of time lost

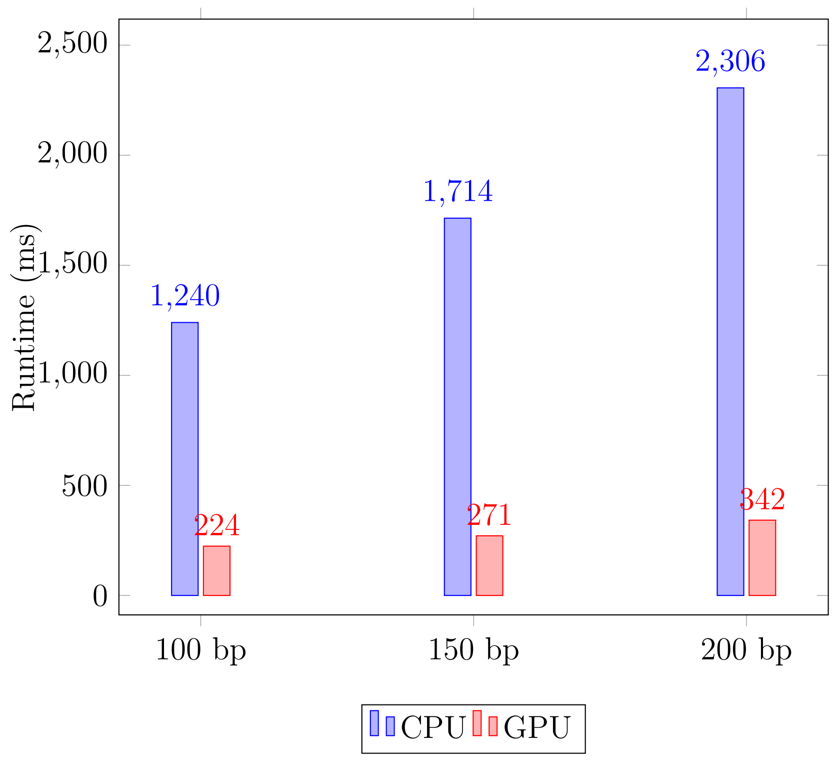

Figure 5.2: GPU Smith-Waterman Kernel Runtime

to processes such as cache evictions outweighs the benefits of more parallelism. It was discovered that 512 threads represent a good tradeoff between parallelism and performance hit from too many different threads.

The amount of speedup can greatly vary depending on the dataset and configuration of the application. While a longer read represents more data to process in serial on the CPU, resulting in a somewhat linear increase in runtime, the effect is not as apparent on GPU. This is because longer reads result in many more seeds, but there is ample parallelism available on GPU for this, mitigating the performance hit.

5.1.3 Application Runtime

The Smith-Waterman memory transfer and execution performance were strong enough that speedup could be expected in the overall application runtime as well. The program source code had to be modified in order to induce parallelism at the seed-level, resulting in some slowdown. Some accuracy had to be sacrificed in order to achieve a high performance on the kernel. This

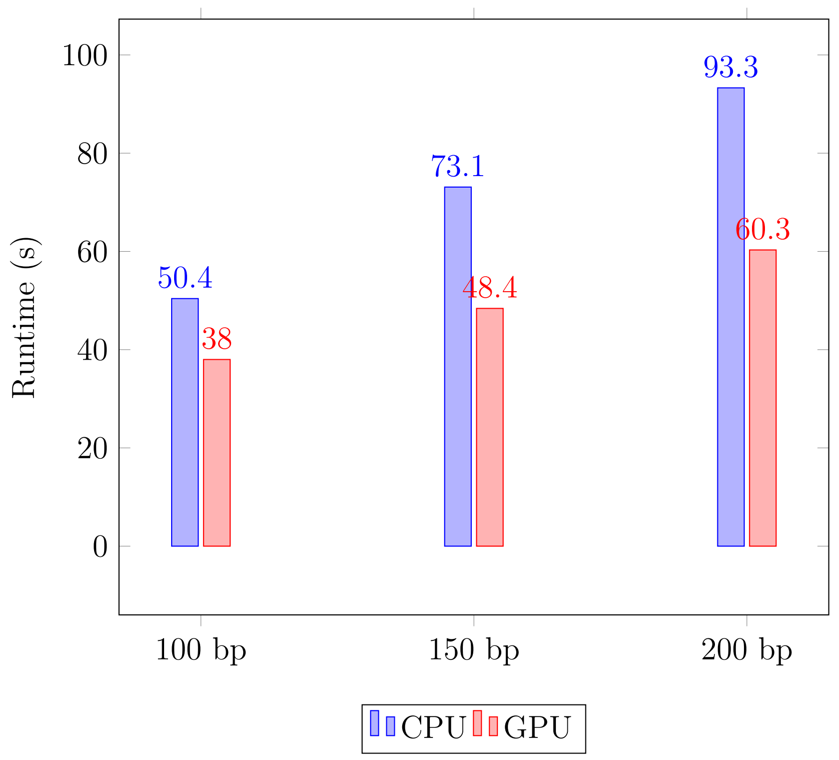

Figure 5.3: GPU Bowtie 2 Runtime

manifested as an alignment rate that usually differed by less than 10% from the unmodified CPU version. The genome used was `anoGam1` with 10000 reads generated using `wgsim`. Various lengths for reads and seed length were used.

Consistent speedup is shown across various read lengths from `anoGam1`, as seen in Figure 5.3. However, there is less of a speedup when the read sizes are smaller, but this is not surprising since more seed-level parallelism exists when the reads are larger.

5.2 High-Level Synthesis for FPGA

After validating the performance of the OpenCL code on GPU, the next step is to synthesize the same code for FPGA. In this project, the Altera Arria 10 GX FPGA is utilized.

5.2.1 Issues

Redundant Initialization on Subsequent Kernel Runs

Getting the kernel to work on FPGA was trickier than the GPU implementation. A few errors resulted in major performance hits, but were later rectified. Firstly, the kernel initialization can persist between multiple uses of the kernel. On average, this initialization took between 1 and 2 seconds and represented an insurmountable overhead when tacked onto each kernel execution. However, after using a persistent kernel object for multiple runs, this dropped to a single instantiation time with near zero initialization for subsequent runs.

Unaligned Host Memory Objects and Direct Memory Access (DMA)

When using FPGA, one can use DMA to move data back and forth between the host and device, bypassing the CPU, resulting in higher performance than traditional methods. Originally, this was not being performed. It was found that this required the host vectors to be 64-byte aligned. This was achieved by padding the structs.

Thread-Safety on Altera OpenCL Host Functions

It was found that the Altera OpenCL library did not have fully thread-safe functions. Attempts were made to resolve this, but it was found that it resulted in unstable behavior. This meant that read-level parallelism could not be exploited on the FPGA version.

5.2.2 Resource Utilization

Table 5.2 shows the resources available on this FPGA.

There were issues with synthesizing the complete kernel. However, it was found that removing the backtracing process from the OpenCL code allowed the synthesis to complete successfully. Altera's high-level synthesis tool, `aoc`, provides a report on the resource usage of the OpenCL kernel. Table 5.3 shows the resource usage of the synthesized Smith-Waterman FPGA kernel.

Table 5.2: FPGA Specifications

Adaptive Logic Modules	427,200
Logic Elements	1,150,000
Registers	1,708,800
IO Pins	992
DSP Blocks	1,518
Memory (Kb)	55,562
RAM Blocks	2,713

Table 5.3: FPGA Resource Usage

Adaptive Lookup Tables	106,891
Registers	191,556
Logic Utilization	95,016
IO Pins	161
DSP Blocks	27
Memory (Kb)	12,087
RAM Blocks	1,007
Actual Clock Frequency (MHz)	150.6
Kernel $f_{\max}$ (MHz)	150.6
Highest non-global fanout	11,068

5.2.3 Kernel Memory Performance

Similar to the GPU analysis, results are collected on the kernel execution itself. The kernel performance data is shown in Figure 5.4. Note that the backtracing process was previously moved onto the GPU in order to reduce memory transfer time.

Due to the increased computational capabilities of the FPGA and the omission of the backtracing process, both the kernel execution time and overall runtime show significant speedup over the GPU. The reduced resource usage from omitting the backtracing process also allows for further optimization using other techniques.

5.2.4 Kernel Execution Performance

The kernel execution runtime is measured using the same process as for GPU. The results are shown in Figure 5.5.

These results show a predictable but significant speedup already suggested

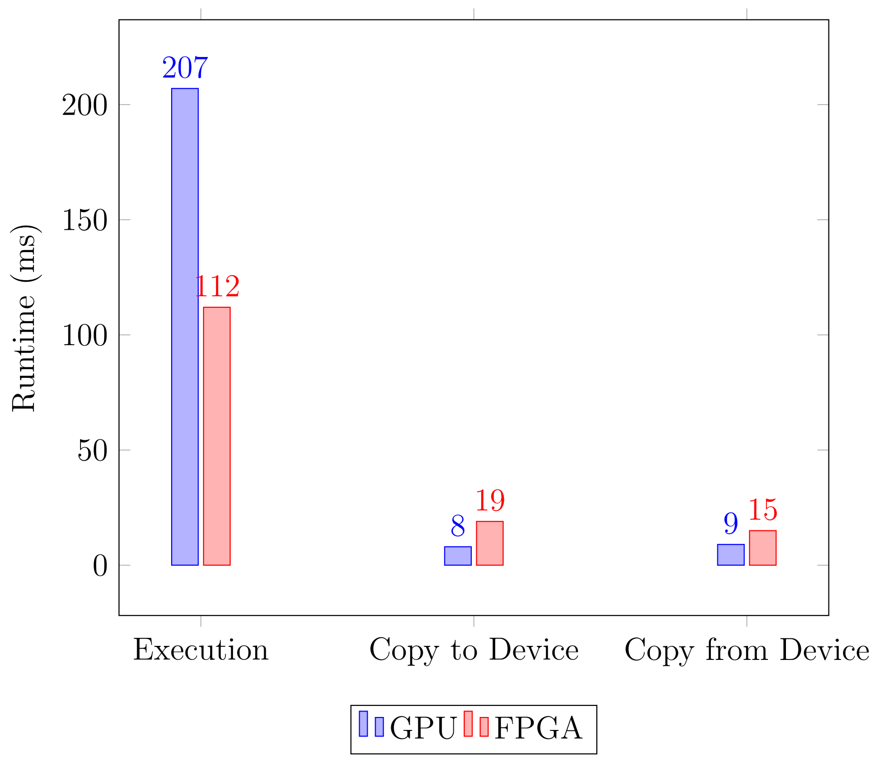

Figure 5.4: FPGA Kernel Memory Performance

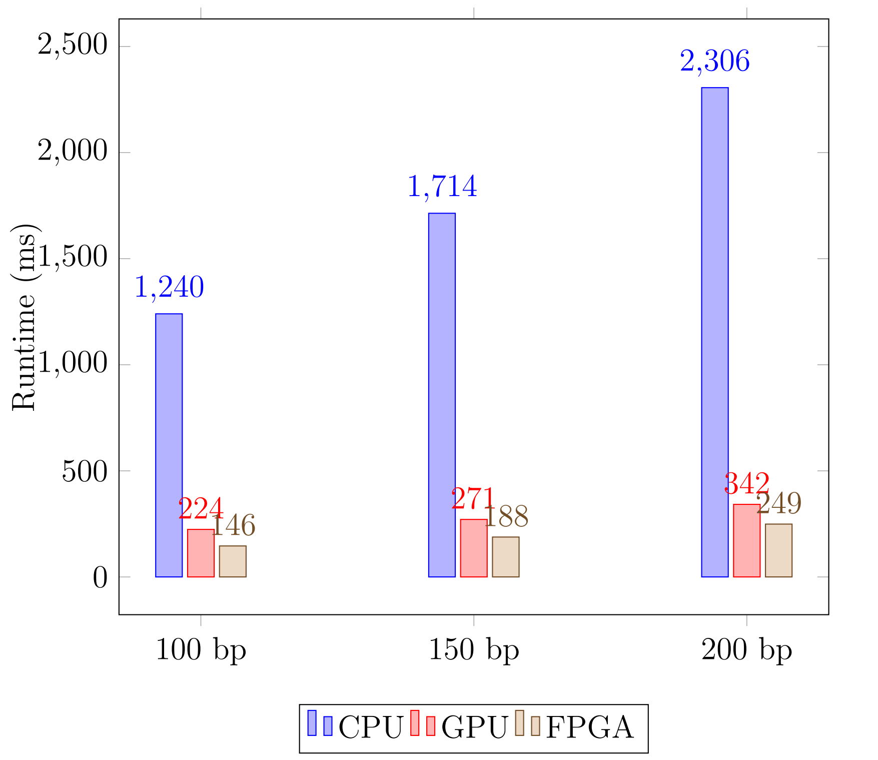

Figure 5.5: FPGA Smith-Waterman Kernel Runtime

by the kernel memory performance.

5.2.5 Application Runtime

The application runtime requires a higher number of reads to mitigate the effects of the overhead. This is especially important for GPU and FPGA where the kernel initialization time may be non-trivial for very small datasets. As such, a size of 10,000 reads with a read length of 100 bp on the `anoGam1` dataset was used. 512 threads were used on the CPU and GPU version, which included an implementation of `pthreads` in order to induce read-level parallelism. This did not work for the FPGA version since the AOCL host OpenCL functions were not fully thread-safe. Attempts to use multiple threads achieved little success and caused the system to hang. Figure 5.6 shows the FPGA runtime results. Although it is slower than the CPU and GPU results, recall that it is not exploiting read level parallelism in the way that the CPU and GPU versions are. Theoretically, one would expect a great performance increase as the number of threads increased. The FPGA kernel computation time would remain relatively constant with only the memory transfer speeds scaling with the number of threads.

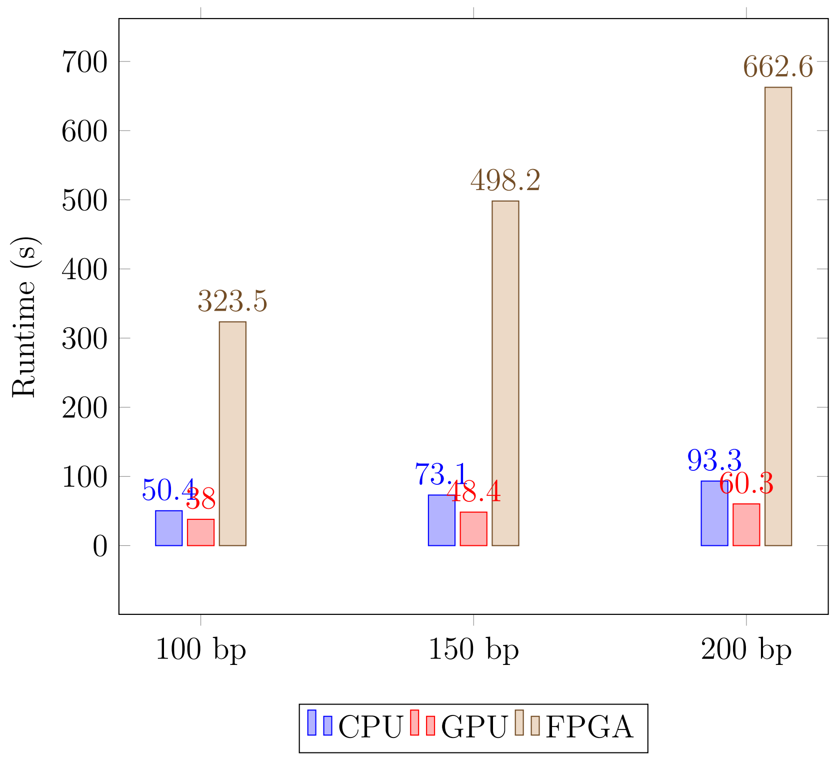

Figure 5.6: FPGA Bowtie 2 Runtime

CHAPTER 6

CONCLUSION

6.1 Discussion

Through a survey and benchmark of the most popular short read aligners, it was determined that Bowtie 2 had the highest performance. Afterwards, algorithmic analysis and profiling were performed on Bowtie 2 in order to identify suitable parallelization opportunities. There were multiple levels of potential parallelization available, all directly related to the Striped Smith-Waterman algorithm that is used to score candidate alignments. The program flow was restructured by using `pthreads` and direct code modification. Vectorization was used within the kernel to achieve higher performance. In the end, read-level, seed-level and instruction-level parallelism were implemented successfully. An efficient OpenCL kernel was developed that showed speedup both by itself and when integrated into the Bowtie 2 application. The program performed well for GPU. The next step was to synthesize this kernel for FPGA.

It was found that the memory transfer times on FPGA were fast enough that the backtracing process could be performed on CPU, freeing up resources for other purposes. As such, the FPGA version of the kernel does not perform backtracing. The synthesis process for this kernel was successful, producing resource utilization rates that were within the capabilities of the Arria 10 GX FPGA. The kernel execution time of the FPGA kernel showed significant speedup over both the CPU and GPU versions. The FPGA kernel execution time showed about 2x speedup over the GPU version.

The FPGA has some tradeoffs that affect performance in various ways compared to the GPU version. Even so, it must be noted that the inputs and outputs are identical even with these differences. The GPU version is able to perform backtracing on the device, reducing the amount of memory that needs to be moved back and forth between the host and device. This proved to be too complex for the high-level synthesis engine. As a result, the FPGA version suffers a hit in the memory performance. Overall, the FPGA kernel performance is higher. However, a big hit in the FPGA performance comes from the inability to use `pthreads` due to the lack of fully thread-safe host functions. This is seen as the primary limiting factor of the FPGA performance.

Although there are existing aligners implemented on GPU and FPGA, this project makes two novel contributions. First, it is a direct port of Bowtie 2, allowing for the use of the same input and output formats while using the same algorithms. Second, OpenCL is used for portability in an extremely large project, allowing for the fast and efficient deployment on both GPU and FPGA with the same codebase.

6.2 Future Work

This project showed great potential with positive results for the kernel speed and overall program acceleration on GPU. Although the FPGA kernel showed promising standalone performance, the memory bandwidth and thread-safety issues prevented the application from achieving max performance. The next step is to explore ways to work around the thread-safety issues, allowing the Smith-Waterman kernel to be efficiently used on the FPGA with multiple host threads.

Even before further optimizing the kernel for FPGA, there are improvements that can be implemented to increase speedup on GPU. It is expected that these optimizations would also increase FPGA performance. First, the utility of `pthreads` was limited due to the overhead presented by the constant context switching on each CPU thread. It is likely that even more speedup can be achieved with direct modification of the program's control flow, allowing for more efficient production of data for the device to consume. The load balancing can also be improved. There are cases where not all threads are being used on the GPU. In those scenarios, speedup could be obtained by distributing the work between more threads. In the current implementation, each thread is responsible for the computation of an entire Smith-Waterman matrix. This could be improved by using multiple threads to compute a single Smith-Waterman matrix when appropriate.

REFERENCES

[1] B. Langmead, "Introduction to the Burrows-Wheeler transform and FM index," 2013.

[2] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome Biology*, vol. 10, 2009.

[3] B. Langmead and S. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature Methods*, vol. 9, pp. 357–359, 2012.

[4] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows-wheeler transform," *Bioinformatics*, vol. 25, pp. 1754–60, 2009.

[5] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. A. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler, "Faster and more accurate sequence alignment with SNAP," *CoRR*, vol. abs/1111.5572, 2011. [Online]. Available: http://arxiv.org/abs/1111.5572

[6] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Molecular Biology*, vol. 147, pp. 195–197, 1981.

[7] K. Benkrid, A. Akoglu, C. Ling, Y. Song, Y. Liu, and X. Tian, "High performance biological pairwise sequence alignment: FPGA versus GPU versus Cell BE versus GPP," *International Journal of Reconfigurable Computing*, vol. 2012, pp. 1966–1967, Feb. 2012.

[8] B. Zeidman, *Chapter 3: Field Programmable Gate Arrays (FPGAs) — Engineering360*, 2016. [Online]. Available: http://www.globalspec.com/reference/32498/203279/chapter-3-field-programmable-gate-arrays-fpgas

[9] J. Oosten, "Cuda memory model," 2016. [Online]. Available: http://www.3dgep.com/cuda-memory-model/

[10] "Faster development cycle for FPGAs" url=https://streamcomputing.eu/campaign/faster-development-cycles-for-fpgas/, urldate=2016-11-22, journal=StreamComputing, year=2016."

[11] C. Nelson, K. Townsend, B. S. Rao, P. H. Jones, and J. Zambreno, “Shepard: A fast exact match short read aligner.” in *MEMOCODE.* IEEE, 2012. [Online]. Available: http://dblp.uni-trier.de/db/conf/memocode/memocode2012.html#NelsonTRJZ12 pp. 91–94.

[12] “Velocimapper.” [Online]. Available: http://www.timelogic.com/catalog/799/velocimapper

[13] J. Arram, T. Kaplan, W. Luk, and P. Jiang, “Leveraging FPGAs for accelerating short read alignment,” *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. PP, no. 99, 2016.

[14] “Sequencing systems,” 2014. [Online]. Available: http://www.illumina.com/systems/sequencing.ilmn

[15] M. Farrar, “Striped Smith-Waterman speeds database searches six times over other simd implementations,” *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/23/2/156.abstract