GPU Acceleration of BWA-MEM DNA Sequence Alignment

by

Stefanos Florescu

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday October 3, 2022 at 11:00 AM.

Student number:	4744918	
Project duration:	September 1, 2021 – October 3, 2022	
Thesis committee:	Dr. ir. Zaid Al-Ars,	CE, TU Delft, supervisor
	Dr. Matthias Möller,	NA, TU Delft
	Dr. ir. Nauman Ahmed,	Netherlands eScience Center

TU Delft

Abstract

Over the past 20 years, the cost of sequencing genomes has reduced drastically. As DNA data grows at an unprecedented rate, the need for fast and affordable software and hardware solutions for DNA analysis is higher than ever. A critical and time-consuming component in any DNA analysis pipeline is sequence alignment, which refers to mapping smaller sequences onto larger reference regions exhibiting high similarity. Many modern aligners break the problem of alignment into seed and extension. First, the algorithm finds matching substrings between the references and the query and then extends these strings on both ends to contain areas around the seeds. One of the most popular and utilized alignment tools is the Burrows-Wheeler aligner, specifically its Maximal Exact Match version called *BWA-MEM*. This tool solves the alignment problem by first finding seeds using a unique index called the FMD index, which reduces the complexity of the string matching problem while the extension is performed using a modified Smith-Waterman algorithm. Since the seed and extension methods are time-consuming tasks, they have been the target of various optimizations and acceleration attempts in the past.

In this thesis, we aimed to integrate two previously developed GPU libraries, namely *GPUSeed* which finds seeds in a *BWA-MEM* like manner and *GASAL2* which aims to accelerate the extension, into the *BWA-MEM* pipeline. The three software components were first analyzed from a theoretical standpoint and then profiled in order to determine the algorithmic and execution differences between the baseline CPU functions of *BWA-MEM* and the accelerated GPU implementations of *GPUSeed* and *GASAL2*. For the *GPUSeed* library, we found that seeding is performed 24× faster compared to the baseline *BWA-MEM*. For the *GASAL2* library, we found around 1.5× better execution time over the baseline, while the main alignment results of the good quality alignments only differ by around 1.8% due to the *GASAL2* acceleration heuristics. Based on our profiling observations, we integrated the three libraries into one complete accelerated *BWA-MEM* implementation and optimized various stages to achieve better performance and lower results deviations. Our integrated solution achieves speedups between 6.82× and 8.72× for single thread execution, depending on the dataset. For multithreaded execution, we find speedups between 2× and 2.8× over the baseline. For the alignment results, we found that our integrated program has around 2% of lines in the main result different from the baseline program, and if we filter alignments with mapping quality below 20, the percentage of different lines reduces to around 1%. Finally, depending on the query dataset, our accelerated library achieved between 60% and 75% GPU utilization during the seeding phase. For the extension part, we found a consistent 100% utilization for a dataset with 250 bases per query when using 14 CPU threads and varying GPU utilization between 40% and 95% for datasets with smaller query sizes. Our accelerated *BWA-MEM* implementation is freely available on GitHub [1].

Acknowledgements

First of all, I would like to begin by thanking my supervisor, Professor Zaid Al-Ars, for his support throughout the time I worked towards my graduation. His weekly advice, insights and encouraging talks helped guide me through this complicated process that is the thesis graduation project.

Secondly, I would like to thank Dr. Nauman Ahmed for providing valuable insights and guidance, especially in the early stages of this project. I would also like to thank him for taking the time to be part of my thesis committee.

I would like to express my gratitude to Dr. Matthias Möller for taking the time out of his busy schedule and agreeing to be part of my thesis committee.

Last but not least, I would like to thank my family and friends for always being there for me and supporting me through thick and thin.

Stefanos Florescu
Delft, September 2022

Contents

List of Figures

List of Tables

1 Introduction

Since the beginning of time, humans have been curious to explore and discover foreign and new places. From the ancient civilizations that traveled across the seas in search of safe settlements and trade to the more recent Age of Discovery, when Europeans traveled to previously uncharted continents. The consensus is that human exploration culminated in the late 20th century when the first person stepped foot on the moon's surface and nowadays continues with our pursuit of other planets and stars. Humanity's quest to continuously move outwards has always been at the center of everyone's attention, and rightly so since it is in our nature to wonder what is out there and conquer the unknown. Nevertheless, maybe the most beautiful story of discovery is not one that happened outwards but one that took place inwards. The Human Genome Project is the story of how humans represented and mapped the first almost complete human genome. This story is not as famous as the story of Apollo 11, but its significance is the same, if not even more remarkable. A culmination of 50 years of unprecedented research and progress in genomics has brought us efficient, affordable, and relatively easy-to-use tools needed for the DNA analysis of today. As we move into a new generation where genomic data is being generated at even higher rates than ever, the software and hardware tools should keep up so we can keep making giant leaps forward in DNA analysis.

This thesis explores the acceleration, integration and optimization of a DNA alignment library, namely the Burrows-Wheeler Alignment tool (*BWA*) [2]. As we will see, this tool is just one piece in the DNA analysis puzzle, but its widespread use across many different analysis pipelines due to its high accuracy and relatively good speed makes it one of the most popular and most cited alignment tools [3, 4]. The following sections of this introductory chapter discuss the motivation behind this thesis and the objectives set for this project. The final section presents an outline of the chapters to follow.

1.1. Motivation and Problem Statement

The modern study of DNA and genomics has enabled us to move towards a future where science can help improve our quality of life even further. Genomics has a vital role in modern medicine and pharmaceutics; some examples of these applications include the early detection and treatment of hereditary diseases and genetic disorders, as well as the identification and creation of vaccines for new viruses and bacteria [5]. In addition, to clinical applications, the market has also adopted some more recreational applications, such as ancestry and heritage identification, making the technology's capabilities more relatable to everyone [6].

Most people in the bioinformatics community attribute the giant leaps in the industry to the Human Genome Project, which ran from 1990 to 2003. The primary outcome of this international effort was the first mapping of the entire euchromatic human genome, with a size of around 2.85 billion DNA letters [7]. On top of this achievement, scientists created a plethora of new streamlined methods and tools, which helped drive down the cost and time required for DNA mapping and analysis. These accomplishments led to the era of Next Generation Sequencing (NGS), where DNA could be extracted and processed in what is called massively parallel sequencing yielding large numbers of relatively shorter sequences [8]. As the market adopts more and more use cases of DNA analysis, the data being produced grows at a very high rate. Researchers project that the genomics study will generate between 2 and 40

exabytes of data until 2025 [7]. This growth is not only caused by the increasing commercial adoption of genomic analysis but also by the new generation of sequencing techniques which create much larger reads of DNA sequences [9]. These new techniques are vital in our pursuit of a more accurate and complete mapping of genomes. Nevertheless, most commercial applications utilize NGS methods, while third-generation techniques are still used primarily in research environments. To sum up, the need for scalable, cost-effective, and accessible software and hardware solutions is higher than ever before and will continue to grow as Big Genomic Data becomes ”bigger.”

The software utilized in a DNA analysis pipeline can include many different components. These components can be from sorting functional units to units used for filtering and merging data segments [10]. In most cases, implementations differ based on what type of analysis the specific pipeline is meant to perform. One of the essential software tools used is the so-called aligner, which aims to map a piece of DNA, the read, to another, usually a reference genome. A closer investigation of the alignment performance in the context of a complete NGS pipeline is crucial as this task takes a significant portion of time out of the complete pipeline [11]. Again, different alignment methodologies depend on the application and the type of DNA data targeted. One of the most famous and utilized alignment solutions is the Burrows-Wheeler Alignment Tool (*BWA-MEM*) [12]. This tool is utilized in many NGS pipelines due to its excellent accuracy and performance for reads from NGS machines [4]. As most alignment pipelines, *BWA-MEM* runs on the CPU, and its performance scales well with an increasing number of processing threads up to a certain extent [13]. Furthermore, the alignment of DNA reads is a good candidate for acceleration since the reads are independent, and the algorithms can be efficiently parallelized. As a result, researchers in the past have utilized various implementations that take advantage of accelerator platforms such as GPUs (Graphics Processing Units) or FPGAs (Field Programmable Gate Arrays) to reduce the significant portion of time taken by alignment [14, 4, 15, 16, 17]. In this thesis, we explore the implementation of the most time-consuming tasks of *BWA-MEM* on a consumer GPU and look at various optimization methods to gain as much performance as possible.

1.2. Thesis Objectives

The main objectives of this thesis are divided into two parts. The first part analyzes the three software tools utilized in this thesis. Namely, *BWA-MEM*, *GPUSeed* and *GASAL2*.

1. Analyze the performance and profile the baseline *BWA-MEM* implementation.
2. Analyze the performance and profile the baseline *GPUSeed* algorithm.
3. Analyze the performance and profile the implementation of *GASAL2* with *BWA-MEM*.
4. Present theoretical maximums concerning application speedup.

The second part focuses on integrating and optimizing these three software components. Also, in this set of objectives, we include the performance evaluation and correctness of the alignment results.

1. Integrate existing baseline *BWA-MEM* with *GASAL2* and *GPUSeed* into one C/C++/CUDA library.
2. Create a data batching strategy for the complete alignment pipeline.
3. Perform kernel and pipelining optimizations to ensure high GPU utilization and overlap of CPU-GPU execution.
4. Optimize integrated library using various techniques.
5. Improve the quality of the results coming from individual stages of the algorithm.
6. Evaluate the speedup and various performance metrics.
7. Evaluate deviation of the alignment results from the integrated solution compared to the baseline.

1.3. Thesis Outline

This thesis report is organized in the following way.

Chapter 2: Background
The second chapter presents all the background information regarding this project. The first part discusses a few key elements regarding DNA sequencing and a brief history of the significant milestones in sequencing technology. The second part looks within the DNA sequencing pipeline at the particular focal point of this thesis, namely the DNA alignment. Finally, the last part of this chapter looks at the software and hardware platform considerations required for DNA alignment.

Chapter 3: Related Work
This chapter presents past work related to and utilized in this project. First, we present the three software components and their respective pipelines. More specifically, we present the *BWA-MEM* application together with the theoretical knowledge required to understand how it performs sequence alignment. Furthermore, we present the *GPUSeed* and *GASAL2* libraries and how their accelerated pipelines aim to emulate functions of *BWA-MEM* while efficiently utilizing a different platform, the GPU.

Chapter 4: Analysis
The analysis chapter looks at the inner workings of the three software components presented in the previous chapter from a practical standpoint. Then, each software component's execution profile is analyzed and discussed. Also, we offer measurements and calculations regarding their initial performance and the utilization of computing resources.

Chapter 5: Integration & Optimization
Chapter 5 presents the integration and discusses the design choices. The second part looks at improving the integrated solution so that the performance of our integrated library is as high as possible. The optimizations are presented incrementally to distinguish between the different improvements.

Chapter 6: Measurements & Performance
In the measurements and performance chapter, we present the hardware setup and the datasets used for evaluating the performance and correctness of our implementation. Then, our integrated library's performance and alignment results are compared against the latest baseline *BWA-MEM* software. In addition, we present a study into the utilization of computing resources by our accelerated library.

Chapter 7: Conclusions
The final chapter concludes this thesis with an overview of what was presented in all the previous chapters. Furthermore, we present the main contributions of our work and future recommendations regarding improvements that can be explored for our accelerated aligner.

2

Background

This chapter presents the background information regarding the biological and computational concepts behind DNA analysis. Sections 2.1 and 2.2 begin by discussing some basic information on DNA and its analysis. After the presentation of some of the fundamental concepts regarding DNA sequencing and analysis, we continue by discussing the focus point of this thesis out of the analysis pipeline, DNA alignment. Finally, this chapter is concluded by looking at high-performance computing platforms and how these can be used to accelerate DNA alignment.

2.1. DNA Background

Deoxyribonucleic acid (i.e., DNA) carries all genetic information in all living organisms. At its core, it is a set of instructions that define how a cell will function, grow, reproduce and develop [18]. DNA is what creates dissimilarities between species as well as within the same species. At the same time, this genetic code only differs by 0.1% between any two human beings, but this difference is enough to create each person's distinctive features [18].

The basic building blocks of DNA are nucleotides, which consist of a sugar molecule attached to phosphate groups and one of the four nucleobases which contain nitrogen [19]. The four bases are adenine (A), cytosine (C), guanine (G) and thymine (T). Multiple nucleotides are arranged in so-called polynucleotides, forming the famous DNA spiral called a double helix [19]. The double helix structure and the basic nucleotide with their building blocks can be seen in Figure 2.1. The bases connect in the following way in order to yield base pairs. Adenine (A) only links with thymine (T) and vice versa, while cytosine (C) and guanine (G) can only connect with each other.

At the core of each cell (i.e., nucleus), one can find threadlike structures in which part or complete pieces of DNA are compacted [20]. These structures are called chromosomes, and their name comes from the Greek words for color (chroma) and body (soma) since these structures are heavily stained by dyes during research. The function of chromosomes is to ensure the DNA is correctly copied during cell division since a living organism constantly grows new cells to replace the worn-out ones [20]. Different species will have different chromosomes; for example, humans have 23 pairs of chromosomes, with the first 22 pairs called autosomes while the last pair is called an allosome and defines the person's sex [20]. Each pair of chromosomes is composed of a chromosome inherited from the person's mother and one chromosome inherited from the person's father, with parts of these chromosomes making up the person's genes. The differences between the DNA of one person and the next are caused by alleles which are forms of the same gene that have slight alterations in the sequence of the DNA bases [21]. The complete set of 23 pairs of chromosomes forms a person's genome [22].

2.2. DNA Sequencing

As mentioned in the previous section, a DNA molecule can be represented as a string containing the four bases, but being able to represent DNA as a string requires sequencing. In other words, sequencing is the laboratory process of representing a DNA molecule as a sequence composed of four bases.

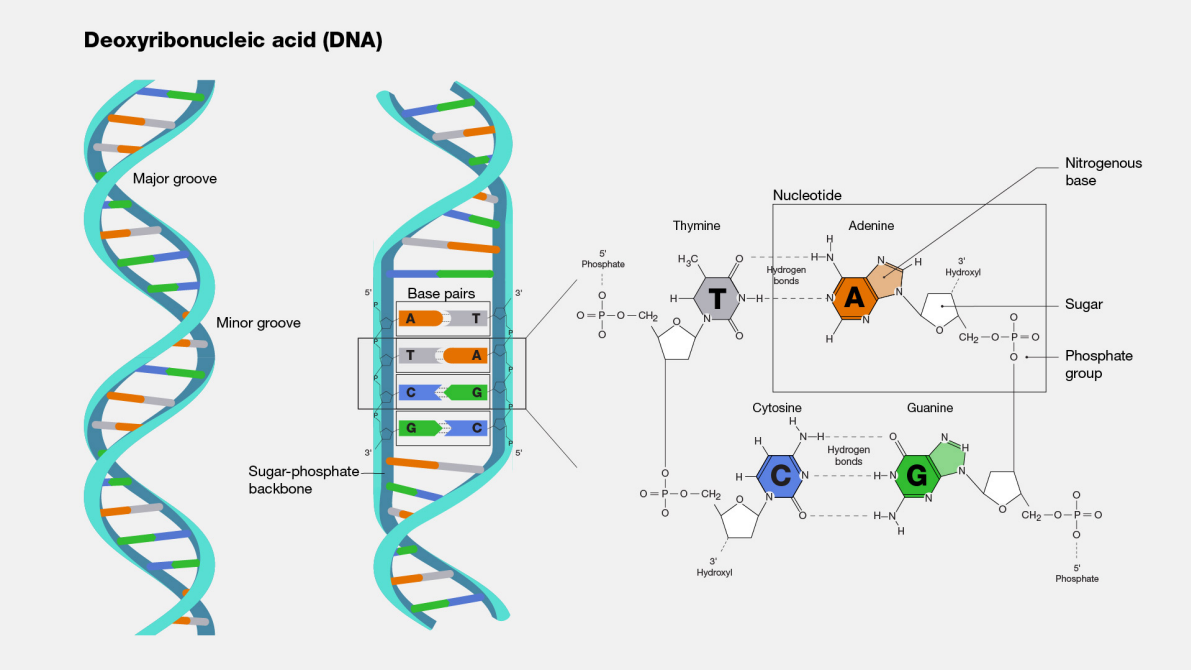


Figure 2.1: Deoxyribonucleic acid (DNA) double helix and nucleobases pairs taken from [18].

Modern DNA sequencing dates back to the late 1970s when Frederick Sanger used the dideoxynucleotide chain termination method to sequence human mitochondrial DNA [23]. As it was later referred to, the Sanger method is still used today but primarily for sequencing reads between 500 and 1000 base pairs (bp) and is considered part of the first generation of sequencers. The main drawback of this method is the low throughput, and high cost [23].

The high price and complex extraction methods led researchers to consider alternative sequencing methods. As a result, second-generation sequencers started appearing around the mid-1990s, with the primary goal of producing high-throughput and low-cost sequencers that have future scalability. Another significant characteristic of this era is the relatively short read length produced by the sequencer of around 50 to 500 bp [24]. Some notable sequencers developed in this generation are Roche's 454, which allowed the high parallelization of the reactions that give the sequences [25], and Illumina's sequencing technology which became the first highly parallelized sequencing technology available on the market [23]. These technologies are known as Next Generation Sequencing (NGS) and are some of the most utilized methods today. An essential outcome after the adoption of NGS technologies is the drastic reduction in the cost required to map a human genome. As shown in Fig 2.2, the cost nowadays is around $1000 while twenty years ago it was around $100,000,000. Furthermore, it is essential to mention the Human Genome Project, which concluded in 2003 and ran for 13 years and was the first international effort to sequence the human genome in its entirety. The cost of this project was around 300 million dollars, and for the first time in history, the euchromatic regions of the genome, which make up around 92.1% of the human genome, were sequenced [26]. On top of this achievement, methods and applications developed during this project paved the way for the market adoption of NGS and the sequencing cost reduction mentioned earlier.

The final and third generation of sequencers, such as the Oxford Nanopore and PacBio, produces very long reads in the order of kilobases [24]. The need for longer reads came from the fact that genomes are inherently very complex, with many areas that repeat, and the short reads produced by the second generation of sequencers could not efficiently map them [24]. In addition, third-generation sequencers remove the need for DNA amplification which was an expensive and time-consuming procedure.

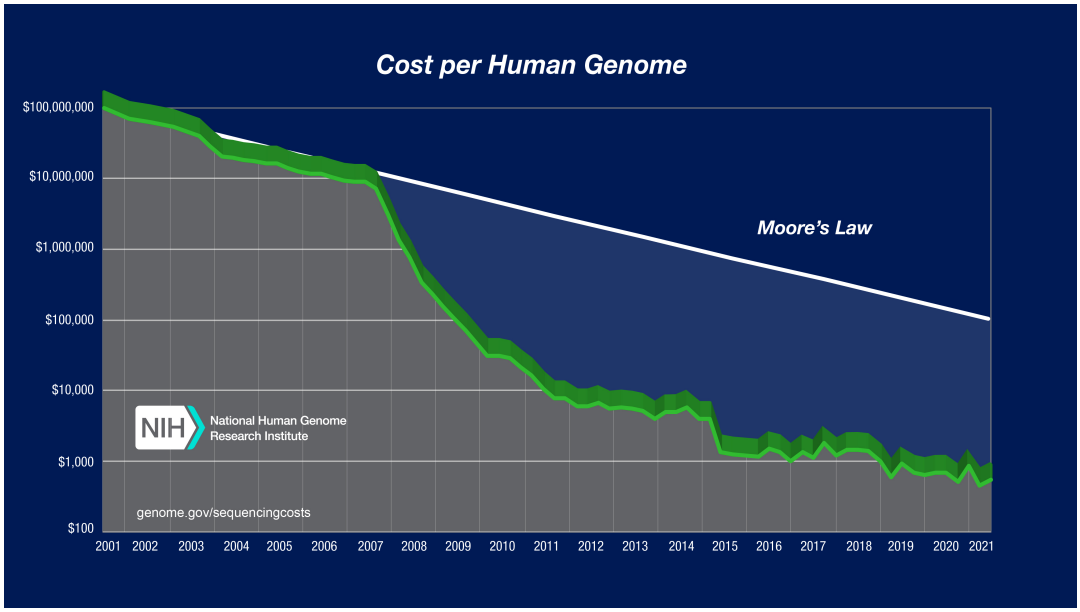


Figure 2.2: Cost per Genome 2001 - 2021 taken from [27].

The continuous reduction in sequencing costs has led to vast market adoption of whole genome sequencing (WGS) and analysis. From law enforcement and healthcare applications to sequencing the virus behind a global pandemic and finding a cure that can return life to normal, DNA analysis is rapidly becoming a vital part of our everyday lives. As DNA sequencing becomes more affordable and its use more widespread, we mention some interesting applications of DNA sequencing [5]:

- Detecting prenatal and newborn pediatric and genetic diseases.
- Pharmacogenomics refers to personalized pharmaceuticals for a person's specific genetics.
- Detecting a person's predisposition to cancer and other diseases.
- Rare virus detection and treatment.
- Gene therapy and gene editing.

2.3. DNA Analysis

A typical NGS pipeline's workflow consists of three steps [28]. The first is library preparation, where the DNA sample is prepared so that it can be processed by the sequencer [28]. The second step is sequencing which was described in the previous section. Finally, the third step is alignment and data analysis. In this step, the newly sequenced reads (also referred to as query sequences) are mapped to a reference genome, and the mapped result is analyzed using various tools. In other words, sequence alignment finds highly similar regions of nucleotide bases between the queries and a reference genome. Each alignment receives a score depending on how well the two regions match.

Alignment and analysis have their respective pipelines, which differ from one implementation to the next. For example, Figure 2.3 presents the *GDC DNA-Seq* analysis pipeline which detects somatic variants inside a whole genome sequence (WGS) and whole exome sequencing (WXS) [10]. Somatic variants, also called somatic mutations, are changes in DNA that occur after human conception and are transmitted neither from the father nor the mother [29]. These variants are, in some cases, responsible for cancer and other diseases, so their detection is vital for the prevention and treatment of the patient [29]. A more general definition that includes the previously mentioned somatic variants is the variant calling analysis which has the goal of identifying the following genetic variations:

- Single-nucleotide polymorphisms (SNP), the most common genetic variations among humans, refer to a single base difference between two DNA sequences [30].

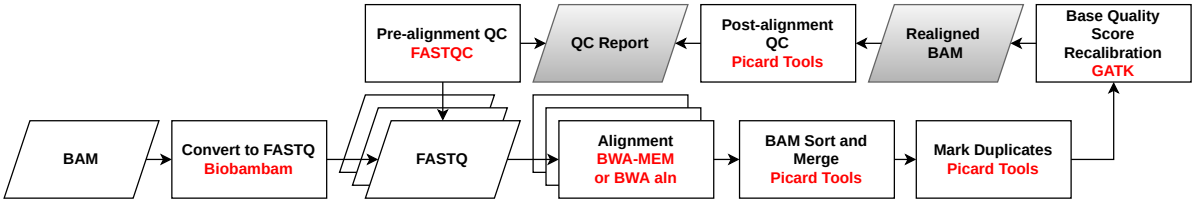


Figure 2.3: GDC DNA-Seq Analysis pipeline recreated from [10].

- Indels refer to insertions, deletions, or both insertions and deletions of bases in a DNA sequence [31].
- Structural variants (SVs) are defined as larger portions of DNA that have variations compared to DNA of the same species [32].

Our focus in the context of this thesis falls on just one block in the analysis pipeline and, more specifically, on the alignment problem. The importance of performing alignments of sequences in the context of the complete DNA pipeline is very high. This significance is reflected mainly through quality and execution time. By quality, we refer to the fact that if the regions of similarity are not correctly found, the downstream analysis will suffer, and eventually, not all mutations or unique characteristics will be marked correctly, yielding an incorrect mapping. As far as execution time is concerned, we can see in Figure 2.4 that mapping takes a significant chunk of time of around one-third out of a typical variant calling workflow which has accelerated portions [33]. Figure 2.4 also shows that in the context of the complete pipeline running on just CPUs, alignment time is not that significant, but once other stages, such as variant calling, are accelerated and improved, alignment can become the slowest part of the pipeline. Furthermore, it is crucial to note that the results from Figure 2.4 show the alignment algorithm *BWA-MEM* running in parallel on a cluster with 16 nodes, each running one instance of an alignment job, meaning that this is a distributed implementation that benefits improved execution [33].

2.4. Sequence Alignment

The alignment problem, as mentioned earlier, is the process of taking a read and trying to find where it fits the best on a reference genome. This task can be reduced to an extended string matching problem where an initial exact matching sub-string between the read and the reference is found and then extended using some logic to contain areas around the match. This methodology is called seed-and-extend and is utilized by most short-read aligners [34]. The alignment between these two strings is given a score depending on the number of base insertions, base deletions and base mismatches between the two strings of nucleotides. DNA mutations are biological errors that appear in sequences in the form of deletion, insertion, or substitution of one or multiple bases. Therefore we are also interested in a scoring scheme that provides valuable information on how similar or how different two regions of DNA are.

2.4.1. Alignment Examples

An example of mapping can be seen in Table 2.1, where a read sequence is compared against a longer reference. The task, in this case, is to find the position where the read fits best in the reference, and in this specific scenario, the whole read matches a sub-string of the reference that starts at position 1.

Position	0	1	2	3	4	5	6
Reference	**C**	**T**	**A**	**G**	**C**	**G**	**C**
Read		**T**	**A**	**G**	**C**		

Table 2.1: Example of exact alignment between a reference and a read.

The read in Table 2.2 is missing the base '*G*' at index 3 of the reference, and the aligner should mark this as a deletion. Therefore in the seeding step, the aligner should detect two different seeds on the left and on the right of that deleted base and perform a correct alignment so that the two sequences

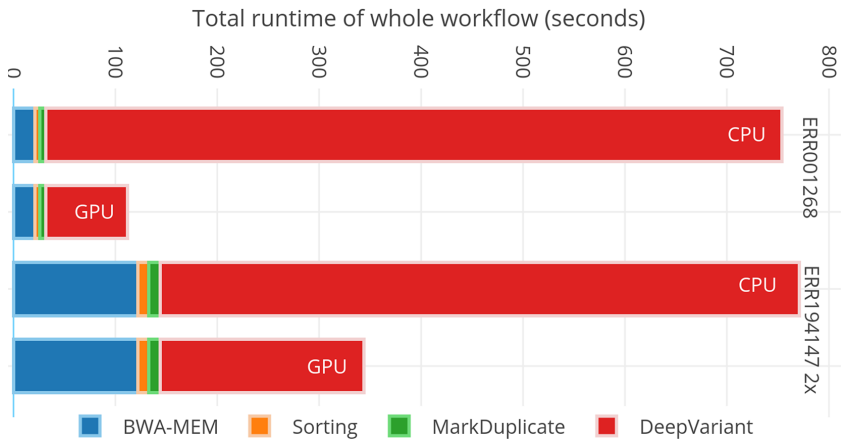


Figure 2.4: *Deepvariant* calling workflow with NGS data set taken from [33].

are aligned like in Table 2.2. Also, it is essential to note that we can say a base is deleted in the query or a base is inserted in the reference, and these mutations are what we mentioned earlier as *indels*.

Position	0	1	2	3	4	5	6
Reference	**C**	**T**	**A**	**G**	**C**	**G**	**C**
Read	**C**	**T**	**A**		**C**	**G**	**C**

Table 2.2: Example of alignment between a reference and a read with one deleted base.

Table 2.3 shows the opposite scenario of what was described in the previous example. In this case, a base was inserted in the read, or a base was deleted in the reference.

Position	0	1	2	3	4	5	6	7
Reference	**C**	**T**	**A**	**G**	**C**	**G**	**C**	
Read	**C**	**T**	**A**	**T**	**G**	**C**	**G**	**C**

Table 2.3: Example of alignment between a reference and a read with one base insertion.

The final read in Table 2.4 has the fourth base changed from '*G*' to '*A*', which should be marked as a mismatch.

Position	0	1	2	3	4	5	6
Reference	**C**	**T**	**A**	**G**	**C**	**G**	**C**
Read	**C**	**T**	**A**	**A**	**C**	**G**	**C**

Table 2.4: Example of alignment between a reference and a read with one base substitution.

These cases present an elementary example of just a short reference with only one point base mutation, but when seeding-and-extending millions of NGS reads over a reference human genome of approximately 3 billion bases, the solution becomes more complex. Therefore, a naive string searching method cannot be applied since it will provide a solution in time relative to the reference size, which, as we said, is very lengthy in cases of complete genomes.

2.4.2. Alignment Methods

The alignment problem can be reduced to the process of editing two biological sequences using insertions, deletions, gaps and substitutions in order for the two sequences to match each other as closely as possible [35]. This problem is solved using dynamic programming algorithms, which provide an optimal solution but are relatively slow and have high memory requirements [36]. Furthermore, performing sequence alignment is a computationally intensive task, with the algorithms having quadratic runtime with respect to the size of the sequences being aligned [37]. This is where seeding comes in

and improves the alignment execution time by performing alignments only around matching subregions of the sequences [37]. In addition to seeding, most aligners introduce additional heuristics to reduce the runtime even further [37].

The most famous dynamic programming algorithms used for sequence alignment are the Needleman-Wunsch algorithm [38], and the Smith-Waterman algorithm [39]. The types of alignments performed on DNA sequences are the following:

- **Global alignment**: also known as end-to-end alignment, is a form of a global optimization problem that tries to find the maximum alignment score while the alignment spans the entire length of the query [35]. The alignment scores for this type of alignment can be negative, especially when the ends of the two sequences are very different. The Needleman-Wunsch algorithm is used for this type of alignment.
- **Local alignment**: is a form of alignment that only focuses on finding the maximum alignment score between two sequences. In this type of alignment, the ends of the query may not be contained in the alignment, and their exclusion does not penalize the score [35]. The alignment scores for this type of alignment stay positive. The Smith-Waterman algorithm is used for this type of alignment.
- **Semi-global alignment**: is a mix of the two previous methods in the sense that it performs global alignment but allows skipping both ends of the query sequence without penalty [35].

In the next chapter, we will detail how the dynamic programming algorithm works for sequence alignment and also discuss some of the heuristics used to improve performance.

2.4.3. Popular Alignment Tools

In this part, we present some of the most popular short-read alignment tools utilized. An excellent and comprehensive list of the most popular aligners is presented by Musich R. et al. in [40]. The authors also present a performance comparison between these aligners where their speed and alignment accuracy are being evaluated. The most utilized aligners, according to the authors in [40] are:

- **BWA** [2]: The Burrows-Wheeler aligner developed by Heng Li utilizes a combination of the Burrows-Wheeler transform (BWT) [41] and a variation of the FM index introduced by Ferragina and Manzini in [42], to index the reference and reduce the time required for string matching [34].
- **Bowtie2**: Utilizes a similar index to *BWA* and calculates the mapping location based on the FM index [40].
- **MUMmer4**: Uses the suffix array, which allows the processing of genomes up to 141 trillion base pairs [40].
- **HISAT2**: Overall faster than *BWA* but with lower alignment rate [40]. *HISAT2* implements a graph-based FM index for the reference genome with a larger index and many smaller ones making memory access more efficient [40].

The comparison of these aligners has the *HISAT2* aligner as the fastest, but *BWA* is the most accurate for reads up to 500 base pairs [40]. In sequence alignment, speed may not be the number one priority, and many consider that a good trade-off between speed and accuracy is more important. Furthermore, the Burrows-Wheeler aligner, specifically the newer version with *BWA-MEM (Maximal Exact Matches)* [12], is one of the most utilized and robust short-read alignment tools. Most variant call workflows utilize *BWA-MEM* for their alignment, making it the industry standard for short-read alignment.

2.5. High-Performance Computing Platforms

2.5.1. Multicore and *Amdahl's Law*

Nowadays, central processing units or CPUs are still the main components utilized for the vast majority of computations. Their main power lies in that they are relatively easier to program, can handle I/O on

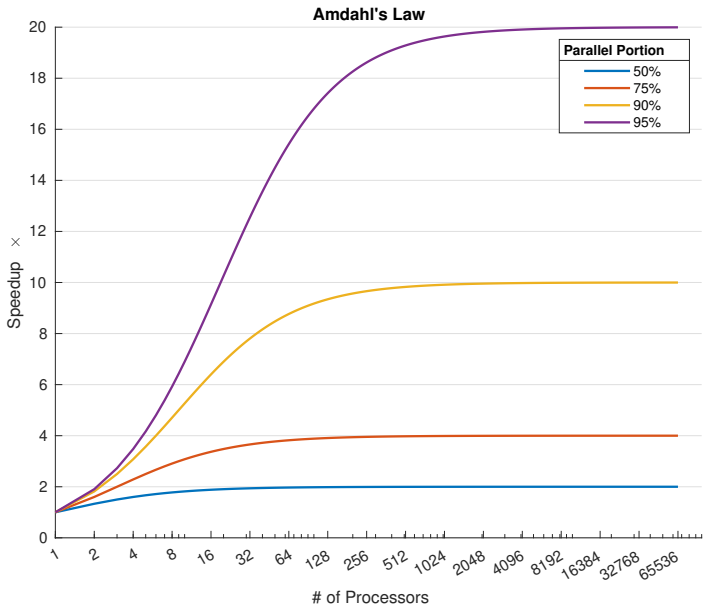


Figure 2.5: Theoretical speedup given by *Amdahl's Law* for varying parallel program fractions.

their own, and have excellent sequential performance in a wide variety of tasks. On the other hand, over the last decade, the increase in processing power of CPUs has been underwhelming, with many claiming that Moore's Law is dead. Moore's Law refers to the observation that the number of transistors contained in a microchip will double every two years, and as a result, the speed and capabilities of computers are also expected to increase [43]. Although the number of transistors still follows the previously mentioned trend, the capabilities of processors have only had incremental improvements. In the consumer market, this stagnation was mainly due to a lack of competition, with just one company selling the majority of consumer CPUs [44]. By looking at the server space, the gap is even more significant, with Intel having a constant market share of over 80% [44]. The last four years have seen a significant revival in the consumer market mainly due to AMD and its use of CCXs or Core Complexes which enabled them to bring higher core count CPUs at more competitive prices. This new competitive market has brought some innovations, with the most important being higher core count CPUs becoming the industry norm rather than the exception and the first heterogeneous consumer CPUs utilizing two types of cores in one package, namely performance and efficiency cores.

As we continue navigating this multicore era of processors, it is important to mention that increasing the number of cores is not useful if the algorithms we run on them are not properly implemented to leverage multiple cores. In other words, the programmers must write programs that can execute in parallel on multiple cores. *Amdahl's Law* is a formal definition that gives a theoretical upper bound on the speedup that can be achieved for an increasing number of processing cores. In Equation 2.1, f refers to the fraction of the program which can run in parallel while $1 - f$ is the sequential portion. The speedup obtained when utilizing n processors is given by Equation 2.1. Figure 2.5 shows these upper bounds for varying parallel portions and an increasing number of cores. From Figure 2.5, we deduce that for programs where the parallel portion is small, increasing the processing cores does not yield an improvement due to the upper bound imposed by f. As a result, it is much more important to have a large fraction of a program that can run concurrently rather than a high core count. In addition, *Amdahl's Law* does not account for overhead induced by scheduling, memory bandwidth, or I/O bandwidth and provides a best-case scenario for speedup. Finally, the maximum attainable speedup where infinite resources are available can be calculated by taking $n \to \infty$, giving us the upper bounds of Figure 2.5.

$$S = \frac{1}{(1 - f + \frac{f}{n})} \tag{2.1}$$

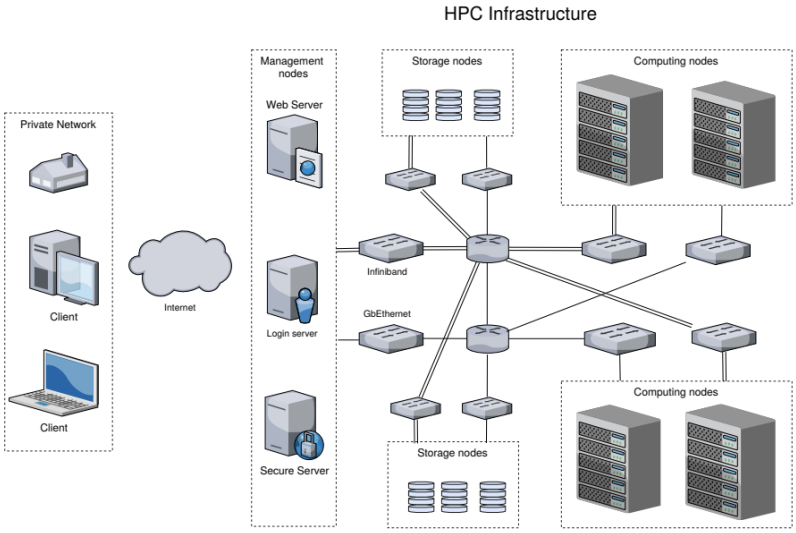


Figure 2.6: Simplified example of the architecture of an HPC system from [45].

2.5.2. High-Performance Computing Platforms for DNA Alignment

After the introduction of Next Generation Sequencing techniques and the conclusion of the Human Genome Project, the cost of mapping a genome has followed a steep downward trend. Up to 2007, the price per genome followed the famous observations made by Gordon Moore. However, in Figure 2.2 can be seen that after 2007, the price per genome saw a steeper decrease which lasted until around 2015, mainly due to the market adoption of NGS techniques. At the moment, we move into an era where the commercial use of NGS methods will continue to increase while third-generation sequencing techniques will also begin to be applied in more commercial scenarios. This new genomic era will be defined by the exponential increase in the production of new data [7], which will require affordable and fast storage as well as high-performance and scalable computing platforms.

2.5.3. Homogeneous Computing Platforms

Today, most DNA analysis algorithms are processed on high-performance computing clusters (HPCs), which usually employ traditional computing nodes connected using a high-speed connection. Each node consists of a CPU (or, in some cases, more) with multiple processing cores connected to its dedicated memory and storage. Of course, this general configuration may vary from one case to another. In such an environment, the analysis is usually performed by splitting the problem among the nodes and then aggregating the result at the end. In order to run a program in such a distributed environment, high-performance computing tools like the Message Passing Interface (MPI) and big data frameworks like Hadoop and Spark are utilized [33]. An example of the topology of such an HPC system can be seen in Figure 2.6.

One of the main benefits of utilizing such an approach, especially for DNA alignment, is the simplicity of breaking the problem into smaller chunks and then solving the smaller chunks in parallel. The simplicity is induced due to the independence between different DNA reads, meaning there are no data dependencies between reads. The second advantage of such a framework is the minimal algorithmic changes required to run a program in parallel and significantly improve execution times. In the case of *BWA-MEM* which has multiple complex stages, it is a difficult programming task to create efficient parallelized algorithms for each stage, and in order to obtain a decent speedup, it is required to parallelize as much as possible as we saw earlier in *Amdahl's Law* [11]. At the same time, the challenge becomes implementing an efficient distributed logic that breaks the problem and efficiently distributes work to all the nodes so that utilization is kept high. Some examples of the *BWA* software being implemented in such frameworks are *SparkBWA*, *BWASpark* and *PipeBWA* [33].

Although CPUs are considered jacks of all trades and their flexibility in dealing with a wide variety of tasks is their main strength. Over the past few years, their performance scaling has been underwhelming compared to GPUs [46] and in combination with the CPU's more limited parallelization potential due to the much lower core count, most modern high-performance cluster architectures have resorted

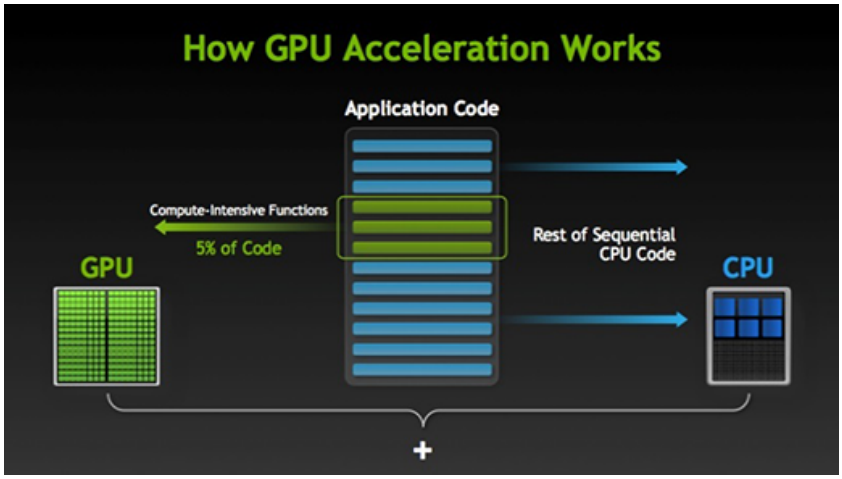


Figure 2.7: How GPU acceleration works from [48].

to the use of accelerators to offload highly-parallelizable tasks, such as DNA alignment.

2.5.4. Heterogeneous Computing Platforms

Today's most common accelerators are GPUs and FPGAs. In heterogeneous computing platforms, an accelerator or more is attached to a node. The node acts as the *host* which runs the program and offloads highly parallelizable sections of it to the accelerator, known as the *device*. Figure 2.7 shows an illustration of a piece of code that runs on the *host*, with the blue lines being the sequential parts while the green lines are known as kernels which are functions running on an accelerator; in this case, a GPU. The main task in heterogeneous platforms becomes how to efficiently program those sections of code to achieve as much speedup as possible while the system's resources are efficiently utilized. Most modern platforms for DNA analysis are heterogeneous and include multiple accelerators. The parts of the DNA pipeline which are usually accelerated are the base calling, which is the process of turning raw device signals in one of the four nucleotide bases, and variant calling, especially in cases where convolutional neural networks are utilized, such as Google's *DeepVariant* [47].

2.5.5. General Purpose GPU Architecture

Modern-day GPUs have moved on from their role as devices utilized primarily for processing and displaying graphics into highly efficient devices which can exploit data-level parallelism to perform a vast number of calculations in parallel. Compared to a CPU, a GPU has thousands of concurrent threads executing the same task. For example, problems like matrix multiplication which includes many independent calculations can be offloaded to a GPU that can perform this calculation in parallel instead of the conventional sequential solution involving looping over rows and columns. At the same time, GPU cores have lower clock speeds and a more specialized instruction set than CPU cores, making them less versatile. Also, GPUs include their onboard memory used for all their calculations.

The power of GPUs lies in how well the programmer can parallelize a program and how efficiently it can leverage this many-core architecture to hide the latency induced by the slower but many cores.

There are two major frameworks used to program GPUs, *OpenCL* and *CUDA*. The first framework has the advantage of being able to program GPUs from both NVIDIA and AMD, as well as being open and royalty-free since a consortium of companies develops it. The second main programming framework for GPUs is *CUDA* (*Compute Unified Device Architecture*) which targets only NVIDIA devices. This framework is a proprietary API created by NVIDIA, which does not have the open-source nature of *OpenCL*, but its advantage lies in the extensive number of high-performance libraries and the much larger community. *CUDA* can be considered an extension of C/C++ with newer implementations allowing programmers to utilize even higher-level languages such as Python and Java through the use of bindings [49]. This thesis focuses on a *CUDA* implementation targeting NVIDIA GPUs in C/C++ project.

Previously we briefly introduced the terms *host* and *device* to refer to the CPU and GPU, respectively, and in this part, we will look at how a *host* can offload a kernel to a GPU *device*. The typical

workflow of such an offloading procedure can be summarized in three steps:

1. Transfer required data for computation from *host* memory into the *device memory*.
2. Launch kernel from *host* to perform GPU computation on previously transferred data.
3. Transfer computed data from *device* memory into *host* memory for further computation or output display.

Figure 2.8 shows a more detailed representation of a sequential algorithm being executed on a CPU which launches *Kernel0* followed by another sequential portion which finally leads to a second *Kernel1* running on the *device*. This example of heterogeneous program execution is how most programs utilizing an accelerator operate. Executing a kernel on the *device* requires a few definitions. Kernels are executed on a *grid* which is divided into *blocks* of *threads* [50]. The number of *threads* per *block* and number of *blocks* per *grid* are defined by the programmer within the launch parameters of each kernel. The organization of a kernel can be seen in Figure 2.8 for two different kernels with a different number of *threads* per *block* as well as different *block* organization.

CUDA GPUs are built around a concept known as *Streaming Multiprocessors (SMs)*, which are the execution units for *blocks*. When a kernel is called for execution the *blocks* in a *grid* are enumerated and sent for execution to available *SMs*. A *SM* executes hundreds of threads in parallel and can even execute different *blocks* concurrently if the resources allow it. *CUDA* multiprocessors employ an architecture named *SIMT (Single Instruction Multiple Thread)* which is similar to the more well-known *SIMD (Single Instruction Multiple Data)* [49]. This architecture is utilized so the programmer can issue instructions to multiple threads to work on independent data parts. Another important definition is *warps*, which refers to groups of 32 *threads* scheduled by the multiprocessor. All the *threads* in a *warp* begin executing together at the same program address, but each thread can branch into different program addresses [49]. The most efficient implementation is one where all 32 *threads* in a *warp* execute the same instruction. In other words, *CUDA* kernels should avoid branching instructions, or if there are branching instructions, then the *threads* within a *warp* should branch together.

GPUs have their own memory hierarchy, separate from the CPU and used for kernel execution and data transfers back and forth with the CPU. In Figure 2.9, the three main memory spaces found on a GPU are presented as well as their respective scopes. The *CUDA* architecture exposes the following memories which we order based on their access time with the lowest first [50]:

- **Registers** are private to each *thread* and visible only by that particular *thread*. The compiler allocates the number of registers for each *thread* at compilation time based on how many variables are defined in a kernel.
- **L1/Shared memory** is visible by all the *threads* within a *block* and can be used to rapidly exchange data among those *threads*. Shared memory has the lifetime of the *block*.
- **Read-only memories** for each *SM* exists a texture memory, instruction cache, read-only cache and constant memory. This memory is only visible to the *blocks* within that *SM*.
- **L2 Cache** is shared among all *SMs* and is accessible to all *blocks* in a kernel.
- **Global memory** is the most prominent memory and what users see as GPU memory in their computers. It is used to send/receive data to/from the *host* and for most of the data that resides on the GPU.
- **Local memory** is also private to each *thread* and is allocated and managed in each kernel for that particular thread.

Finally, it is also important to mention some of the capabilities that *CUDA* provides to the programmer when utilizing kernels for acceleration. The first and one of the most influential is non-blocking kernel invocation, known as asynchronous commands. The CPU calls a kernel and then can continue the serial execution of the code following the kernel call, resulting in an overlap of CPU and GPU execution. This increases system utilization and efficiency, which is desired, especially in costly systems. The second capability provided in *CUDA* are *streams* which are a sequence of *CUDA* operations issued

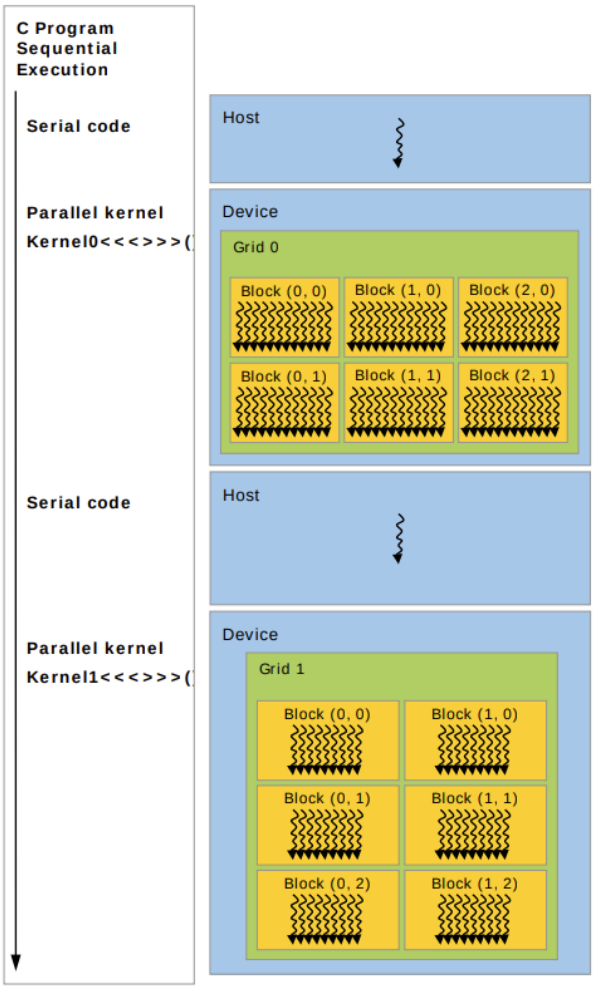


Figure 2.8: *CUDA* serial and parallel execution model from [49].

by the CPU that will execute in a predefined order [51]. Operations not in the same stream can be overlapped and executed concurrently. By default, *CUDA* operations are executed in the *default stream* but since version 7 of the toolkit there is the possibility of defining a *default stream* for each of the *host's* executing threads. Multiple default streams create the opportunity to increase concurrency even further by having multiple threads of the CPU call separate GPU kernels. The third and final capability is using multiple *streams* on the same CPU thread. This capability enables the programmer to launch kernels or perform memory transfers asynchronously from the same *host* thread in different streams if the conditions for implicit synchronization are met [49].

2.5.6. Accelerating DNA Alignment

In the context of this thesis, we utilized consumer GPUs to accelerate the seeding and extension of the *BWA-MEM* algorithm. Most previous implementations discuss the acceleration of the seed or the extension phase separately, specifically utilizing methods such as the FM index and the Smith-Waterman algorithm. For example, in [14] and in [15], the implementation of the seed extension of *BWA-MEM* is discussed on FPGA and GPU, respectively. Furthermore, there are publications discussing seed generation on GPUs [52] and on FPGAs. However, to our knowledge, at the moment of writing this report, there were no freely available integrations of *BWA-MEM*, which included a complete pipeline of generating seeds and then performing the extension on a consumer-grade GPU.

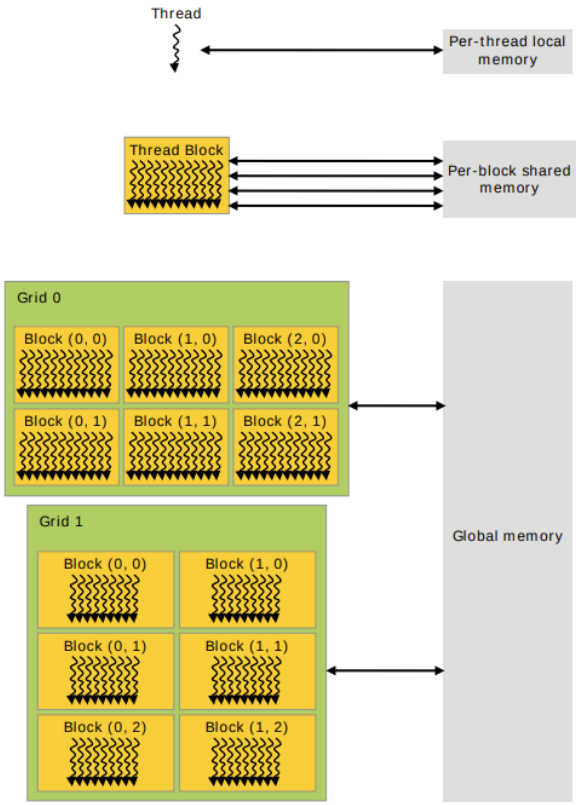


Figure 2.9: *CUDA* memory hierarchy from [49].

3

Related Work

This chapter will present the three related work software components used during the subsequent integration chapter to accelerate the *BWA-MEM* pipeline. The discussion will begin by introducing the Burrows-Wheeler aligner and, specifically, how it performs the seeding and extension. The second section will focus on the *GPUSeed* library, which accelerates the seeding part on a GPU. In contrast, the last part of this chapter will look at *GASAL2*, which is an alignment library that also utilizes kernels for acceleration.

3.1. Burrows-Wheeler Aligner

The Burrows-Wheeler aligner follows the seed-and-extend paradigm as briefly introduced in the previous section. The main goal of this paradigm is to make the alignment faster by looking only at areas where a matching string exists. These seeds are referred to as *maximal exact matches* or *MEMs* for short. We define a *MEM* between string *T*, the reference, and pattern *P*, the query, a matching substring of *T* and *P* which cannot be extended either forward or backward without creating a mismatch [53]. In addition, a *MEM* is said to be a *super-maximal exact match (SMEM)* if it is not contained in any other *MEMs* on the query sequence [53]. *BWA-MEM* finds these seeds and then groups them into chains of seeds. The final step is to extend around the positions of the seeds to create an alignment. The extension uses the *Smith-Waterman* dynamic programming (DP) algorithm. In Figure 3.1 an illustration of the seed-and-extend method utilized in *BWA-MEM* can be seen. The following sections discuss how the seeds are generated, chained and finally aligned. The final part presents the output of the *BWA-MEM* program. The complete *BWA* library is available freely on GitHub [54].

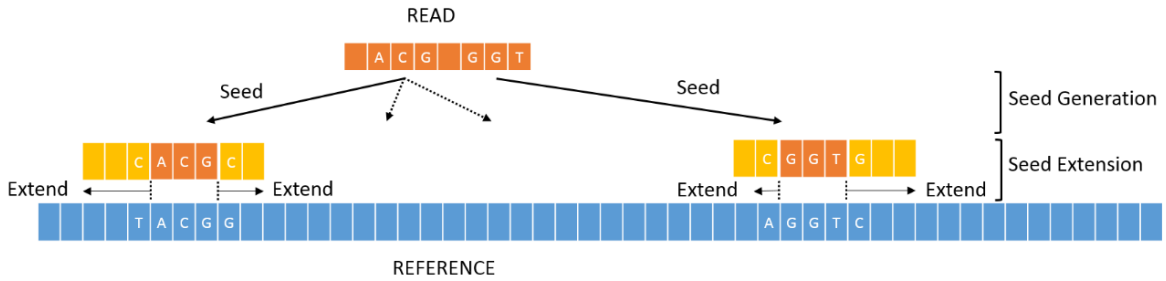


Figure 3.1: *BWA-MEM* seed-and-extend paradigm illustrated from [4].

3.1.1. Seed Generation

Burrows Wheeler Transform & FM Index

The algorithm behind the calculation of seeds in *BWA-MEM* depends on the FM index, a data structure Ferragina and Manzini first introduced in [42]. Supposedly, the FM index stands for ”Full-text Minute-space” since the query time’s complexity is said to be sub-linear with respect to the size of the input.

i	SA[i]		BWT[i]
0	6	$ ataac	g
1	2	aacg$ a	t
2	3	acg$ at	a
3	0	ataacg	$
4	4	cg$ ata	a
5	5	g$ ataa	c
6	1	taacg$	a

Table 3.1: *BWT* of string T = $"ataacg"$.

This index utilizes the underlying properties of the *Burrows Wheeler Transform* introduced by Burrows M. and Wheeler D.J. in [41]. The *BWT* is a string manipulation technique that takes a string, creates all possible rotations, and then sorts the rotations [2]. In the case of DNA strings, we consider that the alphabet contains only the four bases $\Sigma = \{'a', 'c', 'g', 't'\}$. At the same time, we also utilize the special character $ to denote the end of the string and the character $'n'$ to represent an ambiguous base. Furthermore, the lexicographical order of these symbols follows $\$ < a < c < g < t < n$. Also, we define that given a string T, then $T[i], i = 0, ..., |T| - 1$ is the i-th symbol in the string while $|T|$ is the length of the string. Note that throughout this chapter, we will refer to symbol a as any letter from alphabet Σ not to be confused with just the base *'a'*, which will be referred to using quotations.

An example of how the *BWT* is calculated can be seen in Table 3.1 for text $T = "ataacg"$ where the special character $ is added at the end of the string, and then all possible circular shifts are calculated and sorted lexicographically. The last letter from each sorted rotation is the *BWT* of reference T. The main property of the *BWT* is that it can be reversed by storing minimal additional information and, therefore, can be used for block-sorting compression [41].

A second important data structure that appears from the *BWT* is the *suffix array*. This array contains the indices of the starting positions of the sorted suffixes, which are the sub-strings up to the $ sign in each row of Table 3.1. The suffix array for our sample string T is then shown in the second column of Table 3.1 as *SA*. Many indices are built based on suffix arrays, such as suffix trees, enhanced suffix arrays, and FM index [55]. These indices are useful since they enable us to search for matching sub-strings between P and T with complexity independent of the length of T and dependant on the length of P, which in the case of NGS data has at most a couple of hundred bases. The main issue in the case of suffix trees is that it has a much larger memory footprint than a suffix array resulting in around four times more memory required. The final two components that complete the FM index are the occurrence array O and the count array C. The occurrence array $O[a, i]$ gives the number of occurrences of symbol a in BWT up to the i-th element of the BWT array. On the other hand, the count array $C[a]$ gives the number of symbols in T, which are lexicographically smaller than a.

To sum up, the FM index of string T consists of the following components [55]:

- **BWT** of T: the Burrows-Wheeler array which is the blue column in Table 3.2.
- **SA**: the sorted suffix array, which can be seen in the red column of Table 3.2.
- **C**: the count array, $C[a]$ denoting the total number of symbols which are lexicographically smaller than a in the bwt.
- **O**: the occurrence array, $O[a, i]$, holds the number of occurrences of symbol a up to the i-th position in the BWT array.

Maximal Exact Matches using the FM Index

In order to be able to find *MEMs* between a query P of length $|P|$ and a reference T of size $|T|$, we need to define two operations utilizing the FM index. The first operation is known as *count* and aims to determine the number of occurrences of a pattern P in T. The second operation is known as *locate* and seeks to find the position of pattern P in T.

The *count* operation can be reduced to finding a valid suffix array interval where the pattern P appears as a prefix. The lower part of the interval can be defined as in Equation 3.1 while the upper

	SA[i]	BWT[i]	O[a,i]			
i			a	c	g	t
0	6	g	0	0	1	0
1	2	t	0	0	1	1
2	3	a	1	0	1	0
3	0	$	1	0	1	0
4	4	a	2	0	1	0
5	5	c	2	1	1	0
6	1	a	3	1	1	0
		C[a]	0	3	4	5

Table 3.2: FM Index of T = "$ataacg$".

Iteration	q	Symbol a	I^l	I^u
Init	2		0	6
1	2	a	$C('a') + O('a', -1) + 1 = 1$	$C('a') + O('a', 6) = 3$
2	1	a	$C('a') + O('a', 0) + 1 = 1$	$C('a') + O('a', 3) = 1$
3	0	t	$C('t') + O('t', 0) + 1 = 6$	$C('t') + O('t', 1) = 6$
Output			6	6

Table 3.3: FM Index backward search of P = "taa" in T = "$ataacg$".

boundary of the interval is expressed in Equation 3.2 resulting in the suffix array interval of P in T $[I^l(P), I^u(P)]$. Equation 3.3 gives the size of the interval, which is also the number of occurrences of P in T.

$$I^l(P) = min\{k : P \text{ is the prefix of } SA[k]\} \tag{3.1}$$

$$I^u(P) = max\{k : P \text{ is the prefix of } SA[k]\} \tag{3.2}$$

$$I^s(P) = I^u(P) + I^l(P) + 1 \tag{3.3}$$

The main property of the FM index is that string P appears in T if and only if $I^l(P) \leq I^u(P)$ [53]. Furthermore, we define as aP the concatenation of symbol a and string P as a result we calculate the suffix array interval of new string aP by using Equations 3.4 and 3.5 respectively given that P was a sub-string of T. This operation is called *backward search* since we grow our strings backward one DNA character at a time and then recalculate the new suffix array interval of that new string. This operation is repeated until $I^l(aP) \leq I^u(aP)$ does not hold any more or until we use all letters from our query (i.e., we reach the 0 index of the query).

$$I^l(aP) = C[a] + O(a, I^l(P) - 1) + 1 \tag{3.4}$$

$$I^u(aP) = C[a] + O(a, I^u(P)) \tag{3.5}$$

$$I^s(aP) = I^u(aP) + I^l(aP) + 1 \tag{3.6}$$

Using the above-mentioned method we introduce an example where we search for the string P = "taa" in the previously presented T = "$ataacg$". The steps are summarized in Table 3.3.

Algorithm 1 uses Equations 3.4 and 3.5 to calculate all the suffix array intervals for all the *MEMs* which are larger than a minimum *MEM* length, between a reference T and a query P [52]. The algorithm begins at the end of the query, position $|P| - 1$, and adds one base at a time utilizing the *backward search* method we just presented until the first base in the query is reached. The output *MEMs* are in the form of a tuple with $([l, u])$ being the suffix array interval of the *MEM* (i.e., equivalent of $[I^l(P), I^u(P)]$) and $(start, end)$ its starting and ending position in the query [52].

Algorithm 1: Computing the starting position for a given suffix array index taken from [52].

Input: Pattern P and minimum required MEM length min_mem_len
Output: Array M containing all the MEMs in P

```
Function MEM_SA_INTERVAL(P, min_mem_len):
    Initialize M as empty array
    for j ← |P| − 1 to min_mem_len − 1 do
        [l, u] ← [0, |T| − 1]
        q ← j
        while q ≥ 0 do
            prev_l ← l
            prev_u ← u
            l ← C[P[q]] + O(P[q], l − 1) + 1
            u ← C[P[q]] + O(P[q], u)
            if l > u then
                break
            end
            q ← q − 1
        end
        if l ≤ u and j − (q + 1) + 1 ≥ min_mem_len then
            start ← q + 1
            end ← j
            Append([l, u], start, end) to M
        end
        else if j − (q + 1) + 1 ≥ min_mem_len then
            start ← q + 1
            end ← j
            Append ([prev_l, prev_u], start, end) to M
        end
    end
End Function
```

The second function that we use the FM index for after counting the suffix array intervals is *locate*. The goal is to take a suffix array index for a *MEM* as input and return its location in the reference T. Since a *MEM* appears I^s times in the reference for each *MEM* we need to calculate I^s positions in T. Algorithm 2 returns the position in the text T of the *MEM* found at suffix array index sa_idx. For the example in Table 3.3, we found only one occurrence of the *MEM* $P = "taa"$ at $sa\_idx = 6$ for which Algorithm 2 returns $SA[6] = 1$ meaning that the *MEM* can be found at index 1 in the original text $T = "ataacg"$.

Memory Considerations for the FM Index

Storing the complete suffix array is not feasible, especially for large sizes of T, which is the case for the human genome. For example, if we were to store the complete suffix array for the human genome of around 3 billion bases with each suffix array entry being 8 bytes in size, as is the case in *BWA-MEM*, then we would end up with a suffix array of around 24GB. As a result, the suffix array is compressed using a compression ratio, an integer that defines which indices should be stored. In the case of *BWA-MEM*, the default compression ratio is 32, meaning only indices divisible by 32 will be kept in the compressed suffix array. The compression ratio appears as SA_INTV in Algorithm 2 and is defined when building the index.

Similarly to the suffix array, we saw that the FM index requires an array that has the occurrences in the BWT of each letter in the alphabet at every index of the BWT. The occurrences array in its complete form can take even more space than the suffix array. Assuming each of the four characters requires a 64-bit counter at each position of the BWT, we can find that a full occurrence array can take around 100GB of space for a 3 billion base genome. The 4 bytes assumption of each counter comes from the fact that in *BWA-MEM*, the occurrence array is packed within the actual BWT array and to save space,

Algorithm 2: Computing the starting position for a given suffix array index taken from [52].

Input: Suffix array index of a seed sa_idx
Output: Position in text T

```
Function LOCATE_SEEDS(sa_idx):
    itr ← 0
    i ← sa_idx
    while i%SA_INTV ≠ 0 do
        i ← C[BWT[i]] + O(BWT[i], i − 1)
        itr ← itr + 1
    end
    return SA[i] + itr
End Function
```

it is only sampled every 128 BWT bases. As the bases in the BWT of *BWA-MEM* are encoded using 2 bits per base, the occurrence counts appear every 256 bits. The result is a reduced occurrences array. On the other hand, the counts are not readily available anymore at every index of the BWT where we are calculating suffix array intervals using previously presented methods. This compressed method involves a bit of searching, as we can demonstrate with a quick example. In Table 3.2, we can assume that the O array is only sampled at multiples of the number 3, so we only have the counts at indexes 0,3 and 6. For example, to calculate $O['a', 4]$, the approach would be to look at the closest available count, which is at index 3, and walk forward one index on the actual BWT and see the letter that appears at $BWT[4]$ which is a $'a'$ so we would conclude that $O['a', 4] = 2$. This approach can be implemented to have forward and backward walks depending on which index is closer so that we traverse less of the BWT counting characters.

Both of the methods mentioned above are employed in *BWA-MEM* as well as other similar applications so that the size of the index is kept relatively low even for larger genomes. However, to make an accurate estimation of the size of the FM index we expect to have for a human genome, we still need to present the actual index utilized in *BWA-MEM*, which has a few additions and is referred to as the FMD index.

FMD Index

DNA has a forward and a reverse strand, making it two-stranded. The forward strand's sense is 5' to 3' end, meaning that it is read from left to right, while the reverse strand is written as 3' to 5' end, meaning that it is the same sequence only reversed. In a double-stranded DNA, each base is paired with its Watson-Crick complementary base. To obtain the reverse complement of a sequence, also known as the Watson–Crick reverse complement, we take our original forward strand and reverse its order. The resulting reversed sequence is then complemented by changing each base with its complement. A quick example of finding the Watson-Crick reverse complement of a DNA string can be seen below.

- **Forward DNA string**: $T = "ataacg"$.
- **Watson-Crick reverse complement**: $\overline{T} = "cgttat"$.

The FM index we presented previously would only index the reference of the forward strand. As a result, searching for seeds in the reverse complement would have to be solved through some algorithmic method. The authors behind *BWA-MEM*, Li et al., came up with a solution to this issue in the paper titled "Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly" [53]. The solution is called the FMD index and is calculated by taking the original string T, finding its Watson-Crick reverse complement $\overline{T}$ and concatenating the two strings to get a new string $T\overline{T}$. Then proceeding to find the FMD index arrays similarly as we saw previously for the FM index. In Figure 3.2 the BWT and suffix array of our new string $T = "ataacgcgttat"$ can be seen while in Figure 3.2 the complete FMD index is shown. From this point forward, we refer as T to the string, which includes the reverse complement concatenated at the end.

i	SA[i]		BWT[i]
0	12	$ ataacgcgtta	t
1	2	aacgcgttat$ a	t
2	3	acgcgttat$ at	a
3	10	at$ ataacgcgt	t
4	0	ataacgcgttat	$
5	4	cgcgttat$ ata	a
6	6	cgttat$ ataac	g
7	5	gcgttat$ ataa	c
8	7	gttat$ ataacg	c
9	11	t$ ataacgcgtt	a
10	1	taacgcgttat$	a
11	9	tat$ ataacgcg	t
12	8	ttat$ ataacgc	g

	SA[i]	BWT[i]	O[a,i]				
i			$	a	c	g	t
0	12	t	0	0	0	0	1
1	2	t	0	0	0	0	2
2	3	a	0	1	0	0	2
3	10	t	0	1	0	0	3
4	0	$	1	1	0	0	3
5	4	a	1	2	0	0	3
6	6	g	1	2	0	1	3
7	5	c	1	2	1	1	3
8	7	c	1	2	2	1	3
9	11	a	1	3	2	1	3
10	1	a	1	4	2	1	3
11	9	t	1	4	2	1	4
12	8	g	1	4	2	2	4
		C[a]	0	1	5	7	9

Figure 3.2: *BWT* and FMD index of $T = "ataacgcgttat"$.

Maximal Exact Matches using the FMD Index

The process of finding *MEMs* using the FMD index becomes a bit different. The suffix array intervals now become bi-intervals in the form of $[I^l(P), I^l(\overline{P}), I^s(P)]$. Assuming that we already have the bi-interval of P we can calculate $I^l(aP)$ and $I^s(aP)$ using Equations 3.7, 3.8 and 3.6. Note that Equations 3.7, 3.8 differ from their counterparts Equations 3.4, 3.5 in the FM index discussion due to the inclusion of the sentinel "$" calculation in count array C and occurrence array O. This change is also reflected in the FMD index in Figure 3.2.

We also know that by definition $[I^l(\overline{aP}), I^u(\overline{aP})]$ is a sub-interval of $[I^l(\overline{P}), I^u(\overline{P})]$ since $\overline{P}$ is a prefix of $\overline{aP}$ [53]. Due to the inherent symmetry induced by the reverse complement, it stands that $I^s(\overline{cP}) = I^s(cP)$ for all $c \in \Sigma$ with $\sum_c I^s(cP) = I^s(\overline{P}) = I^s(P)$ [53]. As a result we can find $I^s(cP)$ for all $c \in \Sigma$ using Equation 3.7. This operation can be seen in lines 2 to 5 of Algorithm 3. Finally, the computed interval sizes can be used to divide $[I^l(\overline{P}), I^u(\overline{P})]$ and find the bi-interval for aP $[I^l(\overline{aP}), I^u(\overline{aP})]$. This operation can be seen in lines 6 to 11 of Algorithm 3 which shows the *backward extension* using the FMD index in order to find the bi-interval of $[I^l(aP), I^l(\overline{aP}), I^s(aP)]$. An important remark is that the FMD index is bidirectional meaning that *backward extension* of P is equivalent to *forward extension* of $\overline{P}$ and *backward extension* of $\overline{P}$ is the same as extending P forward [53]. Algorithm 4 shows the *forward* extension given a sub-interval of a string and a new base.

$$I^l(aP) = C[a] + O(a, I^l(P) - 1) \tag{3.7}$$

$$I^u(aP) = C[a] + O(a, I^u(P)) - 1 \tag{3.8}$$

Using the above-mentioned method we introduce an example where we search *backwards* for the string $P = "taa"$ in the previously presented T. The steps are summarized in Table 3.4. The initialization is done for the last character $"a"$ using the following assumptions:

- For $k_{init} = C("a") + 1$, since we know the first character in the suffix array is always the sentinel and the interval of any character begins at its count plus one.
- For $l_{init} = C(\overline{"a"}) + 1 = C("t") + 1$ is basically the same statement as in the previous point.
- For $s_{init} = C("a" + 1) - C("a") = C("c") - C("a")$, since we know how many occurrences of $"a"$ we expect at the beginning of the suffixes. Due to the symmetry of the FMD index, the number of occurrences at the start of the prefix of the complement will be equal.

			"$"	"a"	"c"	"g"	"t"
Iteration	q	Symbol a	$[k,l,s]$	$[k,l,s]$	$[k,l,s]$	$[k,l,s]$	$[k,l,s]$
Init	2	a	-	$[1,9,4]$	-	-	-
1	1	a	$[0,9,1]$	$[1,12,1]$	$[5,12,0]$	$[7,12,0]$	$[10,10,2]$
3	0	t	$[0,12,0]$	$[1,13,0]$	$[5,13,0]$	$[7,13,0]$	$[10,12,1]$

Table 3.4: FMD Index backward search of P = "taa" in T = "$ataacgcgttat$".

Algorithm 3: Backward extension taken from [53].

Input: Bi-interval $[k,l,s]$ of string W and a symbol a
Output: Bi-interval of string aW

Function `BackwardExt`$([k,l,s],a)$**:**
  for $b \leftarrow 0$ **to** 5 **do**
    $k_b \leftarrow C(b) + O(b,k-1)$
    $s_b \leftarrow O(b,k+s-1) - O(b,k-1)$
  end
  $l_0 \leftarrow l$;
  $l_4 \leftarrow l_0 + s_0$;
  for $b \leftarrow 3$ **to** 1 **do**
    $l_b \leftarrow l_{b+1} + s_{b+1}$
  end
  $l_5 \leftarrow l_1 + s_1$;
  return $[k_a, l_a, s_a]$
End Function

Finding SMEMs with BWA-MEM

In Algorithm 5 we show the main function of *BWA-MEM* which finds the *super maximal exact matches*. As mentioned earlier, *MEMs* are matches that cannot be extended any further, neither in the forward nor in the backward direction. *SMEMs* are the subset of *MEMs* which are not contained in any other *MEMs* on the query. The function in Algorithm 5 takes as input a start position i_0 and the query string P. The initialization is done at base $P[i_0]$ using the same three assumptions mentioned in the FMD index backward search example. The first half of this algorithm looks for *MEMs* in the *forward* direction starting with the complement of base at $i_0 + 1$ from P and continues up to $|P|$. As we mentioned earlier *forward* search is the equivalent of *backward* search with complement base found in the query. The *forward* loop will only exit on two conditions. The first is when the end of the query is reached, meaning that the *MEM* will span from i_0 to $|P| - 1$. The second condition is when for a given base, there are no matches, meaning that $s' = 0$, in which case the *MEM* found with the previous base will be the longest that is found in the *forward* search. The second loop takes the sorted *MEMs*, which were calculated in the *forward* direction, and extends them in the *backward* direction making sure that the position i_0 of the query is also covered toward its beginning at the index 0. It is important to mention that the *backward* and *forward* search functions from Algorithm 5 implement the functionality of their respective functions from Algorithms 3 and 4. The return value of the *SMEM* function is the maximum i_0 in the query, which was covered by the *forward* search. This process is repeated by calling this function again, with the starting position being the previous return value incremented by one until the end of the query is reached (i.e., returned by the function).

Memory Considerations for the FMD Index

During the previous discussion about the size of the FM index, we mentioned that due to the reverse strand, the final index utilized in *BWA-MEM* differs from the basic definition. The size of the complete FMD index for the Reference Consortium Human Build 37 genome (GRCh37) [56] created using the latest baseline *BWA-MEM* 0.7.17 has a size of around 4.7GB. The bases in the BWT are each encoded with 2 bits, while the occurrence interval is every 128 bases. Therefore, each entry in the BWT will be 32 bits, while the occurrence count for each letter is 64 bits (i.e., each taking two 32-bit entries). Similarly, the suffix array is sampled every 32 indexes. Hence, the breakdown calculation of the BWT and suffix array size for GRCh37 will be:

Algorithm 4: Forward extension taken from [53].

Input: Bi-interval $[k, l, s]$ of string W and a symbol a
Output: Bi-interval of string Wa
Function `ForwardExt` $([k, l, s], a)$ **:**
  $[l', k', s'] \leftarrow BackwardExt([l, k, s], \overline{a})$;
  return $[k', l', s']$
End Function

- BWT size: $\frac{3137161264 \times 2 \times 2}{8} + \frac{3137161264 \times 2}{128} \times 4 \times 8 \approx 3.1 \times 10^9$ bytes.
- Suffix array size: $\frac{3137161264 \times 2 \times 64}{32 \times 8} \approx 1.6 \times 10^9$ bytes.

These numbers are only estimations since the C array of the index, which takes around 5×8 bytes, will also be saved within the BWT. The construction of the FMD index is only performed once, and although it can take a few hours, depending on the reference size, it will not be included in any of the performance evaluations.

3.1.2. Chaining and Chain Filtering

Chaining is the next step in the *BWA-MEM* pipeline after finding the seeds between the various reads and the reference. This process refers to creating groups of seeds that lie close to each other on the reference. The formal definition of a chain of seeds, as provided by the authors of *BWA-MEM*, is a group of co-linear seeds that are close to one another on the reference [12]. In Figure 3.3, an illustration of how chains of seeds look is shown. The main goal of the chaining operation is to group the seeds greedily so that in the next step, the shorter chains, which have a large portion of them contained in longer chains, are filtered out [12]. Each chain receives a score based on the number of bases its seeds cover in the reference genome [55]. Furthermore, chains are ordered by their score in descending order, and if a smaller chain has an overlap of over 50% and 38bp less than a higher score seed, then it is discarded during filtering. Finally, during alignment, the chains with higher scores are aligned first so that if the chain is extended enough to cover a smaller chain, then the smaller chain does not require alignment.

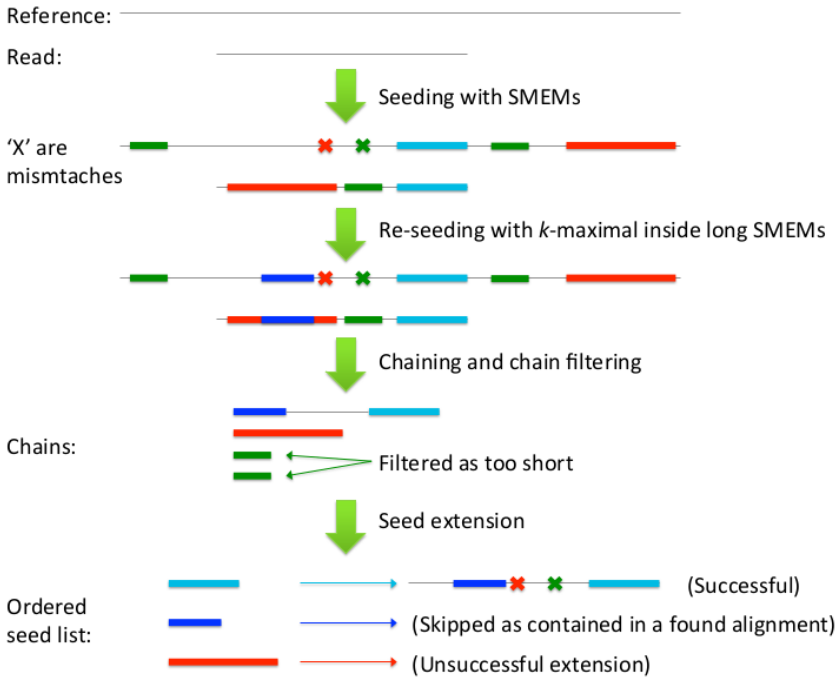


Figure 3.3: Seed chaining and chain filtering illustration taken from [57].

Algorithm 5: Finding SMEMs taken from [53].

Input: String P and start position i_0; $P[-1] = 0$
Output: Set of bi-intervals of SMEMs overlapping i_0

Function `SuperMEM1(`P, i_0`)`**:**
  Initialize **Curr**, **Prev** and **Match** as empty arrays
  $[k, l, s] \leftarrow [C(P[i_0]), C(\overline{P[i_0]}), C(P[i_0]+1) - C(P[i_0])]$;
  for $i_0 \leftarrow i_0 + 1$ **to** $|P|$ **do**
    if $i = |P|$ **then**
      $Append$ $[k, l, s]$ to **Curr**
    end
    else
      $[k', l', s'] \leftarrow ForwardExt([k, l, s], P[i])$;
      if $s' \neq s$ **then**
        $Append$ $[k, l, s]$ to **Curr**
      end
      if *s' = 0* **then**
        break;
      end
      $[k, l, s] \leftarrow [k', l', s']$
    end
  end
  $Swap$ **Curr** and **Prev**;
  $i' \leftarrow |P|$
  for $i_0 \leftarrow i_0 - 1$ **to** -1 **do**
    Reset **Curr** to empty
    $s'' \leftarrow -1$;
    for $[k, l, s]$ ***in*** $Prev$ **do**
      $[k', l', s'] \leftarrow BackwardExt([k, l, s], P[i])$;
      if $s' = 0$ ***or*** $i = -1$ **then**
        if ***Curr*** *is empty* ***and*** $i + 1 < i + 1$ **then**
          $i' \leftarrow i$;
          $Append$ $[k, l, s]$ to **Match**
        end
      end
      if $s' \neq 0$ ***and*** $s' \neq s''$ **then**
        $s'' \leftarrow s'$;
        $Append$ $[k, l, s]$ to **Curr**
      end
    end
    if ***Curr*** *is empty* **then**
      break
    end
    $Swap$ **Curr** and **Prev**;
  end
  return Match
End Function

3.1.3. Seed Extension

Local Alignment with Smith-Waterman

After completing the seeding and chaining steps, we move to align the seeds onto the reference and extend them on both ends so that the best possible score for local alignment is found. *BWA-MEM* performs local alignment between two sequences utilizing a modified version of the *Smith-Waterman* dynamic programming (DP) algorithm.

First, we present some definitions regarding how the scoring system works in such a dynamic programming method. If the two bases being compared between the query P and reference T are the same, then we say we have a match and define a positive score. On the other hand, in the case of a mismatch, we define a negative mismatch score. Furthermore, we would also like a mechanism that penalizes gaps between DNA sequences. As we saw in the previous chapter, insertions or deletions (i.e., *indels*) are very common in DNA strings and are one of the forms in which mutations appear. Therefore we would like to be able to reflect the *indels* into our alignment scoring system, and we do this by defining an affine gap penalty [58]. In the affine gap penalty model, we penalize a gap opening more than we punish a gap extension. As a result, using such a model enables us to continue aligning even areas with *indels* and to have a scoring system. There is also the possibility of a linear gap penalty system that simply penalizes both creation and extension of a gap with the same negative score. In the case of DNA sequences where there is a much higher probability of having a larger gap rather than many shorter ones, an affine penalty gap system gives better results since it penalizes initial gap opening more than extending a gap [37].

The dynamic programming computation of the *Smith-Waterman* local alignment is an iterative process where we calculate all of the cells in a matrix of size $|P||T|$ using the previously presented scoring system. The convention is to arrange the query along a column while the reference is along the first row, as seen in Figure 3.4. Equation 3.9 presents the value that the similarity matrix at position $S[i,j]$ will receive based on the values of three of its neighboring cells. The top calculation in Equation 3.9 uses the north-western cell's (i.e., left diagonal) score added to the positive match score α to calculate the current cell score in case the two bases are the same. Inversely when the two bases being compared do not match, the score is reduced due to the addition of the negative value β. The third calculation, which can be seen in detail in Equation 3.11, is used to penalize a gap opening in the query sequence with the negative constant g. In Equation 3.11, we also see the penalty of just widening a gap which is the negative constant h for which we have that $|h| < |g|$ since we penalize less widening a gap than opening it. Finally, Equation 3.12 shows the same affine gap penalty for the case in which we create or widen a gap in the reference. After completing the calculation of the complete S matrix, we trace back starting from the overall maximum value found in S moving backward.

In Figure 3.4 we show the S matrix calculation between a reference T and a query P. In this example the match score $\alpha = 5$ while the mismatch score $\beta = -3$. In addition, we utilize a constant gap penalty $h = -4$. The blue cells in Figure 3.4 show the traceback process. Based on the movement we make when we trace back the maximum values, we can make the following remarks [58]:

- A horizontal move towards $S[i,j]$ is the same as inserting a gap in the query after $P[i]$.
- A vertical move towards $S[i,j]$ is the same as inserting a gap in the reference after $T[j]$.
- A diagonal move towards $S[i,j]$ is the same as aligning $P[i]$ and $T[j]$.

It is important to note that in the traceback process for the example we just presented, another optimal alignment starts at the last row of S at the other value, 18.

$$S[i,j] = max\begin{cases} S[i-1,j-1] + \delta(P[i],T[j]) \\ G_A[i,j] \\ G_B[i,j] \\ 0 \end{cases} \tag{3.9}$$

$$\delta(x,y) = \begin{cases} \alpha & \text{if bases } x = y \\ \beta & \text{if bases } x \neq y \end{cases} \tag{3.10}$$

$$G_A[i,j] = max\begin{cases} S[i-1,j] + g + h \\ G_A[i-1,j] + h \end{cases} \tag{3.11}$$

$$G_B[i,j] = max\begin{cases} S[i,j-1] + g + h \\ G_B[i,j-1] + h \end{cases} \tag{3.12}$$

In Algorithm 6, we present the pseudocode for the local alignment using affine gap penalties adapted from [59]. Algorithm 6 shows the looping strategy required to calculate the similarity matrix S using Equations 3.9 to 3.12.

		c	g	t	g	a	a	t	t	c	a	t
	0	0	0	0	0	0	0	0	0	0	0	0
g	0	0	5	1	5	1	0	0	0	0	0	0
a	0	0	1	2	1	10	6	2	0	0	5	1
c	0	5	1	0	0	6	7	3	0	5	1	2
t	0	1	2	6	2	2	3	12	8	4	2	6
t	0	0	0	7	3	0	0	8	17	13	9	7
a	0	0	0	3	4	8	5	4	13	14	18	14
c	0	5	1	0	0	4	5	2	9	18	14	15

Figure 3.4: Local Alignment score matrix S with traceback.

BWA-MEM Extension

The actual implementation of the seed extension in *BWA-MEM* differs from the naive *Smith-Waterman* method we introduced in the previous subsection. In this part, we present some key differences between the algorithm *BWA-MEM* and the previously presented local alignment method. First, we look at how *BWA-MEM* loops over the chains to align one seed at a time. In Algorithm 7 we present the pseudocode for the seed extension phase of *BWA-MEM* taken from [15]. The algorithm has an outer loop that iterates over all the chains in a read, starting with the one with the highest score. The inner loop will iterate over each seed in the chain, from the seed with the highest score to the lowest rated seed. The score of each seed is based on the chain it belongs to, and its length [12]. For each seed in the list, there are two possibilities; the first is to have a highly overlapping region with an already performed alignment meaning that it can be discarded without performing an alignment. The second possibility is that the seed can lead to a better alignment than a previous one or that the specific region has not been aligned. In this case, the seed is extended first towards the left and then towards the right using the inexact match functions, which perform in an almost *Smith-Waterman*-like fashion. The left extension will be skipped if the seed starts at the start of the query, or the right extension will be skipped when the seed reaches the end of the query. Finally, we present the main differences of *BWA-MEM* employed during seed extension:

- *BWA-MEM* extends seeds with a banded affine gap penalty dynamic programming method [12]. We already introduced what affine gap penalty means and how the dynamic programming method is applied, but banded is another heuristic used to reduce computations and execution time [58]. The banded alignment assumes that during the calculation of the similarity matrix S, the computation will not wander off the main diagonal or at least not too far from it. The main reason behind this assumption is that the alignment in *BWA-MEM* is done between two highly similar DNA regions due to seeding. As a result, the alignment will move on the main diagonal, which is the case when there are no *indels* that are long.

- *BWA-MEM* will stop extending if it reaches a certain i position in the reference and j in the query, and the alignment score at (i, j) drops by $Z + |i - j| \times \alpha$ below the best score so far [12]. This heuristic is called *Z-dropoff* and avoids the extension over poorly aligned regions with good alignments on the sides of that region [12]. The *Z-dropoff* alleviates the issue introduced by banded DP where the algorithm might continue aligning over a low score region in the case the flanking regions have good scores.

- During seed extension *BWA-MEM* also keeps track of the maximum alignment score reaching the end of the query sequence [12]. When the difference between the best end-to-end and the best local alignment scores falls below a specific value, known as the clipping value, the software will keep only the end-to-end alignment. As a result, *BWA-MEM* can create end-to-end mappings

Algorithm 6: Dynamic programming matrix computation for local alignment adapted from [59].

Input: query_string, target_string, query_length, taget_length, α, β, g, h
Output: end_position_query, end_position_target, score

Function LOCAL_ALIGNMENT(*query_string, target_string,* α, β, g, h)**:**
  for $i \leftarrow -1$ **to** *query_length* **do**
    $S_{i,-1} \leftarrow 0$
  end
  for $j \leftarrow -1$ **to** *target_length* **do**
    $S_{-1,j} \leftarrow 0$
  end
  for $i \leftarrow 0$ **to** *query_length - 1* **do**
    for $j \leftarrow 0$ **to** *target_length - 1* **do**
      if $query\_base[i] = target\_base[j]$ **then**
        $S^1[i,j] \leftarrow S[i-1, j-1] + \alpha$
      end
      else
        $S^2[i,j] \leftarrow S[i-1, j-1] + \beta$
      end
      $G_A[i,j] \leftarrow max(S[i-1,j] + g + h,\ G_A[i-1,j] + h)$
      $G_B[i,j] \leftarrow max(S[i,j-1] + g + h,\ G_B[i,j-1] + h)$
      $S[i,j] \leftarrow max(S^1[i,j],\ S^2[i,j],\ G_A[i,j],\ G_B[i,j],\ 0)$
    end
  end
  Find i_{max} and j_{max} for which $S[i_{max}, j_{max}] = max(S[i,j])$
  score $\leftarrow S_{i_{max}, j_{max}}$
  end_position_query $\leftarrow i_{max}$
  end_position_target $\leftarrow j_{max}$
End Function

without creating long gaps or mismatches, which is the case when global alignment is induced on a read with long variations [12].

- During the extension in the left direction, the initialization score of the similarity matrix in the first step is not with 0, as we saw in the naive dynamic programming example but with the score (i.e., length) of the seed being extended. Similarly, during the right extension, the initialization is done using the score of the left extension.

3.1.4. BWA-MEM Output

The output of the *BWA-MEM* pipeline is the alignments in the *SAM* format which stands for *Sequence Alignment/Map* format. The *SAM* file format displays the alignment information for all the sequences aligned against a reference. A regular *SAM* file consists of two sections, namely, an optional header section and an alignments section. The header section is found at the beginning of the file and is identified by the lines that start with an ”@” symbol. The first lines in the header section begin with ”@SQ”, meaning that the program identified some sequence contigs in the reference file. By contigs, we refer to contiguous segments of DNA that overlap and, when assembled, form a specific region of a genome. In Figure 3.5, we see an example of *SAM* format output where the program correctly identified chromosome 21 as a contig and also displayed the number of bases it contains, which is shown after ”LN”. The second type of line in the header section starts with ”@PG” and provides information about the program that performed the alignment and the command used to execute the program.

The most important information that can be found in a *SAM* file appears in the alignments section, where the output of the alignment is displayed. In Figure 3.5, we present a simple example where there is only one query aligned against chromosome 21 of the human genome, and the result is displayed in the last three lines of the image. In Table 3.5 we present each tab-delimited field in the order it appears in Figure 3.5.

Algorithm 7: BWA-MEM Seed Extension Pseudo Code recreated from [15].

```
Input: List of Chains of Seeds
Output: List of Alignment Regions
Function SEED_EXTENSION(List of Chains of Seeds):
    for each Chain of Seeds do
        sort Seeds based on their length
        for each seed do
            if (no overlap exists between current Seed and
            previously found Alignment Regions) then
                perform Inexact Matching Left
                perform Inexact Matching Right
                store Alignment Region
            end
        end
    end
End Function
```

```
@SQ     SN:chr21        LN:48129895
@PG     ID:bwa  PN:bwa  VN:0.1  CL:./bwa-gasal2 gase_aln -t 1 -l 150 chr21.fa ../../fasta/Simple/rt1.fasta
[M::process] read 1 sequences (151 bp)...
[M::mem_process_seqs] Processed 1 reads in 0.379 CPU sec, 0.379 real sec
SRR949537.1     0       chr21   48119790        39      23M1D14M1I6M1I35M2I27M42S       *       0       0       TAGGGTTAGGGTTAGGGTTAGGGT
AGGGTTAGGGTTAGGGGTTAGGGGTTGGGGTTAGGGTTAGGGTTAGGGTAGGGGTTAGGGGTTAGGGTTAGGGTTAGGGTTAGGGTTAGATCGGAAGAGCACACGTCTGAACTCCAGTCACCTACCA *
NM:i:8  MD:Z:23^T25A22T0A32     AS:i:61 XS:i:38
```

Figure 3.5: *SAM* format example.

From the fields presented in Table 3.5, we are particularly interested in the mapping quality (i.e., *MAPQ*) since it provides a probability estimation of whether that particular query maps to more locations in the reference. Additionally, the *CIGAR* string provides a compressed representation of how the query was aligned to the reference since each letter provides information on whether it was a match, a deletion, etc. Table 3.6 summarizes all the different operations of a *CIGAR* string. For example, part of the *CIGAR* string obtained in Figure 3.5 was $23M1D14M1I$ which translates to 23 bases matched, one base deleted (i.e., a gap in the reference), 14 matches and one insertion in the query sequence.

3.2. GPUSeed

In the section, we will present *GPUSeed*, a program that finds the seeds between queries and a reference using GPU acceleration. The program was first created by Nauman Ahmed and was published in [52]. *GPUSeed* has as a goal to emulate the process of finding *(S)MEMs* of *BWA-MEM* while performing much faster on a GPU.

As we mentioned earlier, the independent nature of the reads and the high number of reads make seeding an excellent candidate for acceleration since there are no dependencies among them, and they can efficiently use the resources of a GPU. The following sections will look at all the stages of the initial *GPUSeed* implementation that is publicly available on GitHub [61]. Also, we will discuss similarities and differences between the processes of *GPUSeed* and *BWA-MEM*.

3.2.1. GPUSeed Index

In the previous sections, we presented the idea of using an index to reduce the time complexity of searching for matches between two strings. *GPUSeed* uses the FM index to search on both the reference T and its complement $\overline{T}$, without utilizing the FMD index discussed earlier. The FM index of *GPUSeed* includes the following fields:

- **BWT**: transform array.
- **C**: count array
- **SA**: suffix array.

Col	Field	Type	Regexp/Range	Brief Description
1	**QNAME**	String	[!-?A- ~]{1,254}	Query template NAME
2	**FLAG**	Int	$[0, 2^{16} - 1]$	bitwise FLAG
3	**RNAME**	String	*|[:rname:\^*=][:rname:]*	Reference sequence NAME
4	**POS**	Int	$[0, 2^{31} - 1]$	1-based leftmost mapping POSition
5	**MAPQ**	Int	$[0, 2^{8} - 1]$	MAPping Quality
6	**CIGAR**	String	*|([0-9]+[MIDNSHPX=])+	CIGAR string
7	**RNEXT**	String	*|=|[:rname:\^*=][:rname:]*	Reference name of the mate/next read
8	**PNEXT**	Int	$[0, 2^{31} - 1]$	Position of the mate/next read
9	**TLEN**	Int	$[-2^{31} + 1, 2^{31} - 1]$	observed Template LENgth
10	**SEQ**	String	*|[A-Za-z=.]+	segment SEQuence
11	**QUAL**	String	[!-~]+	ASCII of Phred-scaled base QUALity+33

Table 3.5: *SAM* format specification recreated form [60].

Op	BAM	Description	Consumes query	Consumes reference
M	0	alignment match (can be a sequence match or mismatch)	yes	yes
I	1	insertion to the reference	yes	no
D	2	deletion from the reference	no	yes
N	3	skipped region from the reference	no	yes
S	4	soft clipping (clipped sequences present in SEQ)	yes	no
H	5	hard clipping (clipped sequences NOT present in SEQ)	no	no
P	6	padding (silent deletion from padded reference)	no	no
$=$	7	sequence match	yes	yes
X	8	sequence mismatch	yes	yes

Table 3.6: *CIGAR* string operations recreated from [60].

- **O**: occurrence array.

Using the checkpoint method discussed during the FM index presentation, the occurrence array is integrated within the *BWT* array. Instead of saving the complete *O* array, the values of the occurrences of each letter in the *BWT* are saved every 64 bases. Since each entry in the *BWT* is 32 bits, and each base is encoded using 2 bits, we will have four entries of 16 bases each in the *BWT* followed by an occurrence checkpoint which again has four entries of 32 bits for each letter. In Figure 3.6, the structure of the compressed BWT array can be seen.

3.2.2. Stage 0: Pre-processing

The first stage in the *GPUSeed* pipeline aims at preparing the query data and generating four arrays that are required for calculating the *(S)MEMs* intervals in the next phase [52]. *(S)MEMs* on *GPUSeed* are found for batches of queries, and the batching scheme is to take the queries corresponding to 1 million bases at a time. It is essential to mention that the batching method will not split a read between two different batches, and there is a tolerance to include a few more bases than 1 million so that the last read in the batch is complete. The batch of bases is loaded from the file into an array where the different reads are packed together. As a result, an offset array is required to distinguish between different reads. These two arrays containing a batch of reads and their offsets are loaded into the GPU global memory.

The next step in pre-processing is to take the array of reads and convert each base from ASCII into a 4-bit representation [52]. The resulting bases are packed into a 32-bit integer array, with each array entry holding eight bases. The conversion into 4-bit and the packing are performed on the GPU by the same kernel. After packing the bases of that specific batch, a kernel called `prepare_batch` will assign a GPU thread to each pattern in the packed array. The assignment will be done based on the maximum number of *(S)MEMs* that can be produced from that read which is $|P| - min\_seed\_len + 1$. Finally, the last array created during pre-processing is an array that assigns a possible *MEM* from a query to a GPU thread. An example of GPU thread assignment can be seen in Figure 3.7.

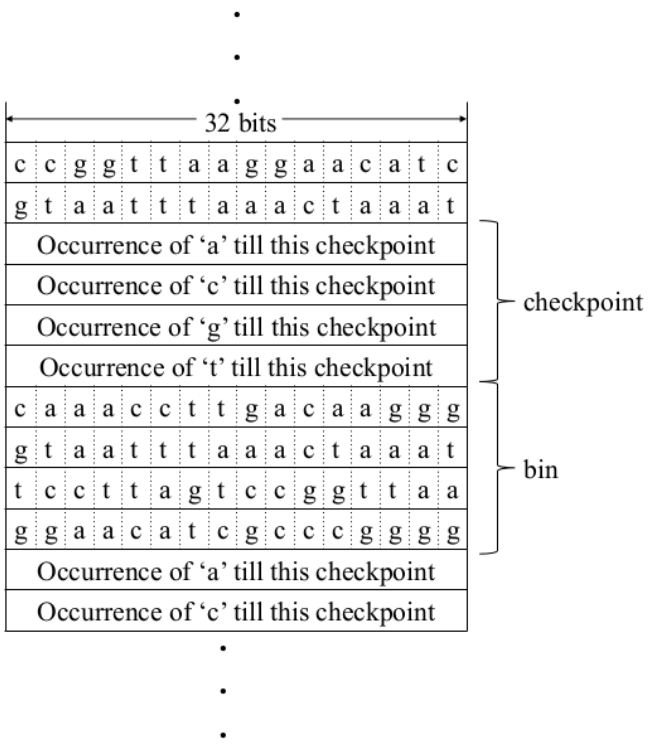


Figure 3.6: Structure of *BWT* used by GPUSeed taken from [55].

3.2.3. Stage 1: Finding Suffix Array Intervals

This stage of the program computes the suffix array intervals for all the different *MEMs* of each query utilizing the kernel `find_seed_intervals_gpu` [52]. Each GPU thread will start at its assigned position and extend backward until it either reaches the start of the read or until the bases do not match. For example, in Figure 3.7, the thread assignment can be seen for a minimum seed length of 6; in this scenario, pattern 0 will have 13 possible *MEMs*, threads 0 to 12 will each calculate a *MEMs* starting from increasingly closer positions to the 0 index of the read. The method in which the kernel extends in the backward direction is the same as we presented in Algorithm 1 with the main difference being that the *for* loop is not required since each thread will start at a different position in its respective pattern P. Additionally, *GPUSeed* has two optimizations applied in order to speed up the process of finding suffix array intervals:

1. **Pre-calculate suffix array intervals**: *GPUSeed* will pre-calculate all possible suffix array intervals for sequences of length pre_calc_len, which are $4^{pre\_calc\_len}$ intervals. This default sequence length is 13. This optimization reduces the time required for backward search since the suffix array interval for the last 13 bases in any *MEM* are known beforehand. This optimization aims to reduce the steps needed for backward search and limit the costly memory accesses to the BWT, especially during the initial steps of the backward search when the suffix array intervals are large due to the small number of bases.

2. **Early detection of redundant *MEMs***: refers to the detection of a *MEM* contained in a larger one while executing the `find_seed_intervals_gpu` kernel. The main idea is that if two threads working on the same pattern reach the same index in the pattern and have the same suffix array interval, then the thread with the higher ID is computing a *MEM* which is already contained in the *MEM* of the thread with the smaller ID. In order to be able to compare intervals among different threads, warp shuffle instructions are utilized.

The end result of the `find_seed_intervals_gpu` kernel are seeds in the form of $[l, u]$ suffix array intervals and $start \rightarrow end$ positions in the query. Since warps only include 32 threads, not all threads working on a pattern will be able to exchange information, and we will need additional filtering.

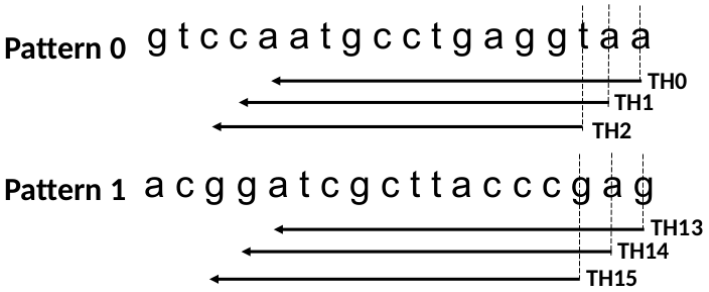


Figure 3.7: GPU thread assignment for MEM computation $min\_seed\_len = 6$ taken from [55].

3.2.4. Stage 2: Filtering (S)MEMs

The filtering stage takes the output of the previous step, which are suffix array intervals saved as one two-dimensional vector (i.e., $uint2$) and one two-dimensional vector (i.e., $int2$) holding the starting and ending positions of all the seeds for all the batch queries and starts by removing all the null entries [52]. These entries are removed using the *CUB* library and, specifically, a function that selects only flagged entries. The flag array is set during stage 1 every time a seed is found to be valid (i.e., not null), and the size of the flag array is equal to the arrays containing the suffix array intervals and the query positions. Then, the primary filtering kernel called `filter_seed_intervals_gpu` will start a GPU thread for each of the seeds in the batch, which will check whether the following conditions evaluate to true:

1. **Condition 1**: $start[i] = start[i-1]$.

2. **Condition 2**: $u[i] - l[i] + 1 = u[i-1] - l[i-1] + 1$.

If both conditions are true, then the *MEM* at $i-1$ will be marked as redundant. Therefore, only the first condition is checked when searching for *SMEMs*. It is essential to mention that since all seeds are in the same array, these comparisons need to happen in the segments marked by the starting and ending positions of each query's set of seeds. This segmentation is achieved using an offset array containing at each read's index the total number of preceding seeds. The final output of this stage is similar to stage 1, with the only difference being that there are fewer seeds.

3.2.5. Stage 3: Split Suffix Array Intervals

In order to locate the positions of the seeds in the reference T, it is required to split the suffix array intervals into individual indexes [52]. The kernel `seeds_to_threads` assigns one GPU thread per suffix array interval which is then split into $l - u + 1$ indexes. It is important in this stage to filter out smaller seeds with the same suffix array index, so *MEMs* which are contained in larger *MEMs* and appear at the same position in the text are excluded. The output of this stage will be in the format $start \rightarrow end$ attached to a suffix array index sa_idx for each seed.

3.2.6. Stage 4: Locate (S)MEMs

The final stage utilizes a similar method to Algorithm 1 to find the final position of a seed in the reference given a suffix array index. The GPU implementation uses a kernel named `locate_seeds_gpu`, which launches one GPU thread for each seed.

3.2.7. GPUSeed Discussion

Finally, it is important to mention that *GPUSeed* can also find seeds for the reverse complement of a pattern $\overline{P}$ [52]. During the packing stage, a kernel finds the reverse complement of each read and then packs it using the same 4-bit method we presented previously. The seeds found using these reverse complemented reads will appear after the forward seeds of each read. Also, the filtering stage includes a filter module that checks the y component of each reverse seed and filters it similarly, as we presented in stage 2.

3.3. GASAL2

The final library which we will present is called *GASAL2* and was also created by Nauman Ahmed and published in [35]. This library is also freely available on GitHub [62]. The goal of *GASAL2* is to perform pairwise sequence alignments on a GPU. The library integrates local, global and semi-global alignment methods letting the user choose the type of alignment they would like to perform. In the following sections, we will present this library's main features as well as some modifications and additions previously implemented by a master's student who extended the capabilities of *GASAL2* to perform *BWA-MEM* like extensions. Since the focus of this report falls on local alignment using the *Smith-Waterman* extension, we will primarily look at the functions of *GASAL2* associated with the extension that tries to replicate the behavior of *BWA-MEM*.

3.3.1. Stage-0: Fill Parameters

During this preparation stage, the program will load all the settings the user provides for the alignment and information about the reference and the queries. The settings include values associated with alignment scoring and parameters such as *Z-Dropoff* [59]. Furthermore, CPU and GPU storage is allocated based on the maximum query length passed at compilation time and some predefined coefficients that aim to predict the number of seeds that will come out of the seeding phase. The main idea behind this pre-allocation is to separate memory allocations from the computation phase so that the workflow can be pipelined [59]. After completing the memory allocation, the program will choose the first available GPU stream and start transferring the first batch of queries and references from the CPU to the GPU. In Figure 3.8, a simplified flow chart shows the complete workflow of *GASAL2* where these steps from stage 0 can be seen.

3.3.2. Stage-1: Packing DNA Sequences

The first step in the *GASAL2* execution pipeline is data packing which is done similarly to *GPUSeed*. The difference, in this case, is that the reference also needs to be packed, while in the case of *GPUSeed*, only the reads require packing. Next, the packing kernel takes bases, initially represented by a byte, and transforms them into a 4-bit representation [35]. Again, 3 bits would have been enough to represent all five bases, but 4 bits are much more convenient and easier to use, especially when dealing with GPUs which have 32-bit registers where the data is loaded. Next, the packing kernel provides one GPU thread for packing 8 bases, which are then saved into an unsigned 32-bit integer array residing on the GPU global memory [35].

3.3.3. Stage-2 (optional): Reverse Complement Kernel

In the scenario the user would like to align the reverse and/or complement of any sequence, *GASAL2* provides the ability to perform this operation directly on the GPU. Each sequence receives a flag that will tell the kernel whether it is required to reverse and/or complement it. This step is optional, and the kernel works directly on the packed data, which is in the GPU memory.

3.3.4. Stage-3: Local Alignment Kernel

The local alignment kernel was implemented in [59] by a master's student to perform seed extension in a manner that resembles as much as possible *BWA-MEM*. First of all, the kernel that performs the alignment has to follow the method of *BWA-MEM* which computes an alignment towards the left side of the seed starting with a score equal to the length of the seed and then performs a right alignment using the score from the left alignment. In the case of the GPU implementation, this would not be possible without calling the left and right alignments sequentially, meaning that a sequential part would limit acceleration. As a result, the developers decided to start the two alignments concurrently with the right alignment, also starting with the seed score.

The main kernel that performs the alignment on the GPU is very similar to the method used in *BWA-MEM* and was presented in Algorithm 6. Each GPU thread is assigned one alignment, and the actual looping methodology for an alignment is slightly changed from Algorithm 6. The main differences appear due to the packing of the query and reference sequence from stage 1. The idea is that if there were only two loops, as we saw in Algorithm 6 and we were to calculate the alignment row by row, then we would always have to load a query base and a reference base from memory at every index. In addition, for each new cell, the values of the cell on the left, the cell on top and the cell on the top-

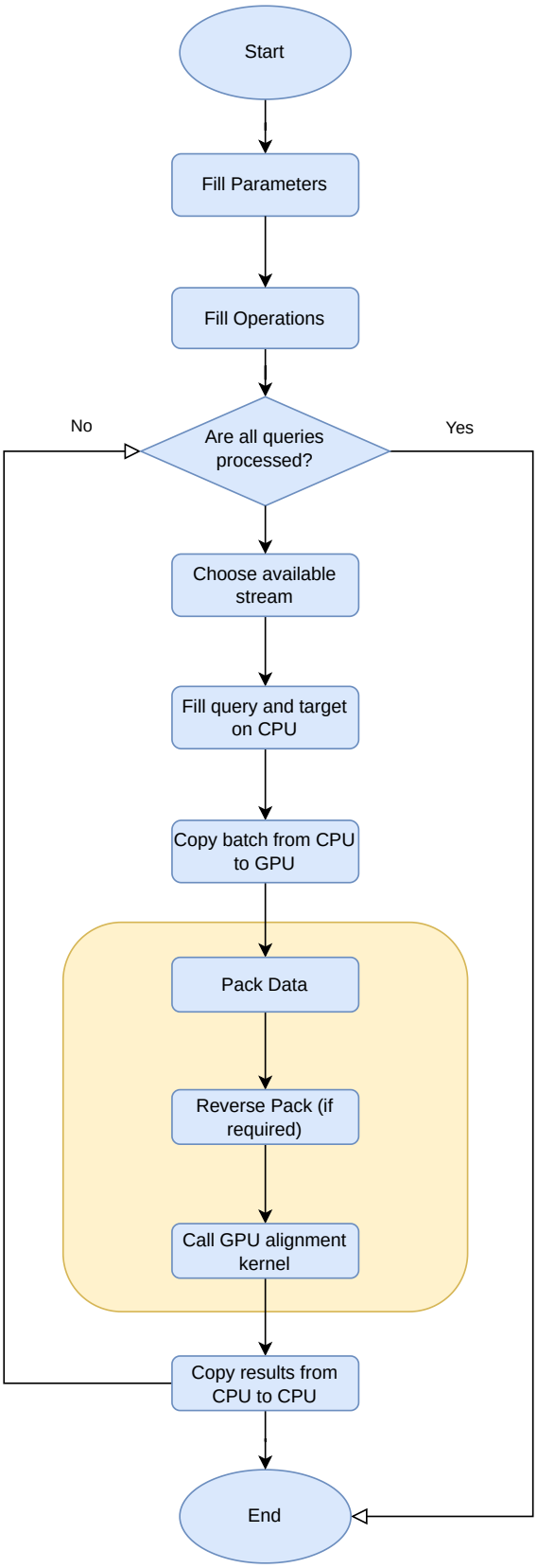


Figure 3.8: *GASAL2* workflow adapted from [59].

left diagonal are required, and again probably for the last two values, unnecessary memory accesses would be required. As a result, the developers utilized a method to compute the alignment scores on tiles of 8 by 8 bases [35]. In a nutshell, the main idea is always to have a column of 8 scores to the left of the tile being computed and to also keep the entire row of scores above the 8 by 8 tile being computed. In Figure 3.9, an illustration is presented where the orange column and the complete yellow row of cells are available for the computation of the cells with numbers 1 to 8. Once cells 1 to 8 are computed, the orange column moves to the right one step to be utilized for the computation of cells marked 9 to 16. After calculating all 8 by 8 tiles in the row, the dark blue tile in Figure 3.9, the yellow line takes the row that includes cells 8,16, etc., since those values will be required for the next row of tiles.

This implementation drastically reduces the memory accesses required for the computation of 64 cells since every tile involves a set of 8 query bases and 8 reference bases, each packed in one 32-bit word [35]. Also, the intermediate results can be kept on the registers of the GPU since only one column of 8 scores and one whole row the size of the query is required.

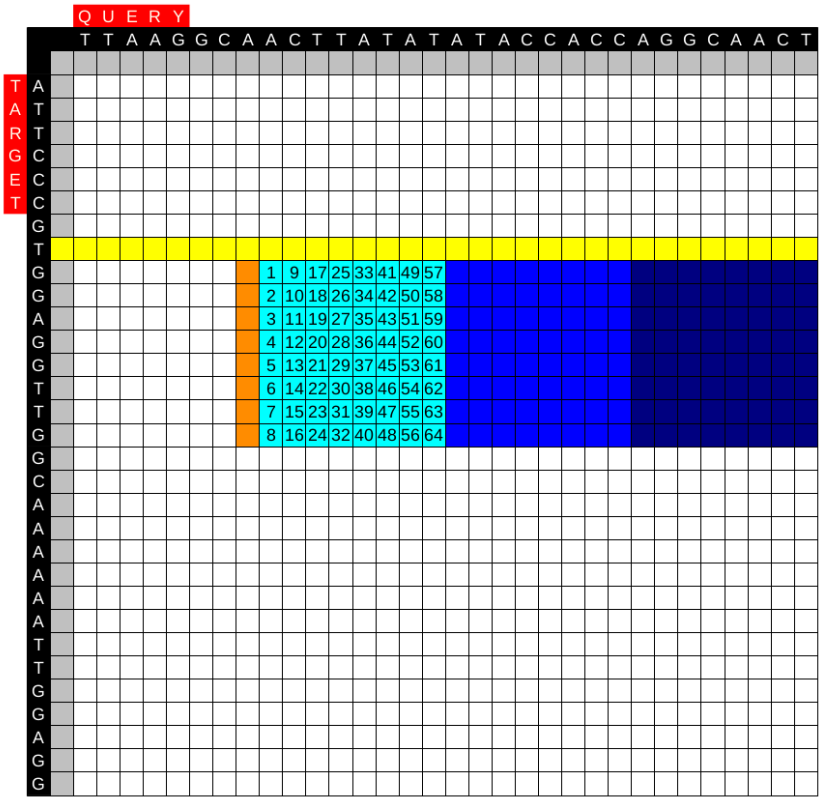


Figure 3.9: *GASAL2* tiling taken from [59].

3.3.5. GASAL2 Features

Finally, in this part, we present some of the features of *GASAL2*. First, the packing kernel and the extension kernel are non-blocking, meaning the CPU execution can continue after the kernel invocation. For example, if we look at Figure 3.8 of the workflow, we can already notice that if the alignment kernel is not finished, the host can continue executing other tasks. This capability is particularly interesting in the context of *BWA-MEM*, where seeding and chaining can be performed for the next batch of data. Additionally, *GASAL2* uses two GPU streams per host thread, enabling parallel execution of kernels and the overlap of kernel and GPU memory transfers.

GASAL2 also implements several optimizations, especially in how it splits the seeds for alignment. In order to not have divergent GPU threads executing the alignment, it batches seeds depending on whether they have to perform a long or a shorter extension based on their position in the query. Shorter batches will launch on the same kernel, while longer ones will be in a different kernel. This optimization

improves warp scheduling by having more similar branching within the same warp, but the left and right alignment of the same seed might end up in different batches. This added complexity requires a kind of monitoring system which keeps track of how the seed was aligned and how the final score of the alignment should be computed. The monitoring system works on each seed which will receive a 0 if no alignments were done, a 1 if it was extended only in one direction, or a 2 if both sides were extended. As a result, if the alignment was done on both sides in different batches, the seed score will have to be subtracted from the final alignment score as both left and right alignments include the seed score as their respective starting score [59]. Also, during the stage 0 discussion, we briefly mentioned that the program pre-allocates memory based on an expectation of how many alignments it expects to perform and an estimate of how many bases will be extended. What was not mentioned is the ability to extend these data fields on-the-fly every time an alignment is longer than expected, or more seeds appear for a read [59]. This feature should be avoided as much as possible since on-the-fly memory reallocation slows the computation.

4

Analysis

In this chapter, we aim to expand the knowledge of the three software components first presented in Chapter 3 by analyzing their respective pipelines and performance. The chapter is divided into three parts, with the first part looking at *BWA-MEM* while the second and third sections look at *GPUSeed* and *GASAL2* respectively. In each section, we will see what a typical program run looks like in terms of function calls and create a profile showing the percentages taken by the most critical functions. Based on this analysis, the goal is to clearly understand how to integrate the three libraries moving forward into the integration chapter.

The hardware platform used for our experiments is a dual-socket configuration with two Intel Xeon Processors E5-2620 v3 [63], each equipped with six physical cores and 12 threads running at 2.4GHz. In addition, the CPUs are paired with 128GB DDR4 memory and 1TB of local storage. For the profiling of the applications which have *CUDA* kernels, we will utilize the CPU platform mentioned above in combination with an NVIDIA RTX 2080Ti with 11GB of VRAM, and 68 SMs [64].

For our applications' profiling, we created simulated reads using a tool named *wgsim* which can be found on GitHub [65]. This tool enables the creation of simulated reads from a reference genome. The method of simulated reads allows us to create multiple datasets from the same reference and keep the rate of mutations and the fraction of *INDELs* at their default constants while altering only the query length. We used the Genome Reference Consortium Human Build 37 (GRCh37) [56] to create 10 million reads with varying bases per read. Due to the focus of *BWA-MEM* on NGS data, we considered that a minimum length of 100 bases per query and a maximum of 250 bases was reasonable. As a result, there are four datasets of synthetic reads, each with 10 million queries and 100, 150, 200 and 250 bases per query. Furthermore, two real datasets, SRR921889 [66] and SRR835433 [67], are also included so that the profiling results are verified with real data. In the context of the profiling study, we take the first 8.3 million queries from the SRR921889 dataset, while from the SRR835433 dataset, we take the first read from each spot resulting in 8.3 million reads. The complete dataset used for the profiling study can be seen in Table 4.1. In Table 4.1, we denote the modified real datasets with "_1" to separate them from the complete datasets. For the profiling of all the applications in this section, we use a minimum seed length of 19, and we map our queries of Table 4.1 against the human genome GRCh37.

Query Dataset	Total Reads	Bases/Query
SRR921889_1	8.3M	100
Simread_100	10M	100
Simread_150	10M	150
Simread_200	10M	200
Simread_250	10M	250
SRR835433_1	8.3M	250

Table 4.1: Profiling Dataset.

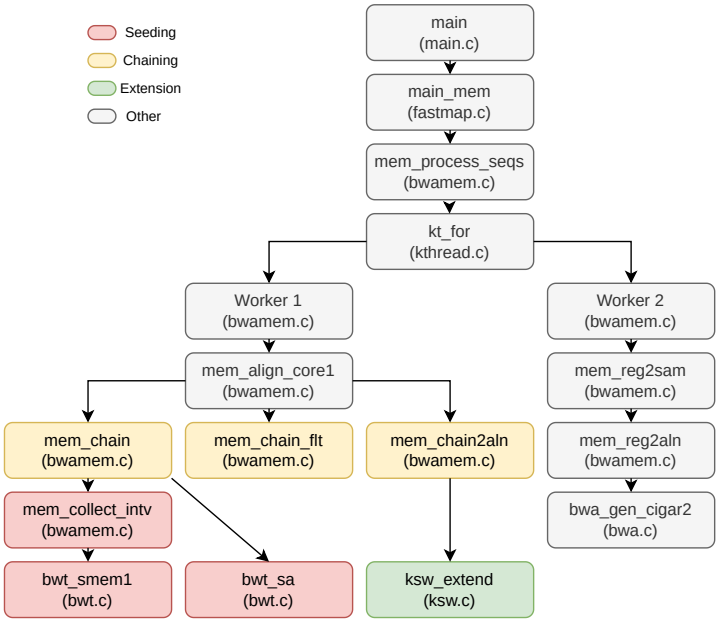


Figure 4.1: *BWA-MEM* call graph.

4.1. Burrows-Wheeler Aligner Analysis (BWA-MEM)

In this section, we discuss the profile of the baseline *BWA-MEM* function of the latest Burrows-Wheeler Aligner, version 0.7.17. In order to profile the application, we use the *gprof* profiling tool, which enables us to see all the various function calls and processes initialized by *BWA-MEM*. The profiling tool provides two types of information at its output, with the first being a flat profile of the application, which is a summary of the time spent per function out of the total execution time. The second and more detailed type of output is a call graph which shows how much time was spent on each function and its children. The first step during profiling was to add the `-pg` flag for the compilation of *BWA-MEM* and run it with the human genome GRCh37 as the reference against the SRR835433_1 dataset as a query. From the results of this first profile run, as well as the general study of the *BWA-MEM* application, we created a simplified visualization of the function call graph during a typical run of *BWA* with the *MEM* command. In Figure 4.1 a high level representation of the most important function calls of *BWA-MEM* can be seen.

A typical program run begins at the `main` function, which selects the algorithm that will be executed based on the user input. In the context of this thesis, we are interested only in the *MEM* functionality, but it is important to mention that the `main` function is the one that calls the FMD index building function, which we assume has been performed previously. The `main` function will pass to the `main_mem` function all the parameters the user requests to perform the alignment. The application will instantiate workers equal to the number of threads the user requires the program to run on. In Figure 4.1, we can see a distinction between `worker1` and `worker2`, and in our case, the former is of particular interest since it includes all the calls to the functions that find *(S)MEMs* (marked in red), chains of seeds (marked in yellow) and finally perform the alignment (marked in green). In the baseline implementation, each instance of a worker will operate on one of the reads from the query file, and once a worker becomes available, it receives another read from the batch. As a result, *BWA-MEM* is very efficient since each CPU thread will perform the alignment for one query in parallel and independently to other threads. The result of `worker1` will be passed on to `worker2` which generates the final alignment information to be included in the *SAM* file.

4.1.1. Application Profile

In order to create an accurate profile of the application, we need to understand how the various functions interact with each other in the *BWA-MEM* pipeline. The high-level call graph from Figure 4.1 provides a good starting point that enables us to distinguish some of the key parts of the algorithm:

- **SMEM Generation**: this is the portion of code that calculates the seeds utilizing Algorithm 5.

Algorithm 8: Original batching structure pseudo code adapted from [4].

Input: a batch of n reads
Output: n aligned reads
1 **for** $i = 1$ *to* n **do**
2 | Seed Generation(read i)
3 | Seed Chaining(read i)
4 | Seed Extension(read i)
5 **end**
6 **for** $i = 1$ *to* n **do**
7 | Output Generation(read i)
8 **end**
9 **End Function**

Inside the *BWA-MEM* pipeline, this function is called from the chaining function, which should be profiled separately.

- **Suffix array accesses (locate)**: this is the function that receives all the suffix array indexes and returns the position of the seed in the reference as in Algorithm 2. Similar to seed generation, this function is invoked from the chain function.
- **Chaining**: this is the function that takes the seeds and creates chains of them, but as we mentioned in this function, the seeding also takes place, which should be included separately, as should the suffix array indexes calculation, which is used to create the chains of seeds. Furthermore, chain filtering is also included in this portion, and the same applies to some of the operations performed on chains during the `mem_chain2aln` function, which has the calls to the extension kernels.
- **Ksw_extend**: the extension fraction of code is only counted as the left and right extension which is called from `mem_chain2aln` and can be seen as the green operation `ksw_extend` in Figure 4.1.
- **Auxiliary Operations**: these are all the operations that load inputs, prepare the parameters and create multiple threads for the workers. In this fraction, we include the load operation on the FMD index, as well as loading the queries into batches.
- **Other**: in this category of operations, we include the remaining functions which are outside the scope of our profiling. The most dominant operation in this category is the set of functions executed by `worker2`.

Dataset	SMEM	SA Access	Chaining	Ksw_extend	Aux Ops	Other	Runtime
SRR921889_1	31.5%	36.1%	8.6%	20.3%	0.2%	2.5%	2681.2s
Simread_100	41.3%	24.7%	5.1%	23.1%	0.6%	5.2%	1651.7s
Simread_150	39.2%	20.9%	3.6%	28.6%	0.4%	7.3%	2803.6s
Simread_200	36.7%	18.6%	2.9%	32.4%	0.3%	9.1%	4007s
Simread_250	34.8%	17.5%	2.4%	34.4%	0.2%	10.7%	5394.1s
SRR835433_1	39.5%	14.6%	4.3%	36.5%	0.4%	4.7%	3027.2s

Table 4.2: *BWA-MEM* profiling results for minimum seed length of 19.

The *BWA-MEM* program is executed using its default parameters and one processing thread. In addition to the *gprof* tool, real-time measurement counters were added to verify each of the program's key parts that were presented in the list above. In Table 4.2 the results are presented for each dataset run. Each result was verified by multiple executions and monitoring CPU and memory usage during each run to monitor any unexpected system loads that could compromise the profiling results.

In Table 4.2 we see that the most time-consuming tasks are seeding, which can take between 31% and 41% of the total execution time, and the extension function, which takes between 20% and 36%

of the total time. These two portions also exhibit an inverse relation. As the simulated read length increases, the alignment takes a more considerable portion while the seeding portion reduces. This behavior can be explained by the complexity of the alignments being higher for longer query sequences since the similarity matrix has larger dimensions and the dynamic programming algorithm requires more time to complete. In the case of the real dataset SRR835433_1, which has much more difficult alignments to perform compared to the simulated reads, it can be seen that we find the highest time percentage for the extension. The more complex alignments are also reflected in the output *SAM* files which for the simulated reads have mostly matches, while for the real dataset, there are many more *indels*. The portion of time taken by the suffix array locate function is also considerable, but it does seem to reduce as the read length increases. The chaining and the other operations, as we named them, can take up to 12% of the total execution time, which is a minor fraction compared to seeding and extension. Similarly, *BWA-MEM* spends an insignificant amount of time on auxiliary operations, especially in the context of a large number of queries where the initial long FMD index read from the disk becomes inconsequential, and the same comment applies to loading new batches of reads.

The results mentioned above were also compared to previous work done in the context of this project, where researchers profiled older versions of *BWAMEM* utilizing different datasets. In [14], the authors found very similar results to the ones shown in Table 4.2 while in [15], the authors only show results for seed generation and seed extension, and the numbers seem to follow our findings. Although the software versions might be dissimilar, the core functionality has not changed much over the past few years.

In conclusion, seeding and extension are the most dominant stages, which together can amount from 64% to 76% of the execution time. According to Amdahl's law presented in Chapter 2, these stages should be the focus of any improvement efforts and could provide a maximum theoretical speedup between 2.7× and 4.2× according to our findings. Finally, if we were to add the slice of time taken for locating the seed in the reference, the portion of the program which is a candidate for improvement would reach 90% yielding a theoretical maximum of 10× speedup, as can be seen in Figure 2.5.

4.1.2. Performance with Multithreading

One of the main capabilities of the *BWA-MEM* aligner is its capability to use multiple parallel threads to execute its worker functions. When the user requests more than one thread, the execution method is to instantiate an equal number of `worker1s` to the number of threads. Each `worker1` will receive one query sequence at a time from the batch and will process it in parallel to the rest of `worker1s`. After the first type of workers has processed all the query sequences, the program instantiates `worker2s` equal to the number of threads and follows the same parallel execution method as the first type of workers. In Figure 4.2, the results of executing the *BWA-MEM* program by increasing the number of threads in increments of 2 and up to the limit of our platform of 24 threads. The dataset used is the SRR835433_1, with around 8.3 million reads and 250 bp per read. From 4.2, it can be deduced that execution time is halved by using two threads instead of just one, while execution time is around 2.7 times faster when using six threads instead of 2. The execution time decrease starts to flatten out after ten threads, and the maximum performance is achieved for 24 threads by an insignificant margin over 20 or 22 threads. The main reasons behind this last statement are the rest of the services running on the server, which require CPU resources to keep executing.

4.2. GPUSeed Analysis

The analysis of the *GPUSeed* program will follow a different methodology than the method used to analyze *BWA-MEM*. First of all, since *GPUSeed* is implemented in *CUDA*, we cannot utilize a tool like *gprof* since we are now dealing with kernels running on a GPU. The solution is to use NVIDIA's Nsight Systems software which comes bundled with the *CUDA* toolkit [68]. This software is a visualization tool that collects GPU traces and creates a visual timeline of all the various *CUDA* API calls, as well as the kernels running on the GPU. The visual profile provides valuable information to the developer regarding potential bottlenecks in the application and inefficient use of GPU resources. In addition, NVIDIA provides the NVTX C-based API, which can be used to annotate ranges of code so that CPU functions or interesting parts of code can also be added to the visualization of Nsight Systems. Another method used to measure the fraction of time it takes for the various kernels of *GPUSeed* is traditional time measurement functions that wrap around kernel or function calls. However, since the default stream

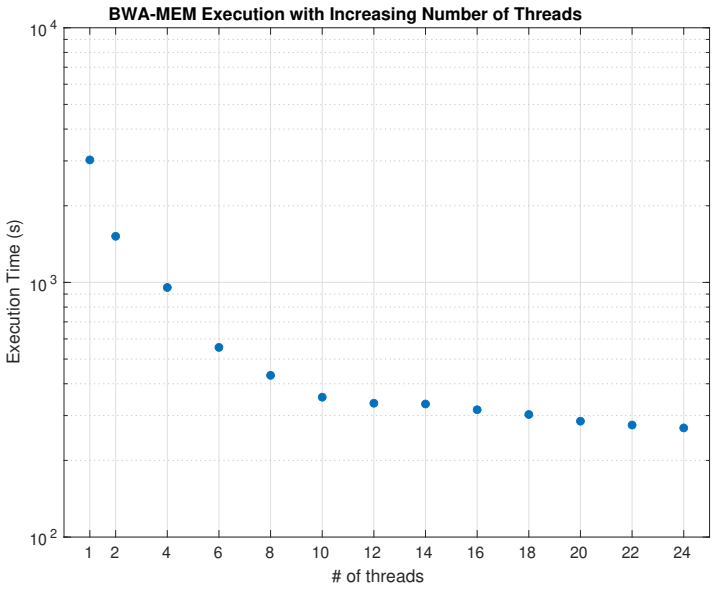


Figure 4.2: *BWA-MEM* multithread execution for SRR835433_1.

is non-blocking after each kernel call, we need to add the *CUDA* device synchronize function calls which stall all other *CUDA* API calls and host execution until the device finishes that particular kernel. An alternative solution to this method is to utilize *CUDA* events, but it would introduce unnecessary complexity since it requires variables and more lines of code.

After presenting the pipeline and evaluating the profile of *GPUSeed*, we will look at the quality of the seeds compared to the seeds that come out of the *BWA-MEM*.

The *GPUSeed* discussed in this section is the publicly available one on GitHub [61] and the same version published in [52]. In the following chapters, during the integration and optimization, we will present some newer unpublished versions of *GPUSeed* which were previously developed, and we will include those as improvements to a baseline implementation. Nevertheless, the general structure of *GPUSeed* is presented in Figure 4.3, and every time we present a change, we will refer back to this flow chart as our baseline.

4.2.1. Application Profile

Figure 4.3 shows the flow chart of a typical run of *GPUSeed*. The first phase will open and load a pre-built FM index with the structure we presented in section 3.2 of Chapter 3. The first GPU kernel being invoked is the one that will calculate all the possible suffix array intervals for a predefined length. The next step is to start loading batches of reads, each batch with the reads corresponding to 1 million bases from the file. The packing and prepare kernels will create the forward and reverse 4-bit packing and assign GPU threads to specific *SMEMs*. The next kernel will take the batch of reads to find their seeds and their intervals on the suffix array for both the batch's forward and reverse complement reads. The GPU seeds require further filtering to remove redundant seeds as there is no way to compare *MEMs* of the same query found by threads residing on different blocks. The last step in the GPU pipeline is to locate the seeds in the text in a similar fashion as *BWA-MEM* finds the reference starting position during chaining. Note that in Figure 4.3, we omitted memory transfers between host and device since this pipeline includes many small back-and-forth data movements. In the final profile analysis, we include three types of transfers. Reading the FM index from the file and transferring it to the GPU is included in the index loading portion. The second memory transfer we include is the copies of each batch of reads from host to device, and the third retrieves the final seeds from the GPU to the CPU. These transfers are included in our profile's memory copy portion of time.

The application is executed for each dataset, with two executions being called from Nsight Systems and two from the regular command line. In order to verify the results, we check the outputs for differences. For example, in the case of Nsight Systems, we look at the timeline and the output file, which

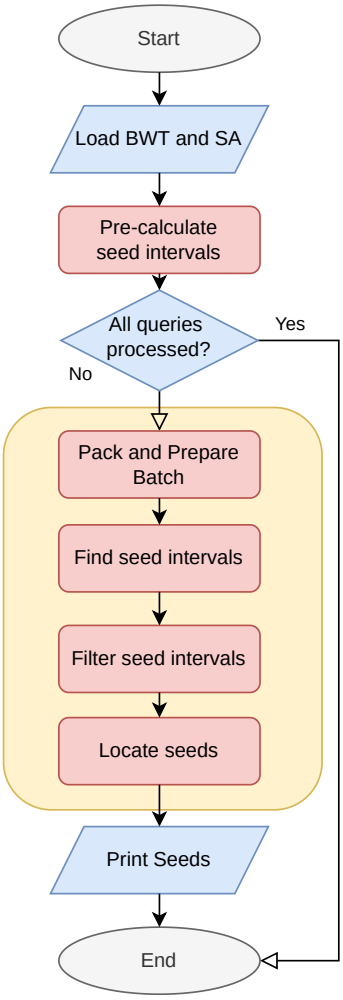


Figure 4.3: *GPUSeed* flow chart.

gives the results of the time counters. These values are then compared against the regular command line runs to account for any significant overheads introduced by Nsight Systems or our time-measuring functions.

The results of the Nsight Systems runs can be seen in the timelines in Figures 4.4 and 4.5. The first image presents the beginning of the pipeline where the initial memory copies transfer the BWT and the suffix array to the device. The timeline continues with the first kernel, which pre-calculates the seed intervals. After this first kernel, we see a series of 16 smaller kernels that start after the 7-second mark. These are the `find_seed_intervals_gpu` kernels repeating for each new batch of queries. In the zoomed-out view of Figure 4.4, the smaller kernels which run before and after the find intervals kernel cannot be distinguished, so we merged two zoomed-in timelines from before and after an instance of a `find_seed_intervals_gpu` kernel into Figure 4.5. In the top part of Figure 4.5, we see the memory transfers passing that specific batch of reads to the GPU and then the kernels packing and preparing the batch before the start of find seed intervals. In the lower part of Figure 4.5, we see that after the interval calculation of the same batch, the execution pipeline continues with the filtering kernel, a series of CUB kernels and finally, the locate kernel. After the locate kernel, which takes the suffix array intervals and returns the seed positions in the text, the resulting seeds are copied to the CPU, and the program moves to the next batch, or it exits if it was the last batch.

4.2.2. Performance

The total execution time and the percentages of time spent in each of the stages of *GPUSeed* can be seen in Table 4.3 for all the different query files. The results show that most of the time is spent in the

Dataset	Index Load	Pre-calculate Intervals	Prepare Batch	Find Intervals	Filter Intervals	Locate	Memcpy	Other	Runtime
SRR921889_1	8.7%	1.4%	5.6%	31.6%	15.1%	7.2%	3.7%	26.7%	17.7s
Simread_100	5.3%	0.9%	9.6%	47.9%	11.3%	3.1%	2.1%	19.8%	28.0s
Simread_150	3.5%	0.6%	7.9%	54.3%	8.0%	3.0%	2.0%	20.7%	43.0s
Simread_200	2.6%	0.4%	7.9%	57.3%	6.3%	3.0%	1.9%	20.6%	58.1s
Simread_250	2.4%	0.3%	7.5%	58.7%	5.3%	2.9%	1.9%	20.9%	73.8s
SRR835433_1	2.7%	0.4%	3.0%	64.2%	5.1%	1.4%	1.5%	21.8%	60.3s

Table 4.3: *GPUSeed* profiling results.

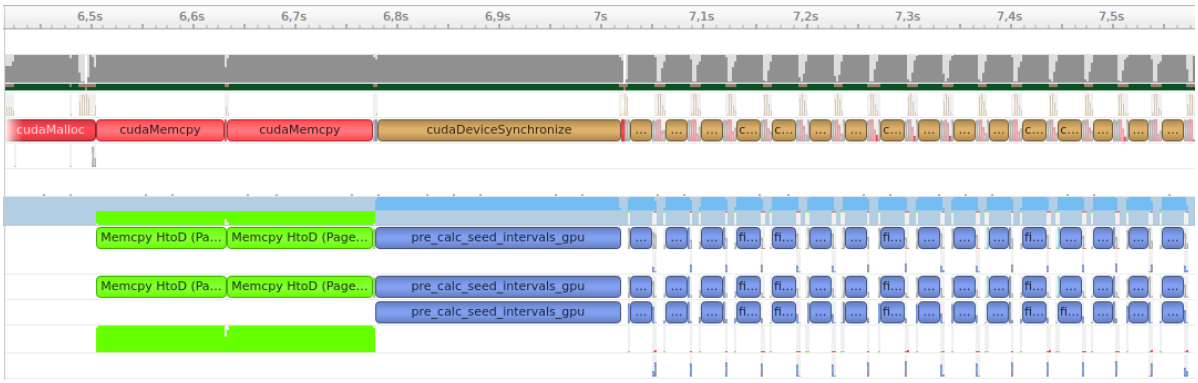


Figure 4.4: *GPUSeed* initialization Nsight Systems Timeline for SRR835433_1.

find intervals portion, which takes more than half of the total execution time for most of our datasets. It is also important to note that the percentage out of the total time spent in the find seeds interval kernel increases for larger queries. Furthermore, we see that the kernel that pre-calculates seed intervals takes an insignificant portion of the total time, especially when we have many queries. This last kernel is independent of the query and only depends on the reference, meaning that for all the datasets, we have the same execution time for this pre-calculation kernel. The locate and filter kernels take very small fractions out of the total time, and both these portions reduce for the larger queries. The same comment applies to memory transfers. The last category includes the other operations where we have the *CUB* functions which perform various device-wide reductions, selections and sorting operations. These functions run on the GPU, and some examples of their uses in our case are sorting pairs of seeds, selecting only *SMEMs* from a list of seeds, exclusive summations and other similar functions. In Figures 4.4 and 4.5 the *CUB* kernels can be seen executing inside the batching pipeline.

Lastly, we compare the results of the *GPUSeed* profile found in Table 4.3 with the results of *BWA-MEM* from Table 4.2 to draw some conclusions on improvements of the seeding phase. In the case of *BWA-MEM*, we take the time required for finding the *SMEMs* and also the time taken by the suffix array lookup function from the third column in Table 4.2. These two portions of time are comparable to the complete pipeline of *GPUSeed* since we are also calculating the positions of the seeds in the reference. The comparison shows that *SMEMs* are found around 33× faster in *GPUSeed* for the synthetic reads, while for the real dataset, the number goes down to 24×. These numbers show an excellent improvement, but a few considerations must be made when evaluating such a huge improvement.

1. *GPUSeed* does not save the seeds in any form of data structure. The seeds are simply transferred back to the CPU, then printed and the memory freed for the next batch of seeds. On the other hand, *BWA-MEM* saves and orders the seeds in data structures to be utilized further down the pipeline. These memory operations add complexity to the seed detection of *BWA-MEM*, which should be considered when looking at the improvement number presented above. *BWA* does offer the *fastmap* command, which only calculates *SMEMs* and prints them in a similar manner to *GPUSeed*. Since we would like to replace the seeding in the pipeline of *BWA-MEM* with the GPU implementation, we consider it a better comparison to look at the profile of the seeding function in the context of the complete *BWA-MEM* pipeline.

2. *GPUSeed* seems to find redundant *MEMs* which should be filtered out. The reason behind these redundant seeds is the grouping of the forward seeds at the start of the array and the reverse seeds at the end. During filtering, the forward seeds are compared among each other, and the

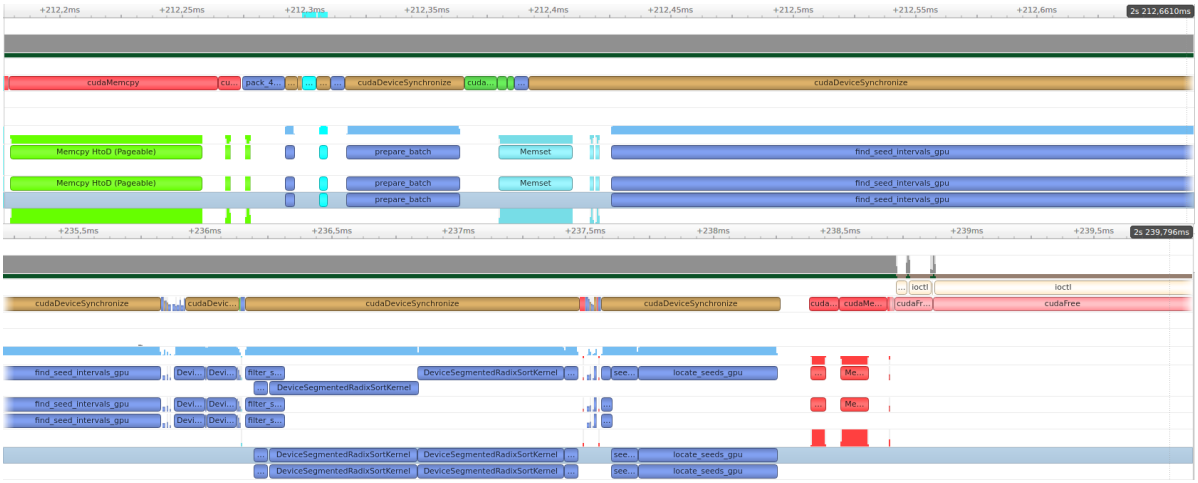


Figure 4.5: *GPUSeed* batch iteration Nsight Systems Timeline for SRR835433_1.

same applies to the reverse seeds. The outcome will be that a reverse seed contained in a forward seed or vice versa will not be filtered, resulting in redundant *MEMs*. In order to have an identical result to *BWA-MEM*, the GPU program has to implement additional filtering or a different search strategy which could potentially harm performance.

4.2.3. Results Evaluation

Seed quality is essential for the rest of the analysis. The first important factor we consider when discussing the quality of the seeds is the number of redundant *MEMs*. These seeds, which theoretically will not be extended, still take memory and computing resources, especially during chaining. The second and more important factor is finding the same *SMEMs* in *GPUSeed* as in *BWA-MEM* so that we can perform the seeding part on the GPU without having significant differences in the output. For example, in *GPUSeed*, a seed that appears multiple times in the reference will be split, and then its location in the reference will be computed separately. Meanwhile, *BWA-MEM* just keeps the starting coordinate of the seed on the suffix array and the interval size. As a result, text positions of multiple instances of one seed on the reference are found during chaining using an iterative process starting from the base coordinate and up to the size of the interval. The two methods will have differences in the final seeds as, in some cases, the same *SMEM* coming from *GPUSeed* will have instances that were found in the forward direction and some found in the reverse direction using the FM index. The consequence of this scenario is that since the forward and reverse seeds are separate, there is no method of selecting the same seed as in *BWA-MEM* where the same seed is found once using the bi-intervals of the FMD index.

In order to also evaluate the quality of the seeds coming from *GPUSeed* compared to *BWA-MEM*, we perform a simple study utilizing a small subset of 100 queries from the SRR835433_1 dataset and finding seeds on the GRCh37 genome. For each query, we measure:

- The percentage of *SMEMs* that are identical between the two implementations is 62%.
- The percentage of redundant *MEMs* found by the GPU program is 19%.
- The percentage of the same *SMEMs* that have a different location in the text is 18%.
- The percentage of *SMEMs* not detected by *GPUSeed* which appear in *BWA-MEM* is around 1%.

4.2.4. GPU Resource Utilization

In Figure 4.6, we present the GPU utilization percentage reported by the NVIDIA driver in Linux over the complete execution of the SRR835433_1 dataset with the GRCh37 genome. In order to obtain the data in Figure 4.6, we utilized the `nvidia-smi` command to sample the GPU utilization percentage every 100ms and wrote the results to a CSV file. From Figure 4.6, we can conclude that the *GPUSeed* library uses around 85% of our GPU, and the utilization remains relatively constant over the execution

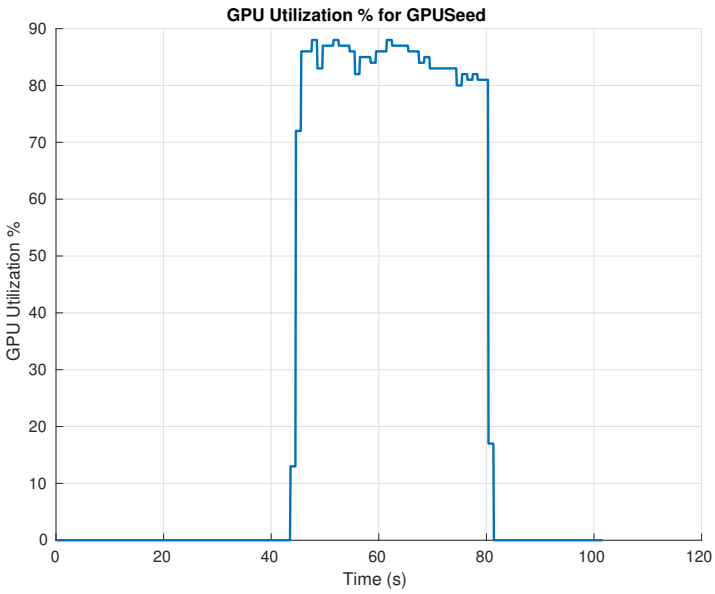


Figure 4.6: GPU % utilization by *GPUSeed* for SRR835433_1.

of different batches of queries. Furthermore, we also monitored the GPU memory usage by the seeding library, finding that over the complete execution of the SRR835433_1 dataset with the GRCh37 genome, *GPUSeed* uses maximally 4230MB of GPU memory.

4.3. GASAL2 Analysis

In this final analysis, we will look at the implementation of *GASAL2* performing alignments on the GPU. This solution has been added into the *BWA-MEM* pipeline to accelerate the seed extension part. The version of *BWA-MEM* used with *GASAL2* is an earlier iteration of 0.7.17, so we do not expect the core program to exhibit too many differences from the baseline. Nevertheless, we will monitor the execution times of all the main stages for any discrepancies.

The methodology followed for the analysis of this program is similar to the methods used in Sections 4.1 and 4.2. For the parts utilizing regular CPU code, such as the seeding and chaining parts, we utilize precisely the same time counting functions placed at the same points as in the profile of the baseline *BWA-MEM*. For the portion of code running on the GPU, we will utilize Nsight Systems to create a timeline visualization of the kernels. In addition, time measurement functions will be used to verify the kernel execution times. The difficulty in profiling this program arises from the asynchronous behavior of the kernel calls and the overlap of CPU and GPU execution. The default version of the program will utilize two GPU streams per CPU thread to launch two kernels. The first kernel packs the query and the reference sequences into 4-bits and stores them into 32-bit entries of two respective arrays. The second kernel of each asynchronous *GASAL2* launch is the extension kernel which performs the alignment calculation on the same pack of queries and reference sequences.

In Figure 3.8 of the previous chapter, we presented the general workflow of the *GASAL2* library, but the batching strategy utilized was not shown. Implementing the local alignment on a GPU requires altering how the batches of data are processed so that the resources are utilized more efficiently. The baseline *BWA-MEM* takes one query sequence at a time and processes it. This method scales well for the CPU implementation since each thread is assigned one sequence from the batch of data. In the case of *GASAL2*, if we were to have only one sequence at a time being processed by the GPU, it would result in a substantial overhead, and probably slower execution than on the CPU [4]. Therefore, the `worker1` function we saw earlier in the *BWA-MEM* profile is altered so that it takes a chunk of 5000 reads which are processed 1000 at a time. Each query from these 1000 reads follows the familiar journey of seeding, chaining and chain filtering. After the 1000 reads have been processed, the program will launch two asynchronous kernels, one for the long extension and one for the short extension for

Algorithm 9: Batching Pseudo Code adapted from [4].

```
Input: a batch of n reads
Output: n aligned reads
for i = 1 to n/chunksize do
    for j = 1 to chunksize/internal_chunksize do
        for k = 1 to internal_chunksize do
            Seed Generation(read k + (j − 1) × internal_chunksize + (i − 1) × chunksize)
            Seed Chaining(read k + (j − 1) × internal_chunksize + (i − 1) × chunksize)
        end
        Seed Extension Kernel (all reads in chunk j)
    end
end
for i = 1 to n do
    Output Generation(read i)
end
End Function
```

this chunk of 1000 reads. This process will continue for the subsequent 1000 reads until the batch of 5000 reads is fully processed and the `worker1` returns the results so that `worker2` can build the *SAM* file for the batch. The short and long extensions are mentioned in section 3.3.5. The main idea of this batching method is presented in Algorithm 9, which is adapted from [4].

4.3.1. Application Profile

The parts of code in which we are interested in the context of this program are very similar to the ones presented for the *BWA-MEM*. Therefore, this method will ensure that we have comparable results between the two programs so that we can draw conclusions regarding performance and improvement due to the GPU acceleration of the extension phase. Thus, the profiling of this program will be done for the following parts:

- **SMEM Generation**: this is the portion of code that calculates the seeds utilizing Algorithm 5. Inside the program pipeline, this function is invoked from the chaining function.
- **Suffix array accesses (locate)**: this is the function that receives all the suffix array intervals and returns the positions of the seeds in the reference using Algorithm 2. This function is invoked from the chain function.
- **Chaining**: this is the function that takes the seeds and creates chains of them, but as we mentioned before, in this function, the seeding also takes place, which should be included separately, as should the suffix array indexes calculation, which is used to create the chains of seeds.
- **Chain2aln**: the function that prepares the query and the reference sequences for alignments. *GASAL2* uses this function to fill its data structures on the CPU side and transfer them to the GPU. Note that we include this function separately as it has multiple *GASAL2* functions, while in the case of the *BWA-MEM* baseline profile, this function was included together with chaining.
- **Ksw_extend (GPU Kernel)**: the extension fraction of code is only counted as the visible time the GPU runs the packing and alignment kernels. The visible time is the time that the CPU thread will wait for the results of the GPU and not the total time spent in the extension kernels. Since the extension kernels are launched on two GPU streams for each CPU thread, we cannot simply count the total extension time spent in the kernels as some kernel executions might overlap. Furthermore, the asynchronous behavior of the kernel launches and the CPU-GPU execution overlap during the seeding, chaining and alignment pipeline, creates a more complex profiling situation where we will have a case in which a kernel is still computing the previous batch while the program moves to find the seeds for the next batch. We consider that in terms of the complete execution profile of our application, visible kernel time provides a more significant figure.

- **Auxiliary Operations**: these are all the operations that load inputs, prepare the parameters and create multiple threads for the workers. In this fraction, we include the load operation on the FMD index, as well as loading the queries into batches. In this category, we also include the initial memory allocation and eventual de-allocation of *GASAL2*.
- **Other**: in this category of operations, we include the remaining functions which are outside the scope of our profiling. The most dominant operation in this category is the set of functions executed by `worker2`.

Dataset	SMEM %	SA Access %	Chaining %	Chain2Aln %	Ksw_extend %	Aux Ops %	Other %	Extensions	Runtime
SRR921889_1	39.0%	44.5%	4.4%	6.8%	0.2%	0.3%	3.2%	316428994	2158.3s
Simread_100	52.7%	31.0%	4.4%	3.4%	0.5%	0.9%	6.2%	70647574	1385.3s
Simread_150	54.0%	28.7%	3.3%	2.5%	0.9%	0.2%	9.6%	76515922	2086.9s
Simread_200	52.8%	27.1%	2.7%	2.1%	1.2%	0.4%	12.9%	81531923	2899.8s
Simread_250	51.5%	25.7%	2.2%	1.9%	1.6%	0.3%	16.0%	85520865	3487.1s
SRR835433_1	60.3%	22.1%	2.7%	4.5%	1.9%	0.5%	6.6%	121277939	2008s

Table 4.4: *GASAL2* performing alignment in *BWA-MEM* profiling results for minimum seed length of 19

In Table 4.4, we present the results of the profile study. In this section, we also add the number of extensions performed by the aligner, as the program provides the capability to measure this metric and can provide additional insights into the results. The execution time taken by the *SMEM* generation and suffix array locate function are identical to the values found for the baseline *BWA-MEM*, which is what we expected even though the program works on batches instead of one query sequence at a time. Furthermore, we see that both *SMEM* and locate functions take more significant percentages of the total execution time when compared to the values of Table 4.2, showing us that even though the extension phase was accelerated, the overall improvement of the program is still heavily restrained by seeding and suffix array functions. In addition, we also compare the execution time of the chain to alignment function (i.e., `chain2aln`) for this program to the execution time of its counterpart from the baseline *BWA-MEM* since there is added complexity required to fill the *GASAL2* data structures while also having a different logic for the exclusion of seeds that should not be aligned. We see a small 8% to 20% increase in execution time over the baseline for this particular function. The time spent in the extension kernel as a percentage of total execution time seems to scale with the number of extensions for the simulated reads, while in the case of real datasets, the short SRR921889_1 set of queries has a very high number of extensions, almost three times more than SRR835433_1, but the time spent in the extension kernel is around 9.5 times less. This behavior is due to the length of the alignments being much smaller (i.e., shorter queries), showing that the extension kernel is much more efficient at short alignments than at larger ones, even though there are many more alignments to be performed. Finally, we summarize the conclusions of this comparison between the execution profiles of two real datasets:

- Datasets with shorter queries have a higher percentage out of the total execution time being spent for the suffix array functions, and the same applies for all the chaining stages.
- Chain to alignment function has to iterate over more seeds, resulting in more time spent in this part when compared to the baseline.
- The extension kernel exhibits much better efficiency for short alignments than for large ones. This is a direct result of the GPU architecture, which is much better suited for simpler tasks where the warps can execute in lockstep rather than a large alignment that can have divergent threads and also the dynamic programming complexity, which is impacted directly by the query length.
- Higher number of extensions does not necessarily result in longer time spent in the extension phase.
- The quality of the queries also plays an essential role as low-quality alignments will exit faster the extension loop, as was the case for SRR921889_1.

The last profiling results are shown in the form of a Nsight Systems timeline presented in Figure 4.7, where we present a merge of a zoomed-out and a zoomed-in timeline similarly to the *GPUSeed* profile.

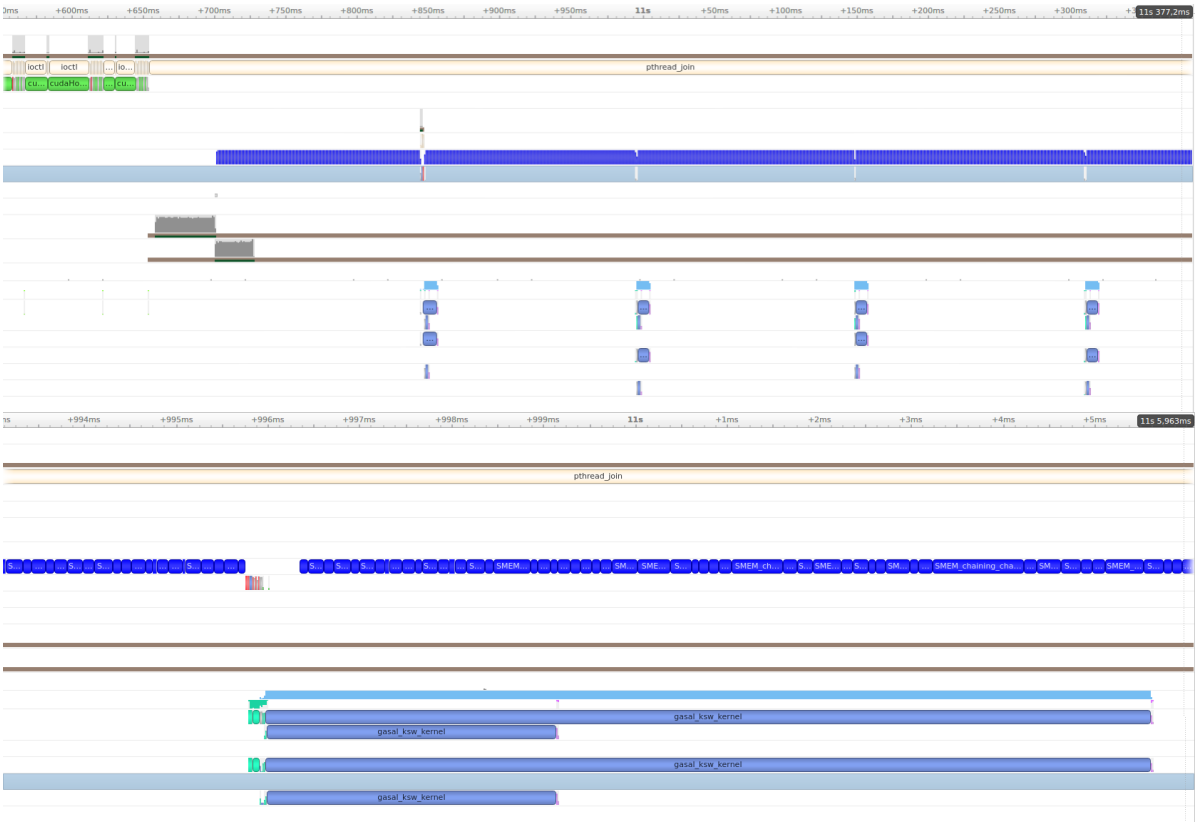


Figure 4.7: *GASAL2* performing alignment in *BWA-MEM* Nsight Systems Timeline for SRR835433_1.

The upper part of Figure 4.7 shows the different chunks of queries being processed in the extension kernels, which are the small purple-blue timelines. The lower part of Figure 4.7 presents a zoomed-in view of one of the kernel invocations, while the dark-blue timeline is an NVTX range showing the combined CPU execution of seeding, chaining and the chain to alignment function. From these images, we can distinguish two essential characteristics of the kernel launches. The first can be seen in both upper and lower timelines and is the launch of two concurrent extension kernels, one for short and one for longer alignments, each running on a different *CUDA* stream. The second feature can be seen more clearly in the lower part and is the overlap of GPU and CPU execution which happens after the launch of the two kernels, with the CPU moving on to process the next chunk of 1000 queries from the batch. We can draw an important conclusion from these visualizations: the CPU part performing seeding and chaining bottlenecks the GPU, and we expect GPU utilization to be relatively low. The low GPU utilization can be alleviated by utilizing more CPU threads; since each CPU thread receives two GPU streams, we expect the GPU to be used more efficiently.

4.3.2. Performance

In this part, we look at the performance of the GPU implementation with respect to the baseline *BWA-MEM*. The first comparison we make is of the two implementations using a single CPU thread for their execution. In Figure 4.8, we see that the best speedup for the datasets from Table 4.1 is achieved for SRR835433_1, which is 1.5× faster. This speedup is very close to the theoretical maximum improvement in the case only the extension phase is accelerated. By taking Amdahl's law and the entry for the portion of time spent in extension from Table 4.2, we find that the theoretical maximum is 1.57×. It is important to mention that in our case, we also have the overlap of CPU-GPU execution we previously mentioned. This overlapped execution could be considered to some extent, a form of acceleration, but we assume that only the CPU functions that were replaced by GPU kernels should be considered in the improvement figures.

The second study we present aims to compare the execution of this program using an increasing number of threads to the previously presented baseline *BWA-MEM* using multithreading from Fig-

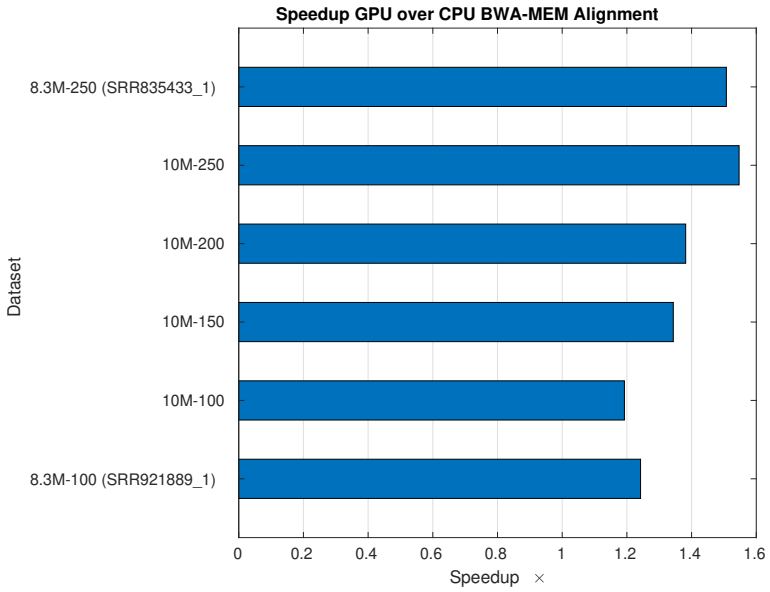


Figure 4.8: GPU alignment speedup over baseline *BWA-MEM*.

ure 4.2. The accelerated program is also executed by increasing the number of threads up to 24, which is the limit of our platform. The combined results are presented in Figure 4.9. The results show that the accelerated program follows the scaling in performance that the baseline *BWA-MEM* follows with an increasing number of threads. The execution time is halved by going from one to two threads and halved again going from two to four. After ten threads, we see a stabilization with a lower rate of decrease in execution time. Furthermore, in Figure 4.9, it can be seen that the best performance can be achieved utilizing 24 threads but the differences compared to execution with 20 or 22 threads are negligible. The same observation was made in the baseline analysis, where this behavior was attributed to the fact that the platform also has other services running that take resources and when we use the maximum available threads, the OS (Operating System) needs to keep those services running while also scheduling threads for our program.

4.3.3. GPU Resource Utilization

In Figure 4.10, we present a study into the utilization of GPU resources by the *GASAL2* library. The first plot in Figure 4.10 presents the GPU utilization percentage over the first 200 seconds of execution of the SRR835433_1 dataset. This plot is obtained in the same manner as in the *GPUSeed* utilization study. In Figure 4.10, we can see that the reported utilization using one thread for execution is below 20%. This result is expected since the reported occupancy for the extension kernel is also low at around 10%, depending on the batch being processed. On the other hand, using 24 threads to execute the same dataset increases the reported utilization to around 80% for some batches and 60% for others. In addition, we also see peaks of utilization, even reaching 100% for brief moments during this execution.

In conclusion, the implementation of *GASAL2* using two GPU streams per CPU thread improves GPU utilization even though the occupancy of the extension kernel is low. Finally, the second plot of Figure 4.10 presents how the GPU memory utilization grows linearly with an increasing number of threads. Specifically, the memory pre-allocated by the *GASAL2* is constant and depends on the number of threads (i.e., 2 GPU streams per CPU thread), the pre-allocation coefficients, and the maximum query length passed at compilation time (i.e., *GASAL2* stores a complete row during the DP calculation).

4.3.4. Results Evaluation

The alignment results using the GPU implementation are evaluated against the baseline alignments of *BWA-MEM*. The dataset utilized for the comparison is the SRR835433_1 dataset mapped against the GRCh37 genome. In order to compare the results, we write the two *SAM* outputs from each program to a file and then utilize the `diff` command that can be found in Linux in order to see how many lines

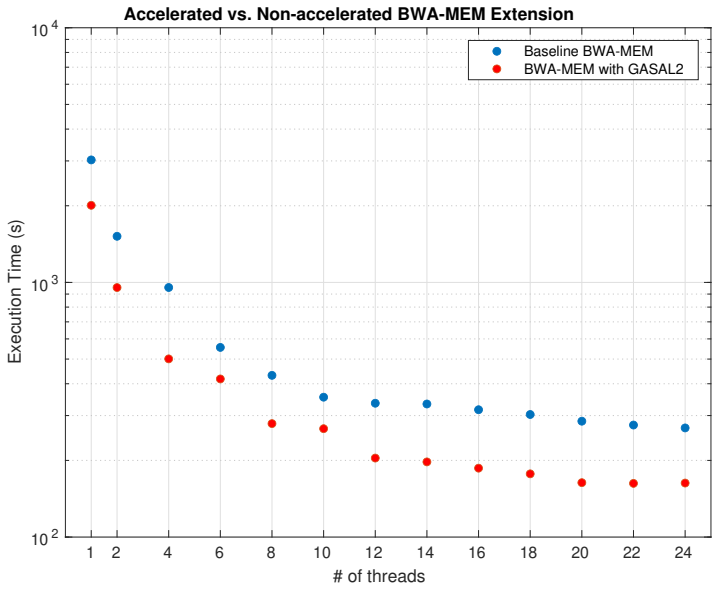


Figure 4.9: Combined *BWA-MEM* multithread execution for SRR835433_1.

match between the two *SAM* files. Using the `diff` command, we find the percentage of lines that differ between the two files is 7.41%. This number is further reduced to 1.1% if we remove all alignments with mapping quality below 20 and all secondary scores where the differences are more pronounced due to *BWA-MEM's* secondary best alignment.

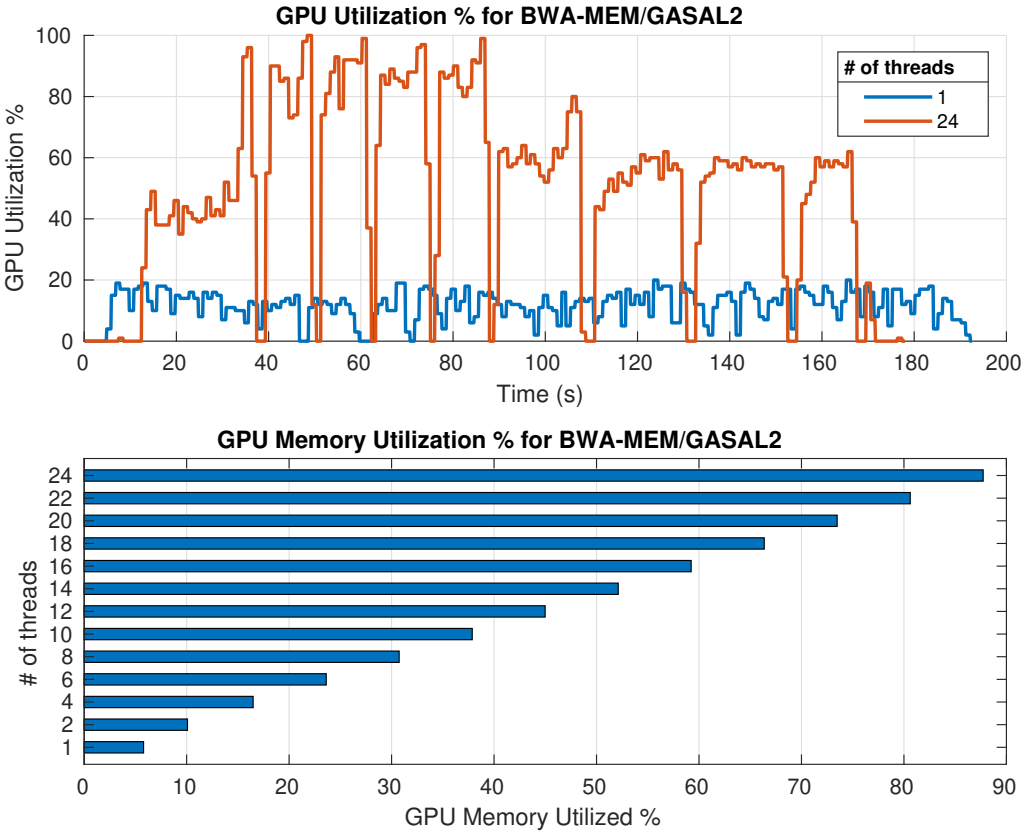


Figure 4.10: GPU Utilization % and memory utilization % for SRR835433_1 dataset.

5

Integration & Optimization

In the first part of this chapter, we present the integration strategy that was utilized in order to add the *GPUSeed* and *GASAL2* libraries into the latest version of *BWA-MEM*. The integration will follow the steps that were taken during the project, with the first part of this chapter presenting the addition of *GPUSeed* into the *BWA-MEM* pipeline and replacing the CPU function that finds *SMEMs* with the *GPUSeed* library. Then, the integration continues with the addition of the *GASAL2* library into the initial integration.

Furthermore, the second part of this chapter explores the optimizations done to the integrated library to achieve better performance as well as functional improvements aiming at bringing the alignment result as close as possible to the output of the baseline *BWA-MEM* implementation.

5.1. GPUSeed Integration into BWA-MEM

In the previous chapter, we witnessed how the acceleration of the *BWA-MEM* algorithm is hindered by the fraction spent in the seeding phase, which can reach around 40% in the baseline version and 60% when the extension is accelerated. At the same time, if the portion of time spent in the suffix array locate function is also taken into account, the total fractions reach around 55%, and 80% for the respective non-accelerated and GPU accelerated implementations. These statements were heavily reflected in the numbers we presented regarding the overall speedup achieved by *GASAL2* performing the local alignment in the *BWA-MEM* pipeline. On the other hand, as expected, the baseline program scaled very well with increasing threads since the worker functions have independent instances on different CPU threads. Consequently, it is crucial to accelerate the seeding and locate stages so that the GPU program can gain any significant execution time advantage over the multithreaded *BWA-MEM* execution.

The first step in our integration process was to add the publicly available *GPUSeed* library that we presented in the earlier chapters into the latest version of *BWA-MEM*. This process begins with a naive implementation where the GPU library replaces the primary seeding function. This kind of integration is then improved to take advantage of the GPU kernel execution while keeping *BWA-MEM's* characteristic of running on multiple threads.

5.1.1. Naive Integration

The naive implementation began by looking at the simplified call graph of *BWA-MEM* in Figure 4.1 and finding candidate spots for the addition of the GPU seeding. The first instinct was to replace the call to the `mem_collect_intv` function with a function that will perform the seeding on a GPU. The main task then becomes adding the *GPUSeed* function in the pipeline, successfully compiling the complete project and calling the GPU seeding function from the chaining function, which is where `mem_collect_intv` gets called. The challenges of this task can be summarized as the following:

1. *BWA-MEM* uses *GCC* to compile its complete project while *GPUSeed* is written in *CUDA* which is compiled with the *NVCC* compiler.

Algorithm 10: Pseudo Code for naive integration of *GPUSeed* into *BWA-MEM* inspired from [4].

```
Input: a batch of n reads
Output: n aligned reads
Seed Generation(all reads n)
for i = 1 to n do
    Seed Chaining(read i)
    Seed Extension(read i)
end
for i = 1 to n do
    Output Generation(read i)
end
End Function
```

2. *GPUSeed* has structures and variable naming that conflict with *BWA-MEM*, specifically structures related to the BWT and FM-index.
3. *GPUSeed* is written as a standalone program using the classic C paradigm of one main function calling various other functions, in this case, GPU kernels, printing the results and then exiting. As a result, the *GPUSeed* library does not have the capability of being called by another function/program and does not provide any method of retrieving the results once it finishes processing.

The solution to the first point is to utilize *NVCC* to compile the complete *BWA-MEM* project since this compiler has the capability of sending the part which is host code to regular C or C++ compilers such as *GCC* or *G++*. The *GPUSeed* library is compiled separately with *NVCC* creating an object file which is then linked to the *BWA-MEM* compilation. Before we can compile the two projects together, a few alterations are required, especially for *GPUSeed*, namely:

- Change the main seed function into a wrapper function that can be called from the outside.
- Place this wrapper function within a block of `extern "C"` so that the wrapper function's name is not mangled during linkage.
- Move all functions prototyping and definitions to a separate header file which can be included in *BWA-MEM*.
- Change naming of BWT and FM-index related structures and variables, so there are no conflicts once included in *BWA-MEM*.

After applying these changes and including the *GPUSeed* header file into the `bwamem.c` file, we move to compile *BWA-MEM* and link the object file of the GPU library. The first important discussion required in this integration is regarding the placement of the call to the GPU wrapper. Invoking *GPUSeed* from the code location of the CPU seed function will not yield any potential improvements as we run into the issue of overhead due to the calculation of one query at a time. The same issue was discussed in section 4.3 in Chapter 4 when we presented the batching scheme for *BWA-MEM* performing GPU alignment. To circumvent this issue, we can apply the same logic as in Algorithm 9, namely to change `worker1` of *BWA-MEM* to take a chunk from the batch of queries for which to compute the seeds on the GPU and then continue as before with the sequential chaining and alignment on a query by query basis until the chunk is completed. The second option, which was utilized in this case, is to call the *GPUSeed* function on the complete batch of queries provided to the function `mem_process_seqs` and then continue the flow of execution as before with `worker1` working on one query at a time. The adapted call graph of the new integrated program can be seen in Figure 5.1 which by comparison to Figure 4.1 from Chapter 4 it now includes the *GPUSeed* function performing the functionality of the `bwt_smem1` and `bwt_sa`. In Algorithm 10, the batching structure can be seen.

In order to retrieve the seeds from the new seeding function, we need to define some sort of data structure that can save seeds in a format that can be utilized by *BWA-MEM*. The default format used to save *SMEMs* in the baseline program uses an array of structures with the following fields:

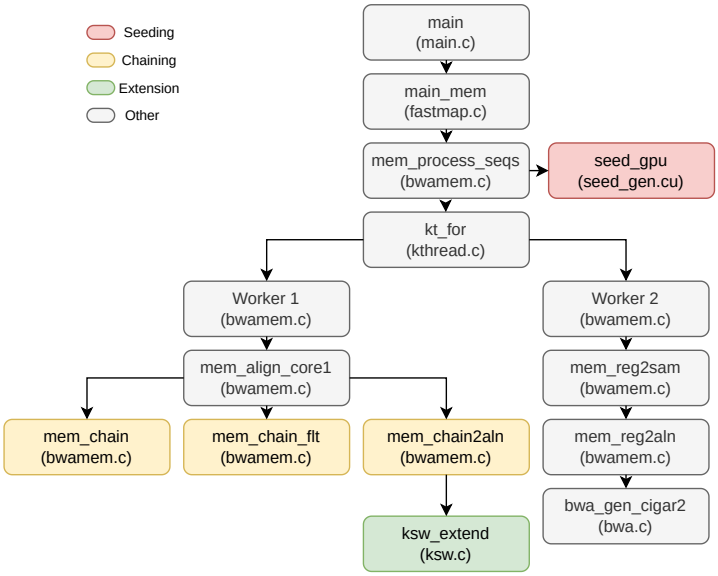


Figure 5.1: *BWA-MEM* with *GPUSeed* integration call graph.

1. The bi-interval of the *SMEM* in the form of an unsigned 64-bit array `x[3]` where each element holds I^l, $\overline{I^l}$ and I^s, as we saw in the subsection about the FMD index from 3.1.
2. A 64-bit value `info` which holds the starting position in the query and the length of the *SMEM*.

In the case of *GPUSeed*, the seed is already located in the text, so utilizing such a structure would be counterintuitive. By looking at the next stages in the `mem_chain` function, we notice that for the chaining of seeds, a data structure called `mem_seed_t` is used to hold the following information of a seed:

1. Reference start position, `Rbeg`, a 64-bit integer returned by the suffix array lookup function, holding the seed's starting position in the reference.
2. Query start position, `Qbeg`, which is a 32-bit integer holding the start position of the *SMEM* in the query sequence.
3. Seed length, `len`.
4. Seed score, `score` which is assigned the seed length.

In order to use this structure to retrieve seeds, we move its declaration from `bwamem.c` into the header file of *GPUSeed*. This method ensures that we can still use it in the baseline program during the chaining procedure, while it is also used in the GPU wrapper function to store the seeds. Furthermore, we expand the structure by defining a `mem_seed_v` vector of the previously presented seed type so that we can store together the complete set of *SMEMs* found for a specific query. The data structure solution is presented in Listing 5.1.

Listing 5.1: Data structure used for seeds.

```
typedef struct {
  int64_t rbeg;
  int32_t qbeg, len;
  int score;
} mem_seed_t;

typedef struct { size_t n, m; mem_seed_t *a; int seed_counter; } mem_seed_v;
```

Finally, we present the method of returning the *SMEMs* found on the GPU. Before every call to the *GPUSeed* wrapper function, we pre-allocate an array of n number of `mem_seed_v` vectors with n being the number of reads that will be processed. The array is in the form of a pointer `mem_seed_v*` and is

passed as an argument to the *GPUSeed* function. The main idea is to have one seed vector for each read in our batch, and each seed vector will hold one `mem_seed_t` element for each of its seeds. The next memory allocation we require is the number of `mem_seed_t` elements per query seed vector which cannot be known in advance, as the exact number of seeds that will be produced is known only after processing that query. The only information known at runtime is the maximum amount of *SMEMs* that can come out of a query of length $|P|$, which is $|P| - min\_seed\_length + 1$. Using this information, we could pre-allocate the maximum amount of seeds we expect from the batch passed to the *GPUSeed*. Assuming the basic batching scheme of *BWA-MEM* execution with one thread, then a batch of queries includes 10 million bases. If the queries are 250 bases long, then a batch sent to *GPUSeed* will have 40000 queries, which in turn can yield a maximum of 232×40000 seeds, assuming a minimum seed length of 19. The calculation of the amount of memory required to accommodate all these seeds is then found by multiplying the maximum number of seeds times the bits taken by our data structure. The main issue in this scenario is the unknown number of occurrences of each seed in the reference. This occurrence cannot be known in advance and is only found after calculating the intervals of the seed in the suffix array. As a result, the final solution implemented in *GPUSeed* would allocate memory for the seeds of each read at the end of the computation of *GPUSeed's* internal batch. It is crucial to remember that *GPUSeed* follows its internal batching scheme, which loads the reads corresponding to 1 million bases, then calculates the seeds for those queries and moves on to the next million bases until the whole larger batch received from *BWA-MEM* is completed. From this point forward, we will be referring to the batches of the GPU seeding function as internal *GPUSeed* batches so that we can distinguish them from the larger *BWA-MEM* batches. After each internal batch calculation, the seeds are transferred from the device back to the host in the following data structures:

1. `int2` array holding all the seed positions for all the queries, with the *x* coordinate holding the starting position and the `y` coordinate holding the end position.
2. `uint32_t` array holding all the corresponding positions in the reference.
3. `uint32_t` array holding the number of *SMEMs* per read in the internal batch.

Together with the total number of seeds found in our batch, these three arrays are all we require to store the *SMEMs* in the data structure of Listing 5.1. Our data structure is filled by looping over each processed query and allocating memory using the third array from the list above for the query's seed vector. In each of these iterations, we will have a nested loop going over all the indexes of the first two arrays from the list above that hold the seeds for that particular query. Each seed will be stored in its corresponding query seed vector as a `mem_seed_t` type. A critical remark must be made regarding the actual filling of the entries in the `mem_seed_t` struct and specifically for the seeds found in the reverse direction. Since *GPUSeed* uses the FM index, the reverse seeds are found with a negative position in the reference, and they require a small transformation so that their compatibility with the *BWA-MEM* pipeline is ensured. For the seeds in the reverse direction, we apply the transformation of Equation 5.1 to find their position in the forward direction.

$$Rbeg_{fow} = 2 \times length(bwt) - Rbeg_{rev} - length(seed) \tag{5.1}$$

5.1.2. Improved Integration

In the previous subsection, we presented a somewhat naive integration of *GPUSeed* into *BWA-MEM* intending to accelerate the seeding and seed locate phases by performing these tasks on a GPU. In this part, we aim to improve this first integration by resolving some of the remaining batching issues and restoring some of the previous functionalities of *BWA-MEM*, which were altered by the new seeding strategy. The summary of extensions that need to be implemented in the initial integration are:

- Adjust starting point of *GPUSeed* batches so that it loads its first batch starting with the first unprocessed query in the *BWA-MEM* batch.
- Adjust the *BWA-MEM* `mem_chain` function in order for it to work with the new seeds.
- Provide `worker1` the correct ranges of seeds so that it can perform chaining and alignment in parallel on multiple threads.

The first point from the list above refers to how *GPUSeed* loads the queries in each batch. The initial *GPUSeed* implementation was meant to work as a standalone function that finds the seeds for all the queries and then exits similarly to *BWA's fastmap* function. The standalone library would open the user's query file and start loading a batch of queries which would then be processed; subsequently, the following batch of queries would be loaded from the query file and so on. In the integrated implementation, the starting point of the GPU seeding library for loading queries should depend on which batch of reads coming from *BWA-MEM* is required to be processed. For example, we assume that for a dataset with 250 bases per query and single thread execution, *BWA-MEM* processes batches of 40000 reads at each batch iteration. The first call to *GPUSeed* will work as expected since the batch loading will start at the first query from the file. At the second call to *GPUSeed* when the batch being processed will be the one starting at query 40000 and up to 79999, the *GPUSeed* library should start loading its first internal batch at the correct read. Therefore, we need to provide the GPU wrapper function with additional information, such as the number of already processed queries and the number of queries to be processed. The former value and minor alterations to the internal batch loading function enable us to start loading at the correct query from the file. Note that for the time being, we let the seeding wrapper function continue reading the queries from the file instead of passing the reads to the function as an argument. In the optimization section, there will be a more extensive discussion regarding the choice of the query loading method for *GPUSeed*.

The second modification is made to the *BWA-MEM's* chaining function. The baseline implementation of `mem_chain` iterates over the seeds of a query to create the chains of seeds. During this process, it utilizes the size of the suffix array interval of each seed which is stored at `x[2]` to check whether a seed is too repetitive based on the user's request for the maximum occurrences of a seed, while if unspecified by the user it takes the default value which is 500. As a result, only the first 500 occurrences of each seed are located in the reference and then added to chains. In our current implementation, we have commented out the CPU `bwt_sa` function since we perform the same functionality using the locate GPU kernel. In addition, as it was discussed earlier during the seed data structure design, the seeds coming from *GPUSeed* are not in the format of suffix array intervals since we have already located them on the reference, meaning that the value for the size of the suffix array interval is not calculated in the GPU kernels and then retrieved. Therefore, during this initial implementation, we gave each seed a dummy value of 1 as its interval size, which will mean that all seeds will be included in the chaining consideration. In the optimization stage, a more accurate solution will be discussed.

Lastly, the ability to run chaining and extension in parallel using multiple instances of `worker1` is re-enabled by giving each worker thread the correct seed vector corresponding to the read it was assigned. In a nutshell, the workflow of the program follows a similar execution path as the baseline since the batching is done based on the number of threads given to the program by the user times the base chunk size of 10 million bases. The complete batch is then sent to *GPUSeed*, which finds all the seeds, which are afterward distributed on a query-by-query basis among the worker threads.

5.2. Complete Library Integration

This section presents the final step in the library integration process where we aim to merge the integration we presented of *GPUSeed* and *BWA-MEM* with the previously developed *GASAL2* and *BWA-MEM*. This process aims to run both seeding and extension on the GPU while keeping the two libraries separate for future maintainability. The following parts of this section will discuss the new compilation strategy, which is different compared to what was presented in 5.1.1, followed by the complete integration using the knowledge acquired throughout the naive *GPUSeed* integration as well as the analysis of Chapter 4.

5.2.1. Initial Compilation

The first step during the integration process was to study the *BWA-MEM/GASAL2* integration and compare it to the latest version of a baseline *BWA-MEM* implementation. This study aims to find out which auxiliary files are required to be brought up to date in *BWA-MEM/GASAL2* based on the latest commits of the *BWA-MEM* repository while keeping the changes and extra files required for the additional functionality. Since the *BWA-MEM/GASAL2* implementation has commented out much of the baseline functionality while keeping only the indexing and *MEM* alignment functionalities, there were minimal changes required only to the files associated with loading the FMD index for CPU seeding

and reading the queries from the files. Therefore, we concluded that these changes should not affect performance in the context of our implementation's scope. The complete integration can continue by taking the *BWA-MEM/GASAL2* as baseline and adding the *GPUSeed* functionality.

The initial compilation of the complete library was done by including the header file created for the GPU seeding wrapper function into the `bwamem.c` source file in a similar manner to what was discussed in 5.1.1. The goal is to compile the complete integrated project without changing any methods. The challenge compared to the compilation in 5.1.1 is that the complete project is now compiled using *G++* instead of *NVCC* and as a result, the *GPUSeed* library is required to produce an object file which the host linker can use instead of an object file that is taken by *NVCC*. The solution is to compile the *GPUSeed* library using the `-dlink` flag which links our device code into one object file which can be passed to *G++* in order to be linked to the *BWA-MEM/GASAL2* implementation. The compilation process of the complete project can be summarized in the following steps:

1. **GASAL2 compilation**: The first step is to compile the *GASAL2* library separately. This compilation requires the user to pass the maximum length of the query sequences and the code for representing the base *'n'*. As mentioned in the previous chapters, when presenting the features of this library, *GASAL2* pre-allocates memory during the program's initialization phase so that dynamic memory allocation during alignment is avoided as much as possible. Secondly, the *GASAL2* library uses *NVCC* for the compilation of its *CUDA* code while *G++* is used for all its C++ template functions. These templates are used for auxiliary operations such as launching asynchronous kernels, data structure filling, memory allocations and reallocations etc. The final library object file **libgasal.a** is linked using *G++* and is ready to be added to the *BWA-MEM* pipeline.

2. **GPUSeed compilation**: The second step is compiling the *GPUSeed* library using the method described earlier, the result of this compilation is the *libseed.a* object file which can be linked by *G++* in *BWA-MEM*

3. **Complete BWA-MEM compilation**: The final compilation of the integrated program using *G++* instead of the baseline *GCC* compilation as we are linking *CUDA* programs, C and C++ functions. Some modifications were required by placing *BWA-MEM* functions in blocks of `extern "C"`.

5.2.2. Integration

After the successful compilation of the complete project, we move to integrate the function that enables us to perform seeding on the GPU. Using the methods developed in 5.1 and leveraging the fact that the structure of the *BWA-MEM/GASAL2* program is mostly unchanged compared to the baseline *BWA-MEM* we integrate the *GPUSeed* wrapper function by making the following list of modifications to the function `mem_process_seqs`:

- Addition of a pointer of type `mem_seed_v` in the worker data structure field. The worker structure is passed to the `worker1` enabling us to pass the seeds to the alignment functions. The structure of the seed vector was presented in Listing 5.1.

- Memory allocation for the previously mentioned pointer of GPU seeds vector data. The allocation is done for n reads, where n is the large batch of reads passed to the `mem_process_seqs` function. This large batching is done based on the number of threads passed to the program multiplied by the number of reads corresponding to 10 million bases. The result of the allocation is n number of seed vectors `mem_seed_v`.

- Invocation of the `seed_gen` GPU wrapper function and passing as parameters the previously allocated pointer data structure, the name of the files of the reference and query, the minimum seed length and a flag telling the function whether to find *SMEMs* or *MEMs*.

The final changes are required within the `mem_align_core1` function that executes the seeding, chaining and the calls to the asynchronous *GASAL2* kernel executions. These changes are required to use the new results from the GPU seeds instead of performing seeding on the CPU. The following alterations are applied:

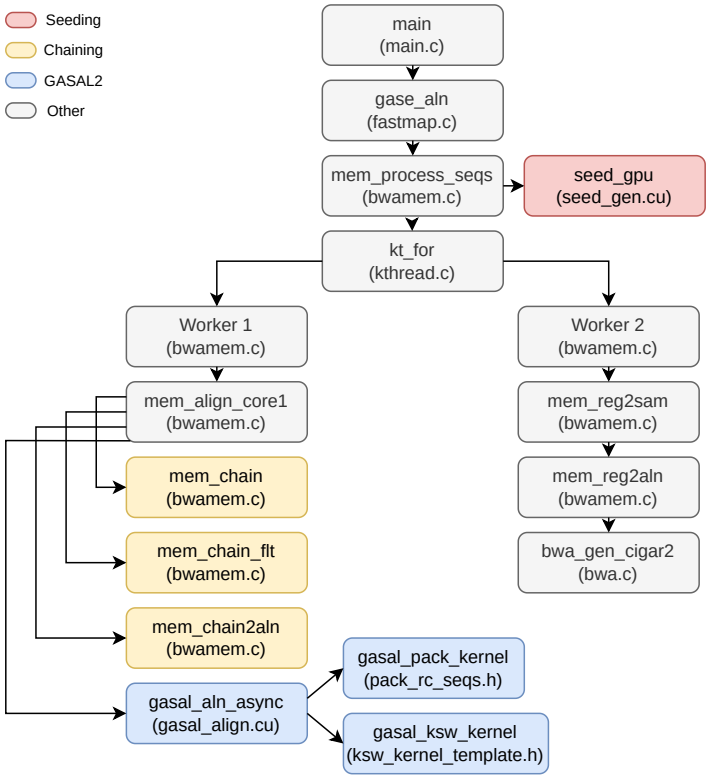


Figure 5.2: Complete *BWA-MEM*, *GPUSeed* and *GASAL2* integration call graph.

- The `mem_align_core1` is altered so that it receives the new data structures as function parameters.
- The `mem_collect_intv` function which is used for seeding is commented out in the `mem_chain` function.
- The `bwt_sa` function which locates a seed in the reference is commented out in the `mem_chain` main loop.
- The `mem_chain` function loops are changed to work with the new seeds structures similarly to what was mentioned in 5.1.2 regarding the suffix array interval of each seed.

In Figure 5.2, we present the integration call graph, using the similar simplified format of previous such graphs used throughout this report. Figure 5.2 shows in blue the asynchronous kernel calls to the *GASAL2* library which replaces the `ksw_extend` function from Figure 5.1. Furthermore, another important difference between the two call graphs is the change in the `mem_align_core1` pipeline, as the extension kernels are now called separately and not from the `mem_chain2aln` function, as was the case in the baseline *BWA-MEM/GPUSeed* integration. A difference that should be noted is that in this latest call graph, we tried to show the order of operations within the `mem_align_core1` by ordering them from top to bottom, while in the previous such graphs, we used a left to right convention. A last small difference of Figure 5.2 compared to Figure 5.1 and also the baseline graph in Figure 4.1 is the change in name of the `main_mem` function to `gase_aln`. This function's name now reflects the expanded capabilities required for pre-allocating memory for the *GASAL2* alignment structures while keeping all the previous capabilities of the baseline *BWA-MEM* implementation.

Lastly, in Algorithm 11, we present the pseudocode for the batching scheme of the integrated *BWA-MEM* running seed and extension on the GPU. The GPU seeding part works on the complete batch of n reads provided from the query file by the baseline batching functions of *BWA-MEM*. Internally, *GPUSeed* uses a looping strategy to work on batches smaller by a factor of 10 than n until the seeds of the complete batch of n reads are found. For simplicity, the internal batching of *GPUSeed* is omitted from the high-level batching pseudo code of Algorithm 11. After seeding, the execution continues

Algorithm 11: Batching Pseudo Code for complete integration.

```
   Input: a batch of n reads
   Output: n aligned reads
1  Seed Generation(all reads n)
2  for i = 1 to n/chunksize do
3  |  for j = 1 to chunksize/internal_chunksize do
4  |  |  for k = 1 to internal_chunksize do
5  |  |  |  Seed Chaining(read k + (j − 1) × internal_chunksize + (i − 1) × chunksize)
6  |  |  end
7  |  |  Seed Extension Kernel (all reads in chunk j)
8  |  end
9  end
10 for i = 1 to n do
11 |  Output Generation(read i)
12 end
13 End Function
```

with the chaining and extension parts which follow the batching scheme of *GASAL2* first introduced in Algorithm 9. The chaining is done on a query-by-query basis for 1000 queries (i.e., the internal chunk size) which are then all packed and extended using the asynchronous kernels. The next batch of 1000 is similarly processed until all 5000 (i.e., the worker chunk size) queries assigned to that specific worker instance are completed. The numbers presented here are the default parameters used in the current implementation of the program. Note that the outer loop at line 2 in Algorithm 11 in the actual program is the actual `kt_for` function which instantiates a `worker1` for each of the threads requested by the user. As a result, a chunk of 5000 reads is distributed to each host thread. A similar statement applies to the output generation loop, but in this case, chunks are not used, and the `kt_for` only distributes the reads among the available CPU threads.

5.2.3. Discussion

During this integration phase, the focus fell solely on creating the required framework to make the three libraries work together. In other words, most of the work revolved around creating data structures and interfaces between the various functions in order for the different modules to be able to cooperate. In addition, the batching scheme utilized kept the characteristics of both the baseline *BWA-MEM* which chooses the number of queries per batch based on the number of threads available for computation, as well as the *GASAL2* methodology of sending more than one query at a time for processing so that the GPU is used more efficiently. Before moving into the optimization phase, it is essential to discuss observations made during the initial testing of the complete integration. The observations are performance-related and accuracy-related, as these are the two main metrics that will guide the optimization process.

The performance observations were made based on testing the integrated program by performing alignments of simulated reads with 1 to 4 million sequences and varying read lengths against the GRCh37 human genome. The simulated reads utilized are cut-down versions of the queries presented in Table 4.1 of Chapter 4. The main reason behind using simulated reads in this early stage is that their uniform distribution of mutations yields very similar execution times between different batches. As a result, any discrepancies between the execution times of different batches raise a red flag about potential issues in our implementation. From the initial testing, the following points of discussion are of particular interest:

- The execution time per batch of queries seems to increase linearly as more batches are processed. As we mentioned, we expect the batches of queries to have highly similar execution times since they produce, on average, the same number of seeds and a similar number of alignments. This behavior was pinpointed to the *GPUSeed* part of the code as the integration running only extension on the GPU was performing within its normal execution parameters on the same datasets.

- The integrated library performed around 1.65 times faster than the baseline *BWA-MEM* implementation and around 1.25 times faster than the *BWA-MEM/GASAL2* program running only extension on the GPU.
- The integrated library performs, on average, 37.8% more extensions than the *BWA-MEM/GASAL2* on the same datasets. For example, the number of extra alignments for the 4 million query dataset with 150 bases per query is around 9 million for the integrated library compared to the *BWA-MEM/GASAL2*.
- The *GPUSeed* function was built as a standalone *SMEM* detection library similarly to the `fastmap` function included in the *BWA* package. As a result, the performance of *GPUSeed* was not tuned for multiple executions on batches of data and was mostly tuned for a single execution and then printing the results.
- The placement of the *GPUSeed* library in the pipeline should be investigated further. A simple approach would be to measure the total time spent in this function during the *BWA-MEM* integrated pipeline and compare that time to the execution time of a standalone version of *GPUSeed*.

On the other hand, to draw some conclusions regarding the accuracy of the integrated library, we utilized real scaled-down datasets of queries taken from Table 4.1 of Chapter 4. The reason behind the use of parts of SRR921889 and SRR835433 instead of the simulated reads is the much greater difficulty in seed detection and alignment due to the realistic nature of queries. Based on the initial testing, the following remarks can be made regarding the accuracy of this first complete integration:

- As first discussed during the result evaluation of the *GPUSeed* analysis in 4.2, the GPU seeding library produces many redundant *MEMs* and *SMEMs* with different locations in the reference. These incorrect seeds end up producing incorrect mappings of queries to the reference. In most cases, we see similar alignments as far as the *CIGAR* strings are concerned, but the query mapping is done to a different chromosome. This phenomenon is a direct result of finding a *SMEM* that maps to multiple locations in the reference and extending it at all the locations.
- A fundamental assumption made by the *GASAL2* library is that all the alignments made for all seeds in a query should cover 85% of the length of the query [59]. In other words, the seeds of a query are taken one by one within the function `chain2aln` and are added to the batch of seeds to be aligned only in the case the estimates of the seed's coverage reach 85%. This assumption is required because the batching strategy has no method of checking retroactively if a seed is already contained in an alignment since all seeds of a query are sent for processing in parallel. Unfortunately, this assumption leads to extra alignments over the baseline *BWA-MEM* implementation and includes many redundant seeds that should not be aligned. Removing redundant *MEMs* should enable us to match the results of the *BWA-MEM/GASAL2* integration.
- Small differences in the final alignment are expected due to the seed-only score used for both left and right alignments as was discussed in Chapter 3 under the *GASAL2* section.

This discussion aimed at pinpointing some of the integrated library's issues regarding performance and accuracy. Due to the nature of the seed and extend paradigm, we find many issues that affect each other and, in turn, affect both performance and accuracy. An example of such a case is the extra seeds leading to more alignments than necessary and incorrect mappings. At the same time, due to the acceleration architecture and the large amount of data that is required to be processed, we need to make some concessions such as the seeds coverage and alignment starting score in order to efficiently use our hardware resources and achieve good improvements over the base *BWA-MEM* program. The remarks presented in the context of the initial integration enable us to move into the optimization phase with clear priorities regarding the first improvements that should be looked at.

5.3. Library Optimization

The structure of this optimization section will follow the steps that were taken during the project in order to improve the integrated *BWA-MEM* library running seed and extension on GPU through the respective *GPUSeed* and *GASAL2* libraries.

The starting point of our improvements is the observations made during the initial integration testing in the previous section's discussion. First, we will look at the redundant operations still being performed as part of the *GPUSeed* pipeline. Secondly, we aim to improve the data transfers between host and device for both the loading and retrieval of data. The third optimization aims to improve the quality of the seeds and reduce redundant alignment computation through the chaining stage. The final improvement is a direct result of the implementation of improved quality *SMEMs*, which results in an architecture that requires more GPU memory than before, limiting the number of threads we can use for our program.

5.3.1. Redundant Operations

As we mentioned earlier, *GPUSeed* was built as a standalone *(S)MEM* detection library that takes as inputs the FM index of the reference and the query file and works on batches of queries while keeping the index in the global memory of the GPU. In the integration that was created, the actual call to the *GPUSeed* wrapper function is made for each batch of n reads. The multiple calls to the GPU seeding function instead of the single call for which it was initially designed result in redundant operations for each new *GPUSeed* invocation. In order to visualize these redundant operations, we used NVIDIA's Nsight Systems tool to create a profile and a visual timeline of the typical execution of the program using the complete human genome as a reference and a query file of 1 million reads with 150bp per query taken from the second entry of Table 4.1. In addition, running the program with the Nsight Systems from the command line provides at the end of a run a summary with statistics regarding the program run. In our case, we considered the following statistics to provide valuable information about the execution profile of our application:

- ***CUDA* API statistics** displaying the number of calls and the total time spent on each API function call.
- ***CUDA* kernel statistics** showing the total time, the number of instances, and minimum and maximum runtimes for each kernel in our program. Another important statistic in this category is the percentage of time spent in a kernel out of the total time spent in all kernels.
- **NVTX statistics** in the case we have defined push-pop ranges within our code.
- ***CUDA* memory operation statistics** indicating total time, the number of operations and the percentage of time spent on the type of memory operations out of the total time spent on all memory operations. Other important statistics which can be found here are the total, average, minimum and maximum KBs transferred for each type of memory operation.

In Figure 5.3 we present a merge of the Nsight Systems timeline for the complete dataset execution on the upper part, and the lower part is zoomed in so the execution of the first two batches of queries can be seen more clearly. It is important to mention that the batches have a size of 66668 reads each, as *BWA-MEM* will take the reads corresponding to around 10 million bases per batch times the number of threads requested by the user. The dataset has 150 bases per query, and we utilize one thread for the execution resulting in batches of 66668 queries. The complete execution of the 1 million query file will include 15 different batches of queries. From Figure 5.3 we can see that the program performs several redundant CPU functions calls in each of the 15 invocations to the *GPUSeed* which should only be performed once:

1. The function `bwt_restore_bwt_gpu` reading the BWT from the file and loading it into host-side variables. The size of the *GPUSeed* BWT is around 1.6GB.
2. The function `bwt_restore_sa_gpu` reading the suffix array from the file and loading it into host-side variables. The size of the suffix array of the *GPUSeed* FM-index is 1.8GB.

As it can be seen in Figure 5.3 where we defined the ranges of these two functions in grey using NVTX, the performance hit of performing these operations in each batch execution is very high and inconsistent since it is a file I/O operation. The solution is to detach them from the *GPUSeed* pipeline by calling both functions only once during the initialization phase of our program in the `gase_aln` function. The next step is to adapt our complete program pipeline so that we can pass a pointer to the structure that holds the FM index to the *GPUSeed* wrapper function and free the structure's allocated

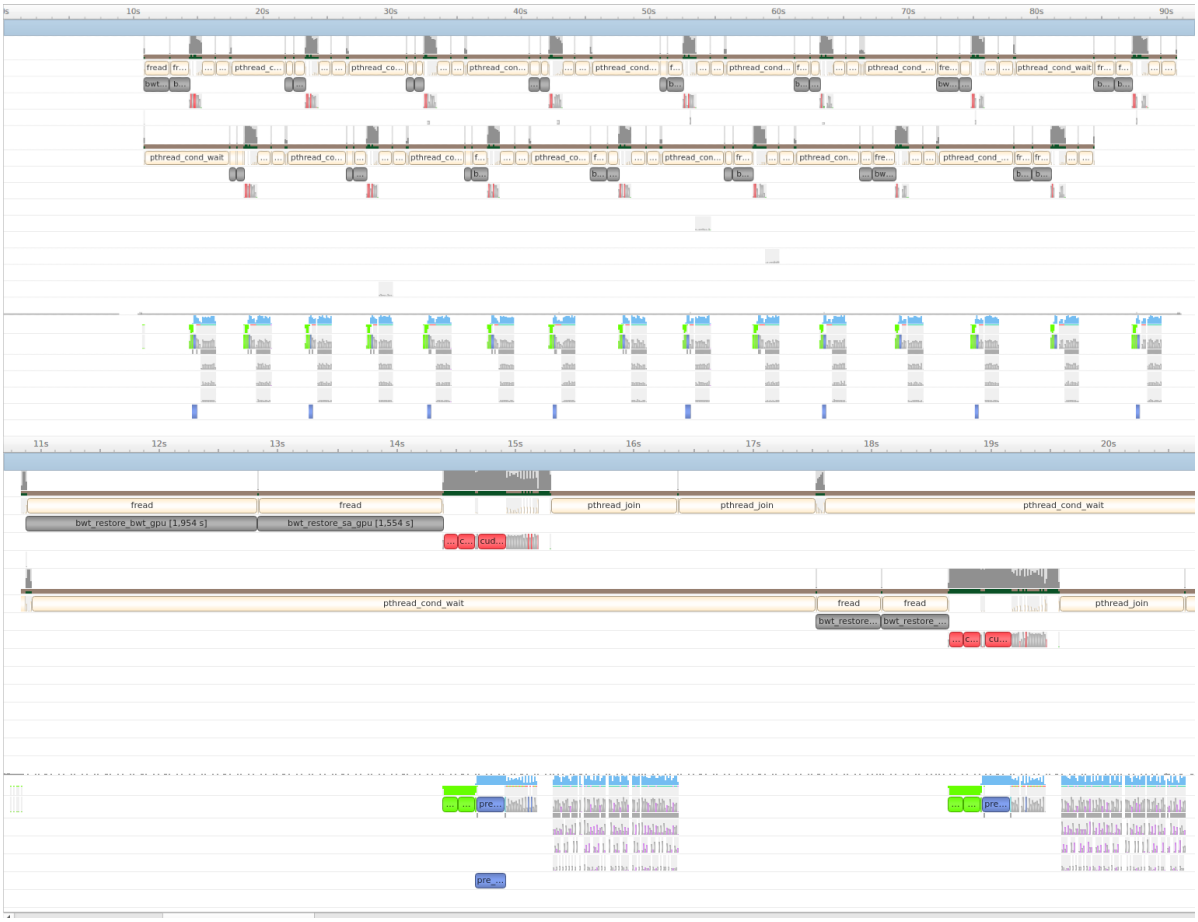


Figure 5.3: Integrated library Nsight Systems timelines.

memory at the end of the complete dataset execution. In addition, we can also remove the function calls of *BWA-MEM*, which load the FMD index utilized for seeding in the baseline CPU program, as these are not required anymore.

The final step in optimizing the redundant operations is to look at the statistics provided in the command line by the Nsight Systems tool and make observations for each of the categories of interest we presented earlier.

- In the *CUDA* API statistics category, we find that 46% of the total API time is spent in the `cudaDeviceSynchronize` function, which is used to block the host execution after each kernel call in our *GPUSeed* pipeline. *CUDA* calls are non-blocking in the sense that if we call a kernel or a memory operation, the hosts' thread can continue the execution of the code after the *CUDA* call without waiting for the GPU operation to be completed. In the case of *GPUSeed*, the call to `cudaDeviceSynchronize` after each kernel invocation is used as a blocking mechanism so that we can measure the time spent in that specific kernel using CPU functions that otherwise would not measure kernel time correctly. Also, it is important to mention that *CUDA* operations issued in the same stream are actually serialized and do not require any synchronization, and in our case, since there is no stream defined, all operations of *GPUSeed* are performed in the default stream. In conclusion, the calls to `cudaDeviceSynchronize` have an insignificant impact on the performance of the GPU seeding function as the blocked host code does not include any significant computations. The second most called API function in this category is the `cudaMemcpy` which is used for memory transfer and takes 31% of the total API call time.

- In the kernel statistics section, we find the most important statistics about our program: the kernels that run on the GPU and perform the various accelerated tasks required for our algorithm. In this category, we find the most dominant stage to be the `gasal_ksw_kernel` that performs the

alignment and takes 68% of the total time spent on GPU kernels. The second most used kernel is the `pre_calc_seed_intervals_gpu` which takes 18% out of the total. This last kernel has 15 instances, executed once for every batch of data passed to *GPUSeed*. This kernel is independent of the current batch queries as it calculates all possible suffix array intervals for all possible sequence combinations for a pre-defined length, in our case, length 13, for a given reference. In other words, this kernel can run only once, and its results can be used in all the *GPUSeed* runs. The third kernel with a portion of 11.5% portion out of the total kernel time is the `find_seed_intervals_gpu` that is called 150 times (i.e., 15 batches multiplied by ten internal *GPUseed* batches). The remaining kernels of both the *GPUSeed* and *GASAL2* libraries take all together 2.5% of the total time spent on kernel time with one of the *CUB* segmented radix sort and the `locate_seeds_gpu` kernels taking together 1.8%.

- The last category of statistics utilized is the *CUDA* memory operations statistics, where we find that 99.1% of memory operations are transfers from the host to the device, while only 0.8% of transfers are from the GPU to the CPU. The amount of data transferred from the CPU to the GPU is around 50GB which immediately raises a red flag as our query file is around 200MB, and the total FM index is 3.4GB. Due to the initial structure of *GPUSeed*, the program moves the FM index in every new invocation and frees it after completing the batch. This particularity can also be seen in Figure 5.3 with the repeating bright green segments on the upper and lower timeline.

Based on these observations, we move to remove the final redundant operations. First of all, the synchronization calls within the *GPUSeed* function are removed as we can now use Nsight Systems to monitor the profile of the library, and we add a time measurement around the *GPUSeed* function call in order to monitor the total time spend in finding seeds on the GPU.

Secondly, we modify the call to the `pre_calc_seed_intervals_gpu` so that this kernel is only executed during processing the first batch of queries, and then its result is stored and reused in all the rest of the GPU seeding batches. There are multiple ways to store the result of the interval pre-calculation. The first method keeps the device pointer associated with the results of this kernel and passes this pointer to subsequent *GPUSeed* invocations. This method is very efficient from the point of view of additional overhead but does take up more of the GPU's global memory as the pre-calculated intervals take around 500MB for the complete human genome with a pre-calculation length of 13. The second method transfers the results back to a host pointer used in the remaining *GPUSeed* calls to pass the pre-calculated intervals back to the GPU to be used for the calculation of the current batch. This method creates additional overhead associated with more memory transfers but does not take up additional GPU storage when the *GPUSeed* function is not running. The third method we can use is writing the results to a file, enabling us to use the results across different program executions. This method would save RAM (Random-access memory) as we load the pre-calculations in each seeding function call. At the same time, this last approach introduces even more memory overhead. Ultimately, it was decided to keep the results in the GPU's global memory and free that portion of memory only at the end of the program's execution during the de-allocation phase.

The third and last optimization removes the memory transfers associated with passing the FM index from the host to the device in every *GPUSeed* call. This is achieved by detaching these transfers from the initialization phase of *GPUSeed* and adding them to the initialization of the complete program in the `gase_aln` function. Finally, we build a copy wrapper function that takes the previously allocated host pointer for the FM index, which has been read from the file, allocates the required GPU memory and copies the FM index to the GPU. The FM index is then kept on the GPU memory for the program's complete execution, taking around 3.4GB out of the GPU's global memory.

Discussion

In Figure 5.4, we present the execution time for our improved integration after the previously mentioned optimizations. The total execution time is broken down into the most critical portions of the program, which follow the portions defined during the analysis in Chapter 4. In Figure 5.4, we see the initial integration, while we also present the execution times of our program after removing the redundant operations. The datasets used for this experiment are a simulated query file with 1 million reads each with 150 bases per read, and as a reference, we used the GRCh37 genome. The total execution time for this dataset has decreased by almost 50% mainly due to the improved seeding function, which now spends around 7.5 times less time performing *SMEM* detection. Furthermore, we notice a decrease

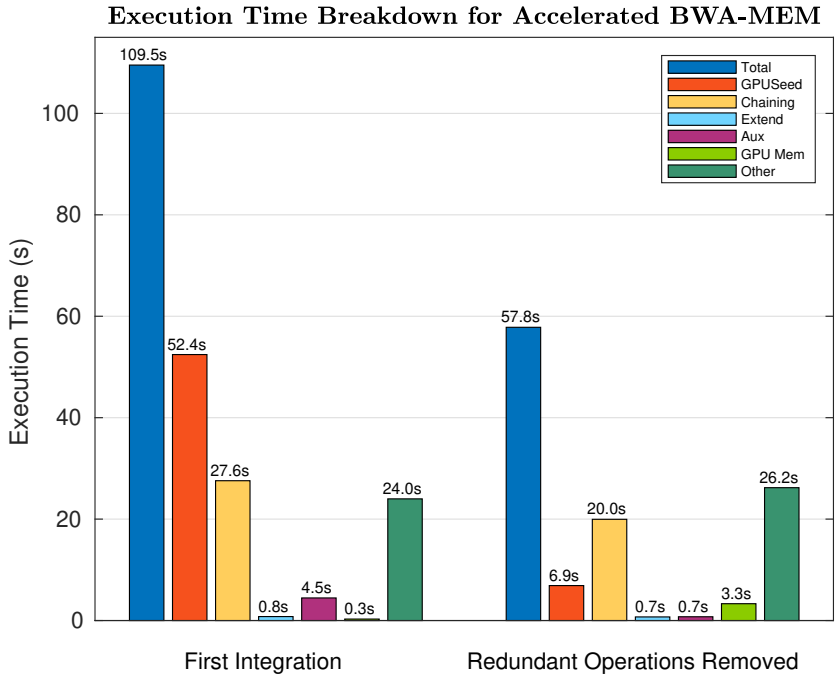


Figure 5.4: Execution breakdown for 1M reads with 150bp on GRCh37 before and after removal of redundant operations.

of around 3 seconds to the auxiliary operations, which is a direct result of removing the *BWA-MEM* functions that read the FMD index required for CPU seeding. Finally, it is essential to mention that the time we represent as extension time is the visible kernel time, not the total time spent in the extension kernel. As we mentioned, the visible kernel time is more representative of our pipeline's execution time context.

Finally, it is important to mention that the improvements in keeping the FM index and the pre-calculated intervals on the GPU memory during the complete computation have reduced the number of maximum threads our program can use. In Figure 4.10 we saw how the *GASAL2* library uses more memory with the increasing number of threads, and in the integration we have, it follows precisely the same memory allocation model while we also have to accommodate around 4GB on the device for the FM index used by *GPUSeed*. In addition, we also require device memory for each *GPUSeed* internal batch calculation. Looking at Figure 4.10, we would expect 18 threads to work with a complete human genome, but the program gave a GPU out-of-memory message when allocating memory for the seeding pipeline. Our experiments found that the current implementation can work on a maximum of 16 threads assuming our device is the NVIDIA RTX 2080Ti with 11GB of VRAM.

5.3.2. Memory Optimizations

This section aims to improve our implementation by exploring some memory optimizations. The focus falls on three categories, with two focusing on the input to the *GPUSeed* while the last one looks at optimizing the output of the seeding pipeline. The processes which we would like to improve are the following:

1. Improved query loading as it was noticed that the current solution, which uses the number of already processed reads received from the *BWA-MEM*, introduces significant overhead. In the current implementation, we pass the number of already processed reads and the number of reads to be processed for every invocation of *GPUSeed*. These two values are then utilized to load the correct queries from the file by starting to load reads from the line corresponding to the first read after the last already processed and up to the total reads, we should process in that batch. The overhead increases linearly since we will have to count more processed reads with every new batch of queries we need to load. This overhead becomes very noticeable, especially when dealing with larger query files.

2. Memory transfers between host and device by utilizing pinned memory instead of the current

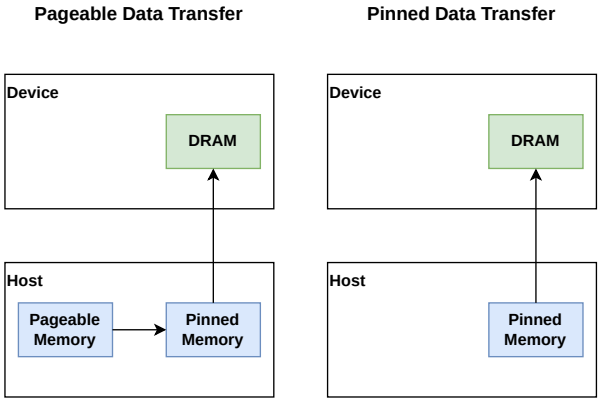


Figure 5.5: Pageable data transfer vs Pinned data transfer adapted from [69].

implementation with pageable memory [69]. By default, the memory allocations made by the host are pageable, and the device cannot directly access pageable memory, so there is an extra step required for every data transfer from pageable host memory to the device memory. This extra step involves transferring the data to a page-locked or pinned memory host array, allocated by the *CUDA* driver. The data can now be transferred to the device from this staging area allocated by the *CUDA* driver. In Figure 5.5, we present an illustration of the pageable data transfer steps, which was adapted from [69]. The intermediate step of moving data from the pageable memory to pinned memory can be skipped if we allocate from the beginning pinned host memory using the *CUDA* API functions. Pinned memory enables faster memory transfers between host and device and possible concurrency between memory transfers, host activity and GPU kernels.

3. The seeds data structure introduced in Listing 5.1 requires additional code that is executed on the host in order to get filled with the seeds, which are computed on the GPU. We measured the overhead introduced by the two nested for loops required for the structure filling at the end of every *GPUSeed* internal batch, and we found it to be around 30ms on average in the case of simulated reads with a length of 150 bases. This overhead highly depends on the number of queries in the total batch and the number of seeds found for each internal batch. For example, the 30ms value we presented is the overhead per internal batch which should be multiplied by the number of internal batches to give a value for this overhead in the context of the execution time for the larger batch. Furthermore, the minimum seed length also plays a vital role as lower values yield more seeds, resulting in more time spent filling the structures. Similarly, in the scenario of a real query file, we noticed that for SRR921889 with 100 bases per query, we find, on average, almost double the seeds per query than in the case of SRR835433, which has queries of 250bp. This difference in the total seed number is caused by the quality of the queries, with SRR835433 having larger matching substrings.

The goal of the first optimization is to utilize a more efficient method of searching for the query where we need to begin our batching in each *GPUSeed* call. The first approach to this problem was to pass the query sequences to the seeding wrapper function, which was already loaded by the *BWA-MEM* batching functions. Unfortunately, this solution resulted in inconsistent executions, with one run executing flawlessly while the next run with the same execution parameters would crash. Due to these bugs, it was decided to keep the functionality of *GPUSeed* reading directly from the query file, but we adapted the complete code so that we can pass a variable that counts the number of bytes from the file that has already been processed. As a result, using the `fseek` function, we can jump directly to the bytes, where we should start loading the first read without any additional searching. In the *GPUSeed* batch loading loop, we also count the number of bytes loaded as this value should be returned and then added to the accumulated sum of loaded bytes that should be passed to the next *GPUSeed* invocation.

The second optimization is applied to the loading of the FM index in the GPU memory. The functions `bwt_restore_bwt_gpu` and `bwt_restore_sa_gpu` are altered so that the allocate pinned host memory using the *CUDA* function `cudaMallocHost`. Furthermore, we adapt the function which frees the host's memory associated with the FM index after the successful transfer to the device so that it

uses the `cudaFreeHost` function. These changes result in an increase in transfer speed from 11.5 GB/s to 13.15 GB/s for the BWT and from 11.7 GB/s to 13.15 for the suffix array transfers of the GRCh37 human genome. Overall, we measure in some quick experiments that this change does not alter the amount of time spent in the GPU memory operations category as the decrease in transfer time is counterbalanced by the increased time spent in the `cudaMallocHost` over a regular *malloc* operation. Nevertheless, we consider this improvement important for future implementations where we will use larger FMD indexes or if the library is used to align larger genomes.

This third optimization removes the extra host-side computations performed at the end of every iteration of *GPUSeed*. These loops are required to fill our seed vector data structure. Our current implementation uses two nested loops, one going over the reads in that internal batch and the second going over the seeds corresponding to each read and filling the seed vector of each read. This implementation is altered so that we take advantage of our GPU when performing the transformation of Equation 5.1. The modifications that were performed in order to remove the looping strategy altogether are the following:

- Alterations in the seed data structure from the structure in Listing 5.1 to the new structure presented in Listing 5.2. This new data structure enables us to copy the seeds directly from the GPU into this structure without needing loops. The seeds are stored in single arrays without separating the seeds of different queries. As a result, we require an additional array holding the offsets that help us distinguish between seeds of different reads.

- Use of the `n_ref_pos_fow_rev_results` that holds the number of seeds for each query.

- Transformation of backward seeds using Equation 5.1 within a new GPU kernel, namely in `transform_seeds_gpu`.

- Addition of *CUB* `ExclusiveSum` call after all internal batches have been calculated in order to calculate the starting offsets of each read's seeds within the single arrays `rbeq` and `qbeg`. The exclusive sums are stored in the `n_ref_pos_fow_rev_prefix_sums` array.

- Alterations in the `mem_chain` function in order to accommodate the new structures.

Listing 5.2: Improved Data structure used for seeds

```
typedef struct {
  bwtint_t_gpu *rbeg;
  int2 *qbeg;
  uint32_t *score;
  uint32_t *n_ref_pos_fow_rev_results;
  uint32_t *n_ref_pos_fow_rev_prefix_sums;
} mem_seed_v_gpu;
```

Discussion

In Figure 5.6, we present the results of this section of improvements to our implementation. In order to show the differences, we used a query file with 4 million queries instead of the 1 million that was utilized in the previous experiment, as the optimizations done in this category require more batches of data to clearly see the improvements. In Figure 5.6, we also show the last optimization performed in the previous section 5.3.1, namely the removal of redundant operations to have a baseline for this round of optimizations. It is important to mention that the optimizations presented in Figure 5.6 are incremental in the sense that improved query loading includes all previous optimizations as well as the pinned memory transfers. The same statement applies to the improved seeds transfers improvement, which includes improved query loading, pinned memory and the optimizations of previous sections. The total execution time for this dataset reduces by 16.5% percent due to the improvement in loading the queries from the file, and we see a further 2.3s improvement in execution time when we add to the query loading optimization the improved seed transfers method. The addition of the improved seed transfer optimization shows a slight improvement in the execution time of the seeding stage, while the remaining stages have similar execution times compared to different implementations.

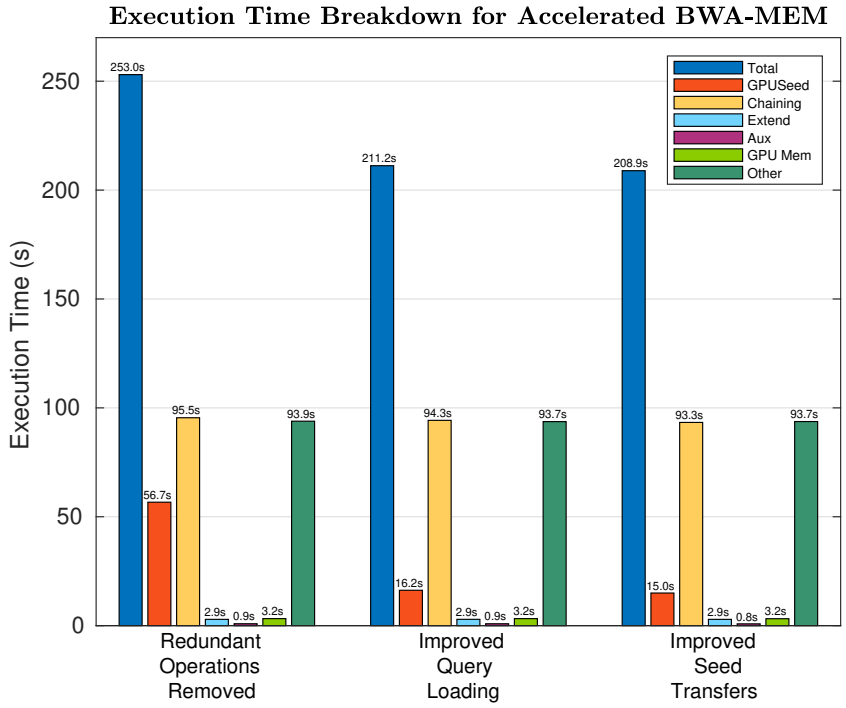


Figure 5.6: Execution breakdown for alignment of 4M reads with 150bp on GRCh37 with memory optimizations.

5.3.3. Improvement of Chaining and SMEMs

This section will explore methods that improve the quality of our alignment results. The end goal of this set of improvements that will be introduced in this section is to have a resulting *SAM* file that matches perfectly the output of *BWA-MEM* using the *GASAL2* library for alignment. Due to the heuristics introduced by the *GASAL2* GPU alignment, we expect to have the same final differences from the baseline *BWA-MEM* as those introduced by the *BWA-MEM/GASAL2* presented in [59]. The main reason behind the need to match the original implementation relates to one of the remarks made after we first presented the complete integration. Currently, the program performs around 37.8% more extensions, with some of these extra extensions being related to the chaining method that does not consider the suffix array interval size of each unique *SMEM*. As a result, we do not currently filter seeds based on their occurrence, and we end up with unnecessary seeds in the chains. The second remark we made was related to the quality of the seeds, which included many redundant *MEMs* when we were searching for *SMEMs* and also the fact that *GPUSeed* does not match the intrinsics of the *BWA-MEM* baseline seeding function. This last statement refers to the different ordering we have for the positions on the reference of the same *SMEM*. The second focus of this section will be to find the exact same *SMEMs* as in the baseline *BWA-MEM* which in turn should help us match the exact alignment results of *BWA-MEM/GASAL2*.

Improved Chaining

The first improvement that is added to the implementation in this section involves the improvement of the chaining stage in order to match the original *BWA-MEM* method while utilizing the data structures of our GPU seeding library. For this improvement, we require the number of occurrences of each seed in the reference, in other words, the size of the suffix array interval, which we defined as I^s. We expand the capability of the *GPUSeed* kernel `seeds_to_threads` in order to calculate this value for each *SMEM*. Furthermore, we return these values into the `score` array of Listing 5.2. As it was mentioned earlier, the baseline *BWA-MEM* chaining function receives a data structure that has the *SMEMs* in the form of suffix array intervals meaning that each *SMEM* is only included once, and its occurrences are determined by the interval that comes attached to it. In our case, each *SMEM* has already been found in the text, so each one comes together with the specific position in the reference, and its occurrences are simply separate entries of the same *SMEM* with a different reference position. The `mem_chain` function is adjusted so that the loops utilize the occurrences array `score` instead of the `x[2]` (both

represent I^s of a seed). From the initial integration and up until this point, we used `x[2]` set to 1, which included all the seeds independent of their number of occurrences. Similarly to the baseline function, the new `mem_chain` method can filter seeds with more than 500 occurrences.

Improving SMEMs using GPUSeed FMD
During the project, it was brought to our attention that there is an unpublished version of *GPUSeed*, which is on a private repository of a previous student who worked on this library intending to add support for the FMD index. This version called *GPUSeed FMD* has a very similar structure to the public version of *GPUSeed*, but it improves some critical components in order to bring the seeding results closer to what *BWA-MEM* produces. As we would like to have our seeds perfectly match the output of a baseline implementation, we took on the task of incorporating this newer GPU seeding implementation into our current, complete pipeline. This first part will explore how the solution from the master branch of the private repository is integrated into our complete library. The second part will look at how we found an even newer version named *GPUSeed FMD Fast Locate* on one of the numerous stale branches of the private repository and added those changes to this project as well.

First of all, we present the main differences of the *GPUSeed FMD* implementation with respect to the baseline *GPUSeed* presented in Chapter 3.2:

- *GPUSeed FMD* uses 64 bits instead of 32 bits for its BWT interval variable in a similar manner to *BWA-MEM*.
- The FMD GPU seeding implementation uses the bi-directional FMD index as it was presented in Chapter 3 instead of the FM index utilized by the public *GPUSeed*.
- The newer implementation employs a novelty as far as its suffix array is concerned since it uses 33 bits (i.e., 32 bits plus one packed bit). This suffix array differs from the basic *GPUSeed* and *BWA-MEM*, which use 32 and 64 bits, respectively. This suffix array implementation aims to reduce the total memory footprint by using only the necessary bits and also speeding up the suffix array lookups since it can use a smaller compression interval.
- **Stage 0 Pre-processing**: the packing of the queries is done only in the forward direction instead of in both directions.
- **Stage 1 Finding Suffix Array Intervals**: the algorithm is altered so that it only extends backward in the regular search fashion as we presented in Algorithm 1 in Chapter 3 using the forward packed queries.
- **Stage 2 Filtering (S)MEMs**: the filtering kernel is also altered as there are no more reverse seeds that require filtering based on their end coordinate on the query. This kernel is identical to the half that only filters based on the start coordinate, which was discussed in Chapter 3.2.
- **Stage 3 Split Suffix Array Intervals**: this kernel includes minor differences to accommodate the larger 64-bit intervals, but its functionality remains similar to the baseline *GPUSeed*.
- **Stage 4 Locate (S)MEMs**: this function remains the same as far as the main loop is concerned, while the only difference comes during the final position calculation where the upper bits of the suffix array are appended for the correct calculation of the position. Differences are found in the device function `bwt_inv_psi_gpu` called from this kernel which is very similar to the function that has the same purpose in the baseline *BWA-MEM*.

The FMD index used by the *GPUSeed FMD* implementation is quite significant as the occurrence interval for the BWT is every 64 bases resulting in 4.7GB for the BWT of the human genome we have been using. On the other hand, the suffix array is built using a compression ratio of 16, and each entry has 33 bits resulting in 1.6GB.

The integration of this new seeding library into our pipeline was seamless, as the names of most variables and kernels have remained the same. Furthermore, during the initial testing, this implementation had two bugs. The first issue was the lack of a check for the unknown base $'n'$ within the backward extension loop in the find seed intervals kernel. As a result, we would get a segmentation fault when the program tries to find the occurrence interval using the unknown base. A check was present in a

past commit of the library that would break out of the backward extension loop if an unknown base was encountered. The second issue of the program was an incorrect memory allocation for the *CUB* library `DeviceReduce` function which would result in the program exiting in the middle of execution without any error or message. The *CUB* library requires determining the temporary device storage by executing the function once, allocating the memory for the temporary storage pointer and then executing the function again to perform the actual *CUB* operation. After fixing this second bug, we focused on altering the macros used for extracting the occurrence intervals to reduce the size of the BWT by utilizing 128 base intervals instead of 64. The result was a BWT, which matches the one used by the latest baseline *BWA-MEM* implementation while keeping the characteristic suffix array of *GPUSeed FMD*. The FMD index is reduced to 3.1GB for the BWT while the suffix array remains the same unique 33-bit per entry with a 1.6GB total size.

The new integration was tested by printing the *SMEMs* to a file and then comparing them to the *SMEMs* produced by a baseline *BWA-MEM* program. The results for both simulated reads and actual queries were perfect matches finding precisely the same seeds. Furthermore, the alignment results match with runs of *BWA-MEM/GASAL2*.

Improving Performance using GPUSeed FMD Fast Locate

The final addition to our library is done from the same *GPUSeed FMD* repository, which included a version called *Fast Locate*. This version introduces several improvements to the *GPUSeed* to mainly improve the performance of the kernels that find the seed intervals. We analyze this version in a stage-by-stage manner in order to present the most important modifications from *GPUSeed FMD*:

- **Stage 0 Pre-processing**: the packing of the batch of queries is done similarly to the previous implementation. The `prepare_batch` which assigns *SMEMs* to thread numbers is removed.
- **Stage 1 Finding Suffix Array Intervals**: this kernel is split into forward and backward search kernels. The forward extension kernel will use one GPU thread for each query to perform forward extension by adding the complements of bases in an iterative manner. The logic behind the interval calculation is very similar to how the baseline *BWA-MEM* searches in the forward direction using Algorithm 5 presented in Chapter 3. The backward search kernel will be executed with the number of GPU threads equal to the number of seeds found in the forward search and should be extended in the backward direction. The backward search is similar to the previous implementations of `find_seed_intervals_gpu` which follow the logic of Algorithm 1 presented in Chapter 3.2. This type of implementation resolves the main issue presented in the original *GPUSeed* paper regarding the efficient parallelization of forward and backward extension [52]. This implementation also removes the need for pre-calculated seed intervals.
- **Stage 2 Filtering (S)MEMs**: the filtering stage remains unchanged from the *GPUSeed FMD*.
- **Stage 3 Split Suffix Array Intervals**: this stage also uses identical methods to *GPUSeed FMD*.
- **Stage 4 Locate (S)MEMs**: this kernel has been altered in order to utilize 4 GPU threads for each *SMEM*. The main logic remains the same with the difference in the addition of shuffle instructions required for the quick suffix array index exchange between the threads working on the same *SMEM*. Also, the `bwt_inv_psi_gpu` device function is brought back to the original version that uses pop counts as in the baseline *GPUSeed*.

Similarly to the previous integration of *GPUSeed FMD*, this program also included some bugs found by executing our integrated program using various query datasets. The first issue encountered was during our initial testing of this new integration, where we noticed that for some datasets, the program would exit without any warning or message in the middle of execution. Using the knowledge acquired during the previous integration where we saw a similar issue caused by one of the *CUB* functions, we quickly pinpointed the issue to a `DeviceReduce::Sum` function, which required more memory to be allocated for its storage buffer. The second bug encountered was *CUDA* raising an error message mentioning an illegal memory access during one of the GPU to CPU seeds array transfers. In order to fix this issue, we compiled the seeding library by passing the `-G` flag to the *NVCC* compiler and then ran our program using `CUDA-MEMCHECK`. The *CUDA* memory checking tool gave us the exact line in the `seeds_to_threads` kernel where we were writing to an out-of-bounds memory address for the

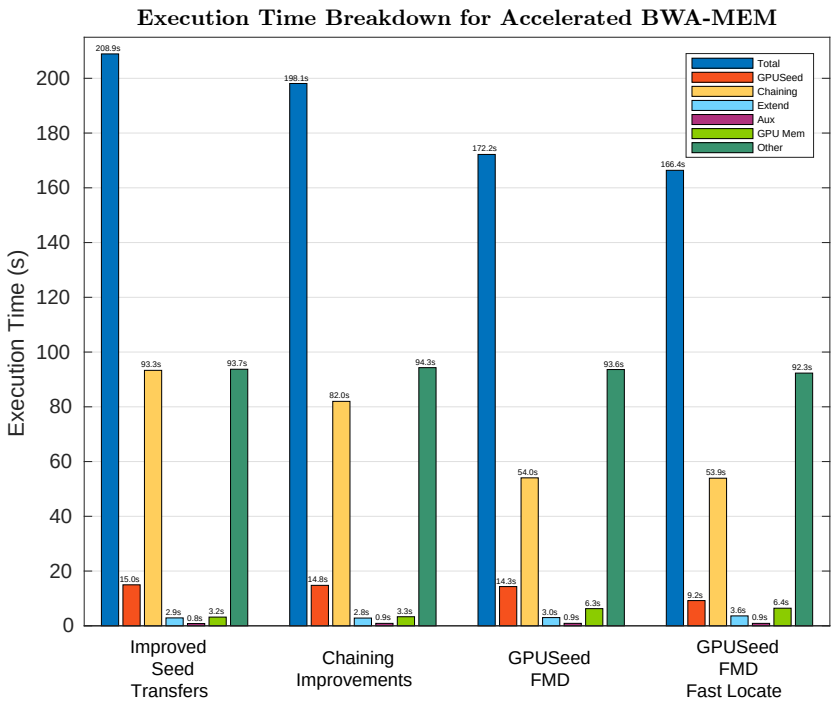


Figure 5.7: Execution breakdown for alignment of 4M reads with 150bp on GRCh37 with chaining and FMD optimizations.

array holding the suffix array index of each seed. Further investigation revealed that this array was not directly being allocated, but it was being passed a pointer that was being allocated. This last chunk of memory was altered to take two times less memory compared to previous versions of *GPUSeed*, and as a result, for some datasets where a seed had too many occurrences, there was not enough memory for the array holding the individual suffix array indexes. This issue was fixed by altering this initial *CUDA* allocation to allocate two times more memory.

Finally, we compared the *SMEMs* coming out of this altered seeding phase to *SMEMs* produced by the baseline *BWA-MEM*, finding the same results. Moreover, we verified the alignment results of this latest integrated program to the results produced by the *BWA-MEM/GASAL2* implementation, finding that the *SAM* files match perfectly.

Discussion

In Figure 5.7, we present grouped bar graphs with the breakdown in execution time of our integrated library with the optimizations performed in this section. The dataset utilized for this plot is the same 4 million query dataset with 150 bases per query, and the leftmost group of bars presents the execution time for the last optimization performed in the previous section 5.3.2 for comparison. The chaining optimizations that we performed decreased our total execution time by around 10s by reducing the time spent in the chaining function. Interestingly, the time spent in the *GPUSeed* function does not increase due to the extra occurrence calculation and memory transfer associated with the new array holding each *SMEM's* number of occurrences. For the optimizations of *GPUSeed* using the two versions with the FMD index, we notice that the execution time decreases even more, with the most significant decrease being in the chaining phase. Our earlier assumption regarding the lower amount of seeds leading to faster execution time on top of better alignment results is proven correct. In addition, for the *GPUSeed FMD* version, we see that the time spent in the seeding wrapper function remains the same compared to the baseline *GPUSeed*, even though the FMD version uses occurrence counting functions that are less optimized for GPU use. The *GPUSeed FMD Fast Locate* version has the best execution time overall, as it decreases the amount spent in GPU seeding by around 5 seconds over the simple FMD version. Finally, we see an increase in the time taken by GPU memory operations for both FMD versions due to the size of the FMD index being larger than the FM index used for the two leftmost versions shown in Figure 5.7.

5.3.4. Improved Pipeline Architecture for Multithreading

After the migration of *GPUSeed* to the version that utilizes the FMD index, the maximum number of threads has received a hit at the cost of improved *SMEM* accuracy. The implementation with the simple FM index could use 18 threads, and as we saw during the analysis chapter, the *GASAL2* library benefits greatly from the increased number of threads as it has the capability of using 2 GPU streams per thread. Our current program can use a maximum of 12 threads in combination with the NVIDIA RTX 2080Ti, as the FMD index takes around 4.7GB of GPU memory. The amount of GPU memory limits the maximum number of threads we can utilize since *GASAL2* pre-allocates memory based on the total number of streams. After the pre-allocation of memory required for the alignment phase, we copy the FMD index onto the GPU memory, where it will reside until the end of our program. Therefore, we present a list of points in our code where the program can run out of GPU memory and exit before completion.

- **Point 1**: During the pre-allocation of *GASAL2*, if we ask for too many threads. In Figure 4.10, one can find a good estimation of the maximum threads allocated in our current implementation and hardware platform.
- **Point 2**: During the BWT and suffix array GPU allocation. In combination with the previous point, we find that 14 threads of our current configuration will make the program exit at this point in the code.
- **Point 3**: During *GPUSeed*, if the internal batch allocation exceeds the available GPU memory or if the larger batch is too large and we cannot perform the exclusive sums required for the seeds offsets. At this point, there is no certainty for the second scenario since we do not know how many seeds will come out of a batch of queries; only a maximum can be roughly estimated. The current limit of 12 threads is imposed by the 120 million bases in each batch that, in combination with the previous points, is the maximum thread number that can be accommodated in *GPUSeed*.
- **Point 4**: *GASAL2* can allocate more memory on the fly if it is required for an alignment. During our testing, the program has never exited during such a reallocation.

From the list mentioned above, we conclude that we can improve the maximum number of threads that can be utilized by trying to detach the *GPUSeed* library from the batching pipeline of *BWA-MEM* and *GASAL2*. The main idea is to free our GPU of the 4.7GB taken by the FMD index over the complete program execution so that more *GASAL2* GPU streams can be allocated. The issue that arises by removing the *GPUSeed* call from the batching pipeline is how we can efficiently store the seeds. The first and most space-efficient option as far as RAM is concerned would be to write the seeds to the disk and only load the seeds corresponding to the currently processed queries in our pipeline. However, this method would introduce overhead associated with writing and reading the seeds to the file. The second option is to calculate all the seeds for all the queries and then keep them in RAM for the whole duration of our program. In order to see the viability of such a solution, we studied how much memory the seeds of the entire 50 million SRR921889 dataset take, and the result was around 21GB. Considering our platform's 128GB of RAM, we can safely assume that the seeds can be kept in memory for the duration of the program.

The improved implementation is created by moving the call to the *GPUSeed* library out of the main batching scheme of `mem_process_seqs` and into the `gase_aln` function. In Figure 5.8 we present this alteration that can be seen by the placement of the call to the `seed_gpu` wrapper function when compared to what we previously presented in Figure 5.2. In the call graph of Figure 5.8, we also show the *GASAL2* initialization function, which initializes the GPU streams and pre-allocates host and device memory for alignment. Although no alterations have been made to the placement of this function, we show it in this graph in order to demonstrate that our GPU seeding function runs before any of these initializations occur. As a result, the seeding function will calculate all the *SMEMs* for the complete query file and then keep the seeds in RAM. The device memory taken up by the FMD index is freed after *GPUSeed* finishes, and we move to allocate memory for the *GASAL2* structures with the full GPU memory available, as was the case when there was no GPU seeding. The improved integration will have during the batching iterations the same GPU memory utilization as we saw in Figure 4.10 of Chapter 4.

Finally, we summarize a list of the modifications performed to our application in order to accommodate this new architecture:

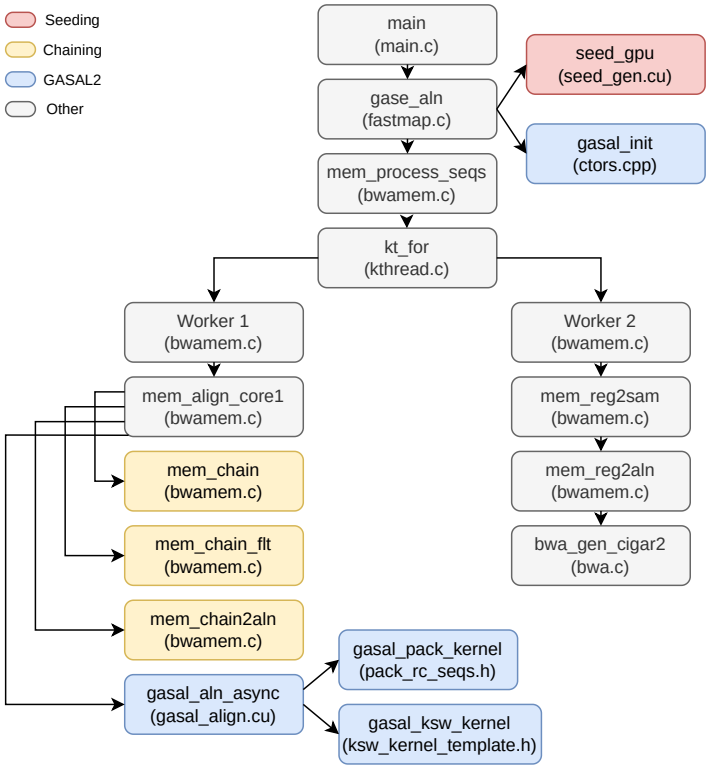


Figure 5.8: Improved *BWA-MEM*, *GPUSeed* and *GASAL2* integration call graph for better multithreading.

- Altered locations of FMD index loading and copying functions.
- Altered *GPUSeed* call location from the `mem_process_seqs` to the `gase_aln` function.
- Adjusted memory allocations within *GPUSeed* for the return structures holding the seeds.
- Added device memory de-allocation immediately after exiting *GPUSeed* instead of at the end of the program.
- Adjusted `mem_chain` function to work with the arrays holding the seeds for all queries, not only the ones in the batch.

Discussion
Since the latest architecture introduces changes that enable us to use more processing threads, we will compare the best execution scenario for the latest architecture to the best execution obtained for the last implementation of section 5.3.3. We compare the previous implementation from 5.3.3 using 12 threads to the latest implementation using 16 threads for execution. The dataset used for this experiment is the same 4 million queries with 150 bases per query. The total execution time is reduced by 18 seconds or 26.5%. By looking at the individual stages of the algorithm, we make the following observations:

- *GPUSeed* time is reduced by 1 second, meaning that the seeding is more efficiently performed in one go instead of in batches.
- The average visible extension time is reduced by around 1.5 seconds, benefiting from the assignment of extra threads (i.e., GPU streams).
- The time spent in chaining operations is the same even though we use more processing threads for these operations. This leads us to believe that the larger arrays used to hold the seeds for all the reads instead of a batch create additional memory access overhead in the chaining processes.
- The *other* operations and, more specifically, `worker2` benefit greatly from the additional threads, with the remaining 15.5 seconds reduction being gained from these functions.

To sum up, the increased number of threads helps us reduce the amount of time spent in the most dominant stage of the current implementation, which is, as we saw in Figure 5.7 the operations performed by `worker2`. Lastly, the seeding library performs marginally faster when detached from the batching pipeline, while the visible extension time also benefits from the extra processing threads.

5.3.5. Tuning GASAL2 Memory Allocations and Batching Sizes

During our initial testing of this latest implementation, it was noticed that for real datasets such as the SRR835433, many host and device memory reallocations were happening during the preparation of the alignment data structures for some of the batches. As a result, we decided to alter the coefficients used for the *GASAL2* pre-allocations to allocate more memory during the initialization phase so that we avoid having extra allocations during the extension phase. The new coefficients allocate 660MB of GPU memory per CPU thread instead of 484MB, and we found that for our datasets, the program can execute without requiring any extra reallocations during its extension pipeline. An important downside of allocating more memory is that the maximum number of threads we can use is reduced to 18 in the case of the RTX 2080Ti. Nevertheless, our preliminary testing showed that, in most cases, our current software implementation does not benefit from more than 16 threads.

In addition to improving the memory allocations of the *GASAL2* library, we also looked at the internal batching strategy used for the chaining and alignment pipeline. As mentioned, each `worker1` instance will receive a batch of 5000 reads and process them in chunks of 1000 reads at a time. In an effort to improve the GPU occupancy of the extension kernel, which was initially found to be around 10% for most *GASAL2* extension kernel instances, we tried to increase the number of reads that are being processed in each chunk. In addition to changing the chunk from 1000 to 5000 reads, we increased the number of reads passed to each `worker1` from 5000 to 20000 queries. As a result, our program will now pass 20000 queries to each `worker1` instance, where we will take chunks of 5000 reads for which we perform chaining and chain filtering before packing them and launching the short and long alignment kernels.

These changes improve the occupancy of the *GASAL2* packing and extension kernels from an average of 10% to 30%. Finally, we performed a quick experiment using the same 4 million queries file with 150 bases per read we have been utilizing throughout this chapter and compared the execution of the initial batching scheme executed with 1 and 16 threads to the execution time of the implementation employing the larger batches, again with 1 and 16 threads. For the single thread execution, we find marginally less time spent in the visible extension kernel time and a negligible decrease in the `chain2aln` function resulting in a similar overall execution time. On the other hand, for the scenario utilizing 16 threads, we find that the total execution time has decreased by 4 seconds, with the visible extension time decreasing around 0.5s and the remaining difference being caused by less time spent in `chain2aln` function.

6

Measurements & Performance

This chapter presents a study into the performance and accuracy of our integrated library. The integrated *BWA-MEM/GPUSeed/GASAL2* solution we presented in Chapter 5 is compared to the baseline *BWA-MEM* implementation in order to determine the improvement in execution time that is achieved by using a GPU to perform seeding and extension. The following sections of the report are divided in the following way. The first part presents the experimental setup, which consists of the hardware and software platform used for our experiments and the datasets used to benchmark our application. The second part will focus on the performance comparison of our integration to the baseline program, while the third part of this chapter will look at the differences in the alignment results between our implementation and the baseline *BWA-MEM* program. The final section of this chapter offers insights into the utilization of our compute resources by our accelerated *BWA-MEM* implementation.

6.1. Experimental Setup

This section presents the hardware platform used for our experiments and its configured software solution. Subsequently, we show the datasets used for the performance and accuracy study presented in the following sections of this chapter.

6.1.1. Hardware and Software Platforms

The compute cluster is provided by the Quantum and Computer Engineering department of the Delft University of Technology and includes several application and development servers. In our case, the *CUDA* development server is used for all our experiments. The hardware platform used for these experiments remains the same as in Chapter 4, ensuring that our results are consistent with our profiling analysis. In Table 6.1, the hardware specification of our compute platform is presented, while Table 6.2 shows the software specifications of the platform.

Platform	Hardware
CPU	2 Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz [63]
RAM	128GB DDR4
GPU	NVIDIA RTX 2080Ti [64]
VRAM	11GB GDDR6

Table 6.1: Hardware Platform.

The compilation of our integrated implementation is done by first compiling the *GASAL2* library using its default compilation settings, then compiling the *GPUSeed* library and finally compiling and linking our complete library using *G++*. The optimization level of the code compiled with *G++* is left at `-O2` while for the *NVCC* compiler we use `-O3`. This flag in *NVCC* aims to optimize host code, so in the case of *GPUSeed*, we do not see any improvements from adding such a flag as most of our code is running on the device. In addition, the *GASAL2* library requires at compilation time the maximum query length size

Software	Platform
Operating System	CentOS Linux release 7.9.2009
Linux Kernel	3.10.0-1160.71.1.el7.x86_64
C/C++ Compiler	g++ (GCC) 9.3.1 20200408 (Red Hat 9.3.1-2)
CUDA Version	11.0
NVIDIA Driver	470.82.01

Table 6.2: Software specification

and the code representation of the unknown base $'n'$ which should be `0x4E` if regular ASCII values are utilized in the query file or the integer four if the *BWA-MEM* convention is used. The maximum query length is required for the memory allocation of the complete row during the alignment matrix calculation so that there are no dynamic re-allocations that slow down the execution [59]. Furthermore, for *GASAL2* we utilize two streams per host thread while the *GPUSeed* library runs on the default *CUDA* stream.

The execution parameters for both the accelerated and baseline program are left at their defaults which are identical. It is important to mention that all experiments are executed with the same minimum seed length of 19 bases that we have used throughout this thesis.

6.1.2. Datasets

The datasets utilized for our implementation study are two of the complete real datasets used in Chapter 4 and one more dataset, which is added in this part of the study.

The first dataset, SRR921889, has 100 bases per query with 50 million queries. This query file is interesting because, as we saw during the analysis chapter, it maps poorly to the human genome and creates a particularly high strain on the seeding phase. The second dataset, SRR949537, is a two-paired end dataset with two reads for each query position. The first read in each position is the forward strand of the DNA, while the second read includes the reverse complement of the forward strand. In order to analyze this set of data, we take all the reverse complement reads and add them as separate queries at the end of the same file. The resulting dataset has around 10.4 million reads, with each query including 151 bases.The final query file utilized is SRR835433 [67]. This dataset is similar to the previous one in that it includes two-paired end queries. By applying the same concatenation as before, we add the reverse complement reads at the end of the file resulting in a dataset with around 16.6 million queries, each with 250 bases. Table 6.3 presents the query datasets utilized for the performance study.

Finally, the reference utilized in all experiments is the Genome Reference Consortium Human Build 37 (GRCh37/hg19) [56]. In addition, the FMD index of the reference is assumed to be created beforehand and is available on our platform's storage taking around 4.7GB of space. Similarly, the 2-bit packed reference used by the aligner is also created as part of the *BWA-MEM* indexing and takes around 800MB.

Query Dataset	Number of Queries	Bases/Query
SRR921889 [66]	50 Million	100
SRR949537 [70]	10.4 Million	151
SRR835433 [67]	16.6 Million	250

Table 6.3: Query datasets used for performance measurements.

6.2. Performance Measurements

The performance study of our integrated alignment solution will be presented in this section. For each of the datasets shown in Table 6.3, we will look at two crucial performance evaluations.

The first evaluation will focus on the single-threaded execution for both the accelerated and baseline *BWA-MEM* and present the execution time for fractions of code and the total runtime. This performance study is conducted to provide us with valuable information regarding the execution profile of our accelerated library compared to the original *BWA-MEM*. Furthermore, this study should give us a good idea

about future improvements and what parts of our application should be targeted for further optimizations. In order to create a more comparable result between the two different architectures, we alter the categories in which we break down our code. The following fractions are considered:

- **SMEM**: in this fraction, we include the time required for the baseline to find the *SMEMs* and also locate them in the reference using the suffix array locate function. On the other hand, in our accelerated implementation, we count as *SMEM* time the entire execution time of the *GPUSeed* library.
- **Chaining**: in this category, we include the amount of time required for building and filtering the chains of seeds. In addition, we include the time spent in the `chain2aln` function here. In the case of the baseline implementation, we exclude only the extension time from the `chain2aln` time, while for the accelerated library, we measure the entire time of `chain2aln` since the extension is not part of this function.
- **Extend**: we define the portion of time spent in the extension phase as the total time spent on the left and right alignment for the baseline implementation, while for the accelerated *BWA-MEM* we define the extension time as the visible time that the CPU is waiting for the results from the GPU extension.
- **Other**: in this category we include the time spent in the `worker2` functions for both implementations. In addition, for the accelerated program, we include some extra operations performed by `worker1`, which cannot be classified as chaining or extension.
- **Auxiliary**: in this portion, we include all the auxiliary operations performed by the programs. In the case of the baseline *BWA-MEM*, we include operations such as reading and loading the FMD index to memory, reading and loading the batch of queries from the file and memory allocations and de-allocations. For our accelerated program, we include the additional time required for GPU memory allocations, host-to-device memory transfers and freeing the GPU memory.

The second study will evaluate the performance of our accelerated program against the original *BWA-MEM* program using multiple threads. As we saw during the analysis in Chapter 4, the performance of the *BWA-MEM* pipeline scales very well with an increasing number of available threads, and in a real-life scenario *BWA-MEM* is always utilized using multiple threads. This study will provide insight into how well our implementation fares against one of the industry's most utilized CPU-based aligners.

6.2.1. Dataset: SRR921889

Execution Profile

In Figure 6.1, we present the execution breakdown for the SRR921889 dataset running on a single processing thread. This plot also shows the percentages out of the total taken by each stage defined earlier. In the case of the baseline *BWA-MEM* implementation, we find that almost 70% of the total time is taken by seeding and suffix array lookups. This dataset produces, on average, 21.1 seeds per query (for a minimum seed length of 19) which is almost twice the number of seeds compared to the rest of the datasets utilized in the performance study. The increased number of seeds results in large portions of time taken by the seeding and locate functions, with the locate function taking 60% of the *SMEM* portion seen in Figure 6.1.

On the other hand, the accelerated implementation performs the same seeding operations 170× faster, taking only around 95 seconds to complete. The most dominant stage in the accelerated code version is the chaining operation, especially the `chain2aln`, which takes 76.5% of this portion. This last function is responsible for filling the data structures required for the GPU alignment, and while it has more computational parts than the baseline version, we can hide some of its latency by overlapping its CPU execution with the GPU extension of the previous batch. As a result, we can see in Figure 6.1 that the total time spent in the chaining phase decreases going from the baseline to the accelerated implementation. The visible time performing GPU extensions is around 10 seconds for this dataset. Although this portion of time seems low, we manage to hide a lot of the latency associated with this stage by having asynchronous kernel launches and then immediately returning control to the CPU to work on the chaining operations of the next batch of queries. As a result, much of the time taken by the extension kernels is hidden in the chaining time of the next set of queries. At the same time, the

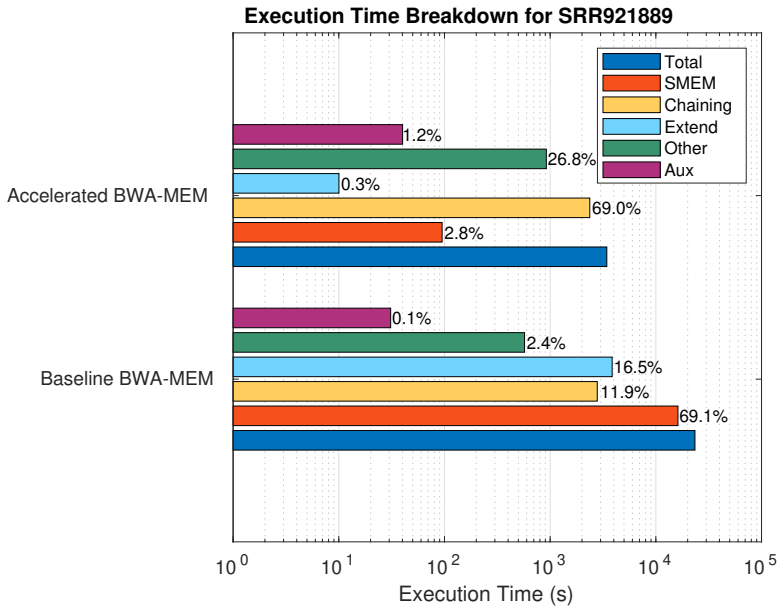


Figure 6.1: Execution time breakdown of Baseline vs. Accelerated *BWA-MEM* for SRR921889 for single thread execution.

extra operations required within `worker1` are reflected through an increase in the execution time of the *Other* operations, while the time spent in `worker2` for both programs is very similar. Finally, we see a 10-second increase in the auxiliary operations as it takes around 6 seconds to read and load the FMD index into the GPU memory, while the *GASAL2* initialization and complete GPU memory de-allocations take the remaining 4 seconds of the additional time.

Multithreaded Performance

The second study presented for the SRR921889 dataset focuses on comparing the accelerated against the non-accelerated library using multiple threads for their execution. In Figure 6.4a, we present the execution time for both libraries using between 1 and 18 threads while Figure 6.4b presents the speedup. In the single-threaded scenario, our accelerated library executes faster by 6.82×, which is very close to the theoretical maximum of 6.94×, assuming acceleration of the seed and extension phases. Furthermore, we find that for the accelerated library, the only negative effect of increasing the number of threads is on the *GASAL2* stream initialization and memory allocations and de-allocations. This happens due to the initialization of more streams since we have two streams per CPU thread, and also more memory needs to be allocated for those streams. In addition, we see a much better performance scaling of the baseline *BWA-MEM* when using more threads. This result is expected as the full seed and extension pipeline can run in parallel on different threads. On the other hand, in the accelerated version, we have seeding, which runs once on the GPU and always executes the same, regardless of the number of threads. The chaining and extension parts benefit from the extra threads since the chaining can run in parallel in a similar manner to the baseline, while the extension will use more GPU streams. Nevertheless, we see the speedup of our improved application go from 6.82× to 3.22× as the number of executing threads goes from 1 to 18. For the baseline application, we also used the maximum available threads of 24 and compared the result to the best execution of the accelerated library using 18 threads finding that our accelerated program runs 2.8× faster.

6.2.2. Dataset: SRR949537

Execution Profile

In Figure 6.2, we present the breakdown of the execution of both libraries for the SRR949537 dataset. The most dominant stage in the baseline version is again the detection of *SMEMs* with 65.7%, with the extension phase taking 21.7% of the time. For this dataset, we find that the reduced number of average seeds per read compared to the first dataset reduces the strain on the seeding portion of code. At the same time, the higher query length increases the time spent in the extension phase, which is in line with

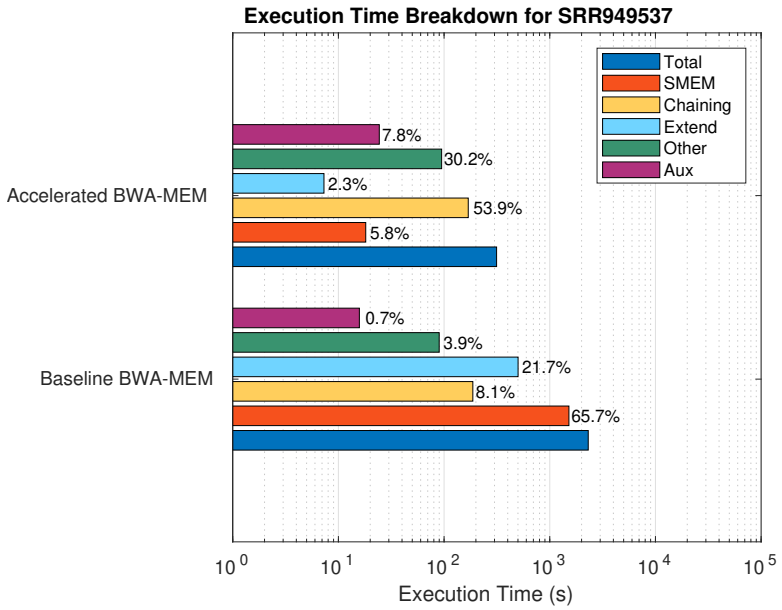


Figure 6.2: Execution time breakdown of Baseline vs Accelerated *BWA-MEM* for SRR949537 for single thread execution.

what was presented and discussed in Chapter 4. Additionally, the total time spent finding the seeds and detecting them in the reference is reduced by 83.7× going from the baseline to the accelerated program. The seeding in the accelerated program takes only 5.8% out of the total execution time. The visible time taken by the extension kernels of the accelerated program is around 7 seconds, while the chaining phases of both programs seem to take around the same amount of time. The most dominant portion in the accelerated program is again the chaining portion, with 70% of this portion taken by the `chain2aln` function. For the auxiliary operations, we see the same increase in time for the GPU and *GASAL2* memory operations as it was also seen for the previous set of queries. This extra auxiliary time should remain the same across datasets as it depends on the FMD index size and the number of executing threads. Finally, as far as the *Other* operations are concerned, we find in both programs the same amount of time spent for the `worker2` function, while in the case of the accelerated program, we have the additional operations performed in the context of `worker1` which amount to around seven extra seconds.

Multithreaded Performance

In Figure 6.5a we present the execution time of both programs for varying number of CPU threads, and in Figure 6.5b we show the speedup achieved by our accelerated implementation. First, we see that our accelerated program is 7.36× faster than the baseline *BWA-MEM* with the maximum theoretical speedup being 7.93× given the acceleration of the seeding and extensions phases of the program for this particular dataset. For this dataset, the results show an interesting behavior for the accelerated program when increasing the number of threads above 12. As seen in Figure 6.5a, the accelerated program starts losing performance if we increase the number of threads above 12, where the minimum execution time can be found for this dataset. By looking at the time spent in the various stages of the program, we can draw the following conclusions explaining this behavior:

- Above ten threads, the visible extension time starts to increase. In other words, our CPU has to wait more time for the results of the GPU extension. This happens because we have 24 streams with 2 for each thread and each of these threads launches asynchronous extension kernels, which must be prioritized as the GPU does not have the resources to execute them all concurrently. As a result, kernels waste time waiting to be taken into the execution pipeline of the GPU and the more threads we give to our program, the more streams we will have, resulting in more kernels queued for execution.
- Although we see the visible kernel time growing from 10 threads, our total execution time still decreases as we go from 10 to 12 threads, where we find our minimum. The reason behind

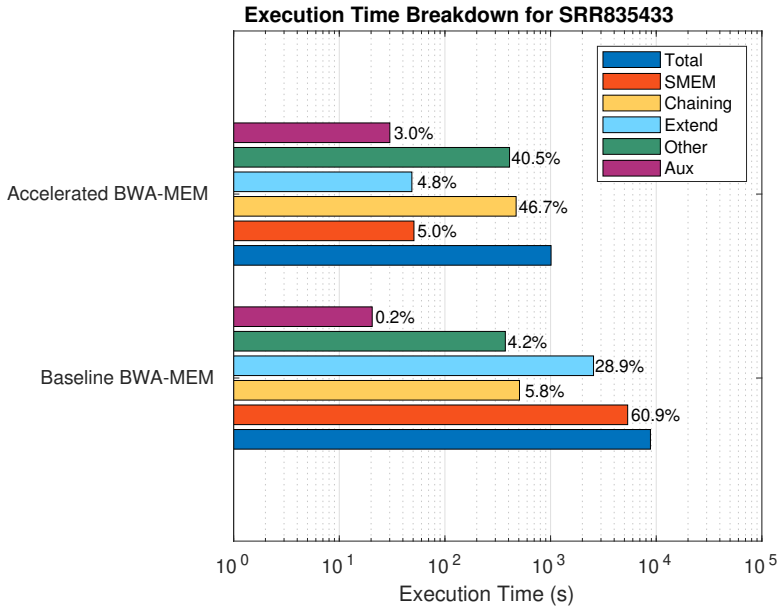


Figure 6.3: Execution time breakdown of Baseline vs Accelerated *BWA-MEM* for SRR835433 for single thread execution.

this behavior is that the time spent in our chaining functions keeps decreasing as we use more threads. As a result, the decrease in chaining time is more than enough to account for the increase in visible kernel time and then some. However, as we move above 12 threads, the decrease in chaining time is not enough to account for the increase in visible kernel time caused by what was discussed in the previous point, resulting in a worse overall execution time.

To sum up, for this dataset, we find that the portion of time taken by the extension is higher while the seeding portion is smaller compared to the first dataset we discussed. For the accelerated program, when we have more strain on the extension phase, we end up with a smaller performance gain than in the case where we have more seeding computation. The overall speedup for the best case execution scenario of both programs shows that the accelerated program is faster by 2× when using 12 threads against the best baseline program execution utilizing 22 threads.

6.2.3. Dataset: SRR835433

Execution Profile

We conclude the performance evaluation of our accelerated *BWA-MEM* library by presenting the last measurements taken with the SRR835433 dataset. In Figure 6.3, we present the execution time breakdown for the various segments we have defined in our code for both the accelerated and non-accelerated programs. Compared to the previous datasets, we note that the portion taken by extension increases even further while the *SMEM* fraction shrinks. Furthermore, the percentage for the visible extension portion increases and is almost equal to the time spent on finding seeds on the GPU. In addition, the chaining operations take up a smaller portion of the total execution time compared to runs with datasets with fewer bases per query. Based on the observations made for all three datasets, we can conclude that datasets with longer queries result in a higher percentage of time taken by extension. Nevertheless, the accelerated program spends almost half the time performing chaining operations.

Multithreaded Performance

Lastly, we present the study into the performance of the two applications using one and up to 18 threads for this final dataset. The total execution time for the different number of threads can be seen in Figure 6.6a while the resulting speedup is shown in Figure 6.6b. The accelerated program achieves 8.72× speedup over the baseline program, while the theoretical maximum given by Amdahl's law is 9.8×. Again, we note that a dataset that requires a higher portion of time to be spent on extension takes the total speedup further from the theoretical maximum. However, the maximum theoretical speedup

assumes infinite compute resources and no overhead, and in our case, the accelerated program can use the finite resources provided by the GPU while having the overhead introduced by our implementation. For this dataset, we also notice the negative impact on performance when increasing the number of threads above a specific number. In this scenario, above 14 threads, we see in Figure 6.6a that the total execution time of the accelerated program starts to increase. By looking at the individual execution time of portions of the program, we find that the total time spent in chaining decreases as we increase the number of threads. On the other hand, the visible extension kernel time decreases up to 10 threads and, at that point, starts to increase slowly. As a result, from 16 threads and onwards, the decrease in chaining time is not enough to balance out the increase in the visible extension time, which is caused by the same queue that we mentioned in the previous dataset measurements. Finally, we compare our best execution for the accelerated program using 16 threads to the best execution for the baseline *BWA-MEM*, which is achieved using 24 threads, and we find that the speedup of the accelerated program is 2.34×.

6.3. Alignment Results Comparison

In this section, the study focuses on the results that come out of our accelerated aligner. First, the accelerated library is executed using one thread for each of the datasets in Table 6.3, and the resulting *SAM* outputs are written to files. Then, the same analysis methodology is applied to the baseline *BWA-MEM*. For the comparison, we utilize the `diff` command in Linux to find the number of lines that do not match between the two files, and a percentage of the different lines out of the complete lines is calculated.

Before comparing the *SAM* files, we would like to verify that our implementation finds the exact same *SMEMs* as the baseline program for each dataset. In order to compare the *SMEMs*, we modify both programs to print the *SMEMs* found for each query to a file together with the location in the reference text as well as the number of occurrences in the reference. The resulting *SMEMs* are compared using the `diff` utility in Linux, and we find that for all three datasets, the accelerated library finds precisely the same *SMEMs*. The seeds are further verified by checking the number of alignments and the alignment results between the *BWA-MEM/GASAL2* implementation and the final accelerated program. The *BWA-MEM/GASAL2* uses the seeding and chaining of the baseline *BWA-MEM* program while it introduces the particular heuristics required for parallel alignment. This comparison will verify that our seeding and chaining operations perform identically to the baseline while we match the alignment operation of the *BWA-MEM/GASAL2* library. The results show that our integrated library performs the same number of alignments, and the *SAM* file matches exactly the output of *BWA-MEM/GASAL2*.

Since the *GASAL2* library has to make some compromises in order to accelerate the alignment process, we expect to have some differences from the baseline program. These differences are mainly caused by the seed-only paradigm used for the starting scores of both the left and right alignment and the seed filtering heuristics introduced when filling the alignment data structures. During the various runs of our program, it was noticed that the heuristics introduced by the *GASAL2* library manifested differences mainly in the secondary alignment scores, while in much fewer cases, we would see differences in the primary alignments. In addition, it was also noticed that many of the differences appeared in alignments that had a lower mapping quality. The lower mapping quality alignments are usually discarded in the downstream analysis as a lower mapping quality value indicates a higher probability that the read sequence was mapped incorrectly. The *BWA* software outputs a mapping quality in the range of 0 to 60, and we decided to look at a scenario where we remove alignments with mapping quality below 20. As a result, for the study of the *SAM* files, we create the following three scenarios:

- The first scenario compares the complete alignment outputs as they come out of the accelerated and baseline *BWA-MEM* implementations.
- The second scenario compares the two resulting *SAM* files after removing the secondary alignment scores.
- The third scenario compares the *SAM* files without secondary alignments and also with alignments with mapping quality below 20 removed.

The results of our analysis for all three datasets can be seen in Table 6.4. For the SRR921889 dataset, we find the lowest percentage of different lines for all three scenarios. For the SRR949537

Dataset	Complete SAM File	Secondary Scores Removed	MAPQ ≥ 20
SRR921889	2.48%	1.83%	0.46%
SRR949537	7.77%	1.98%	1.27%
SRR835433	8.85%	2.08%	1.17%

Table 6.4: *SAM* file differences compared to baseline *BWA-MEM*.

and SRR835433 datasets, we find that the complete *SAM* files have quite significant differences, with 7.77% and 8.85% of lines not matching in each respective case. Our assumption that the secondary scores cause the majority of differences is proven to be correct since the percentage of different lines decreases when we remove these scores, as can be seen in column 3 of Table 6.4. Furthermore, by removing the alignments with mapping quality below our threshold of 20, we end up with alignment differences of 0.46%, 1.27% and 1.17%, respectively, for each of our datasets.

An additional study is performed for each of the results of our datasets, where we remove alignments with mapping quality below a certain threshold, and we increase this threshold in increments of five. The results of this study are shown in Figures 6.7, 6.8 and 6.9 for each of our datasets. Each of these figures presents in blue the complete results of the alignment, while for the red data points, we removed the secondary scores. For the SRR921889 dataset, we notice that removing the secondary scores does not reduce the percentage of different lines between the baseline and the accelerated results as much as for the other two datasets, where we see a reduction of around 6%. In addition, for the SRR949537 and SRR835433, we see that removing alignments with mapping quality below 30 drops the percentage of different lines below 1%, and this percentage continues to drop linearly as the mapping quality threshold is increased.

It is essential to mention that our analysis will show two lines as being different if any of the fields of Table 3.5 have the slightest difference. In many cases, we saw alignment lines between the baseline and our implementation having the same *CIGAR* strings and mapping positions but minor differences in mapping quality. Due to our analysis methodology, we flag them as entirely different lines.

6.4. Resource Utilization

The final section of this chapter will present a study into how the compute resources are utilized by our accelerated *BWA-MEM* application. This study's primary goal is to provide clear limits imposed either by our software implementation or the hardware used to run the accelerated library.

6.4.1. Memory Utilization

RAM Utilization

The first vital resource for our implementation is memory, specifically the RAM attached to the CPU. Since the detachment of the *GPUSeed* from the *BWA-MEM* batching pipeline that was discussed in the final part of Chapter 5, the program now depends on the available RAM. The main reason is that we now have to keep all the seeds for all the reads in our RAM for the complete duration of our execution. As a result, when dealing with millions of queries that produce an unknown number of seeds, we need to have some estimates on how much RAM is required to process a specific dataset. Furthermore, after completing the seeding phase, the *GASAL2* library will allocate host memory for its data structures. The amount of RAM allocated by *GASAL2* depends on the number of streams utilized, which depends on the number of threads allocated to the application. After the *GASAL2* allocations, the program will go into the batching pipeline, where batches of queries will be loaded and then processed. In this stage, memory is needed for the batch of queries and the metadata that comes with those queries.

In order to measure the memory taken during these three phases of our program, we run an experiment using SRR921889, which is the largest dataset with 50 million queries, and it also produced the most seeds per read at around 21 compared to 12 and 10 for the other two datasets from Table 6.3. This experiment is executed using 1 and 18 threads as this will give us a memory usage estimation for the best and worst-case scenarios.

It is important to mention that for the *GASAL2* memory allocations, we leave the coefficients that estimate the amount of memory allocated to the values set in the last section of Chapter 5. These estimates will allocate the same amount of memory regardless of the dataset. The other option would

be to set the coefficients really low and let the program grow the memory it requires using its automatic memory extension. This method would enable us to see exactly how much memory is allocated by the *GASAL2* library for each dataset, but as we saw in Chapter 5, the automatic memory extension slows down the execution. In the end, we choose the case where we will allocate a bit more memory for the *GASAL2* structures since this is the case where we extract the most performance out of our software.

The memory utilization for each phase is found by adding `sleep` functions in our code before and after the execution of each of the three phases and then looking at the RAM utilization of our platform using the Linux `htop` command. From our experiments, we find the following:

- Memory used after *GPUSeed* and taken by the seeds is around 21.7GB and is independent of the number of threads we use for our application. This value is in line with our estimation using the data structure presented in Chapter 5 for holding the seeds and the total number of seeds found for the SRR921889 dataset.
- Memory used after the *GASAL2* allocations is 22.2GB in the single thread case, while for the case using 18 threads, we have 28.7GB allocated.
- Memory used within the batching pipeline is 22.8GB for the single thread execution, while in the case we use 18 threads, we use a total of 40.3GB.

From this experiment, we can conclude the maximum number of queries we can process using the current software implementation and hardware platform. Assuming that the queries will have a similar amount of average seeds as in our experiment, we expect to accommodate a maximum of 300 million queries for a single thread case, while for the execution using 18 threads, we can have a maximum of 250 million queries. These estimations are done on the basis that our platform has 128GB of RAM and the minimum seed length is 19. In the scenario, a larger minimum seed length is utilized the total number of seeds is expected to drop.

We can conclude that the maximum amount of queries that our current implementation can process is limited by the amount of RAM available. This limitation is imposed by our decision to keep the complete results of *GPUSeed* in memory during the complete execution of our program.

GPU Memory Utilization

The second type of memory that creates limitations for the execution of our program is the GPU memory, also referred to as VRAM. As shown in Figure 4.10 of Chapter 4, the amount of GPU memory that is available is the only limiting factor in how many threads we can use for our program since the *GASAL2* library uses two streams for each CPU thread, and these streams require us to pre-allocate memory on the GPU. Furthermore, in the *GASAL2* analysis of Chapter 4, we also saw that memory grows linearly with respect to the number of threads provided to the program to execute on.

At the same time, during the performance analysis of this chapter, it was also concluded that even though we are limited to 18 CPU threads in the case of a GPU with 11GB of VRAM, using more than a certain number of threads can harm our program's performance. Nevertheless, we would like to present figures regarding our program's GPU memory utilization. Specifically, we are interested in the GPU memory utilization by the *GPUSeed* pipeline and the subsequent GPU memory utilization of *GASAL2*. Furthermore, the VRAM is monitored using the NVIDIA System Management Interface, `nvidia-smi`, which provides the user with monitoring information about the NVIDIA GPUs in a system. From this analysis, we arrive at the following findings:

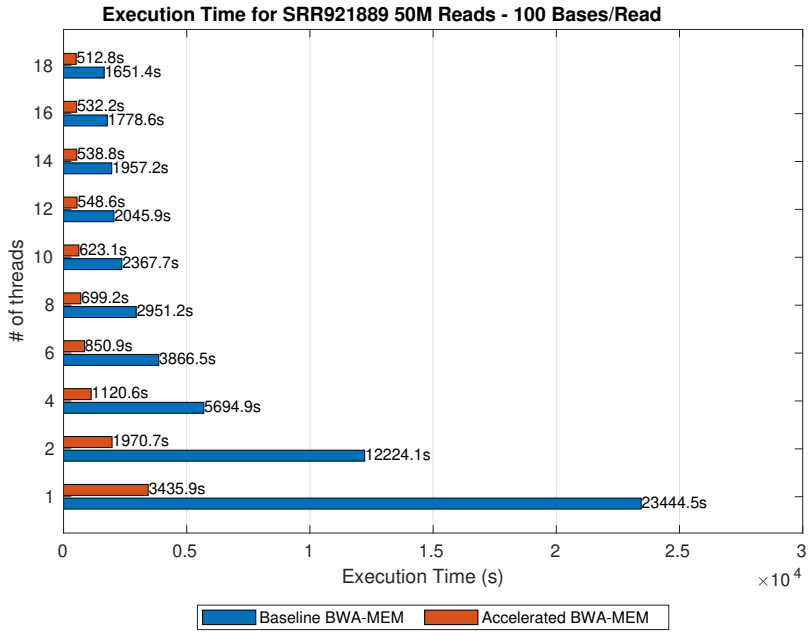
- During its execution *GPUSeed* utilized a maximum of 5.3GB of GPU memory. Out of this chunk of memory, the highest portion is taken by the FMD index, which takes around 4.7GB. The remaining chunk of memory is utilized in the seeding internal batching pipeline to store seed information and metadata.
- *GASAL2* allocates 570MB of GPU memory per CPU thread. Therefore, we can also say that 285MB is allocated for each GPU stream. In addition, *GASAL2* allocates 251MB for various alignment parameters such as substitution scores and match scores. *GASAL2* can grow this memory dynamically if required during a particular alignment preparation. However, the tuned coefficients presented in the last section of Chapter 5 ensure that the initialization memory is sufficient for the three datasets we use.

6.4.2. GPU Utilization

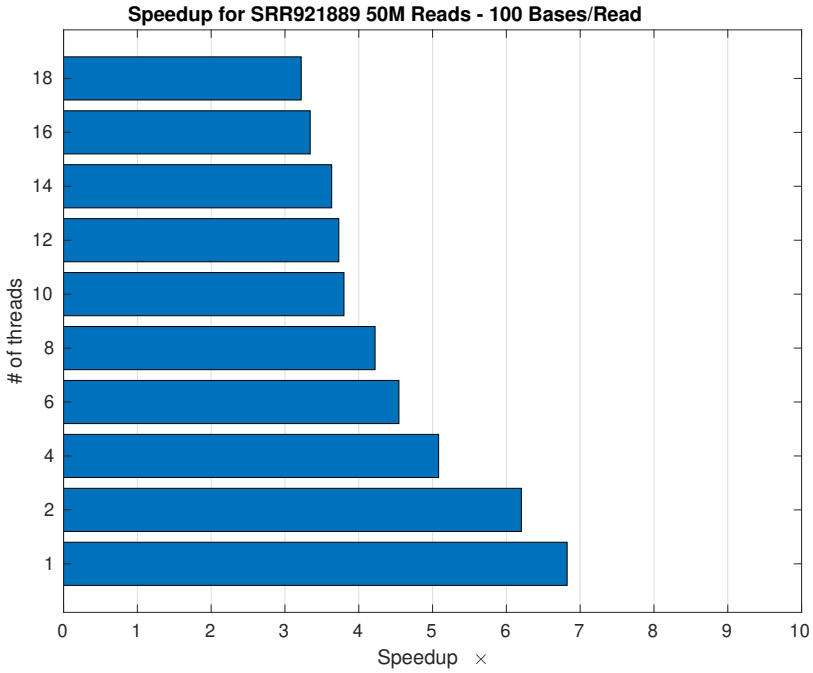
The final utilization study looks at our device's reported utilization during the execution of the three datasets in Table 6.3. It is vital to have an overview and understanding of how our GPU is used during the execution of our program. In order to measure the GPU utilization, which the NVIDIA driver reports, we use the `nvidia-smi` command and log to a CSV file the utilization percentage. Furthermore, we use a sampling rate of 1 sample every 100ms and execute our program twice for each dataset. The first execution is launched with one thread, while the second will use the number of threads that yields the best execution time for that particular dataset. The GPU utilization percentages are then plotted and presented in Figure 6.10 for all three datasets. Based on the utilization reported in Figure 6.10 we can make the following remarks:

- The utilization of the GPU during the *GPUSeed* portion can be seen as the initial wider blob between 0 and 100 seconds for the SRR921889, between 10 and 20 seconds for SRR949537 and between 10 and 30 seconds for the SRR835433.
- For the three datasets, we find that as the query length increases from 100 to 250, the GPU utilization during the *GPUSeed* phase increases from around 60% to around 75%. As expected, GPU utilization during *GPUSeed* does not depend on the number of CPU threads.
- The utilization of the GPU during the alignment phase can be seen as the repetitive smaller blobs after the GPU seeding has been completed. These repetitions indicate the different batches of queries being processed on the GPU. It can be seen in Figure 6.10 that a higher number of threads makes these alignment blobs wider since we have a larger chunk of reads to process.
- For the alignment phase, we find similar utilization trends, more specifically, that utilization is lower for shorter reads. In addition, we notice that using more threads results in a higher utilization percentage. For the SRR921889 dataset, we see around ten times higher percentage when using 18 threads instead of 1, while for the SRR949537, we find two times more utilization going from 1 to 12 threads. The SRR835433 shows the best device utilization out of all our datasets, with the 14 thread runs saturating the GPU.
- An interesting remark can be made by comparing the single thread utilization for the SRR835433 dataset from Figure 6.10 to the utilization of the same dataset presented during Chapter 4 in Figure 4.10. From these two graphs, we can see that for the single thread execution, the utilization has more than doubled from 20% close to 50%. This is caused mainly by the fact that the integrated library's alignment kernels have the seeds already calculated and are not bottlenecked by the CPU seeding phase. As a result, we can launch alignment kernels more frequently, utilizing our device more efficiently.

From our study into the utilization of the GPU over the execution cycle of our program, we have seen that our GPU stays active the vast majority of the time, leading us to conclude that there are no significant CPU bottlenecks or overhead issues that would keep our GPU idle.

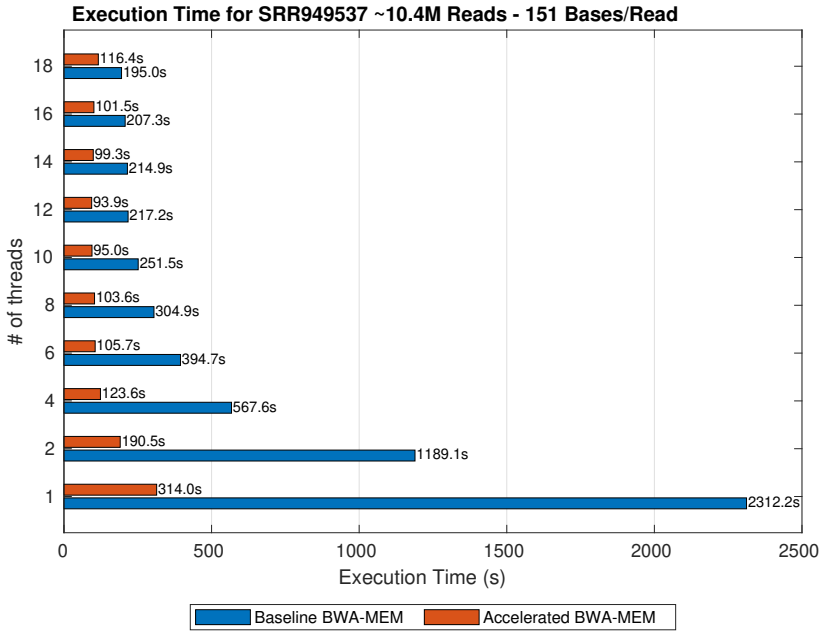


(a) Execution time Baseline vs. Accelerated *BWA-MEM* for SRR921889 dataset.

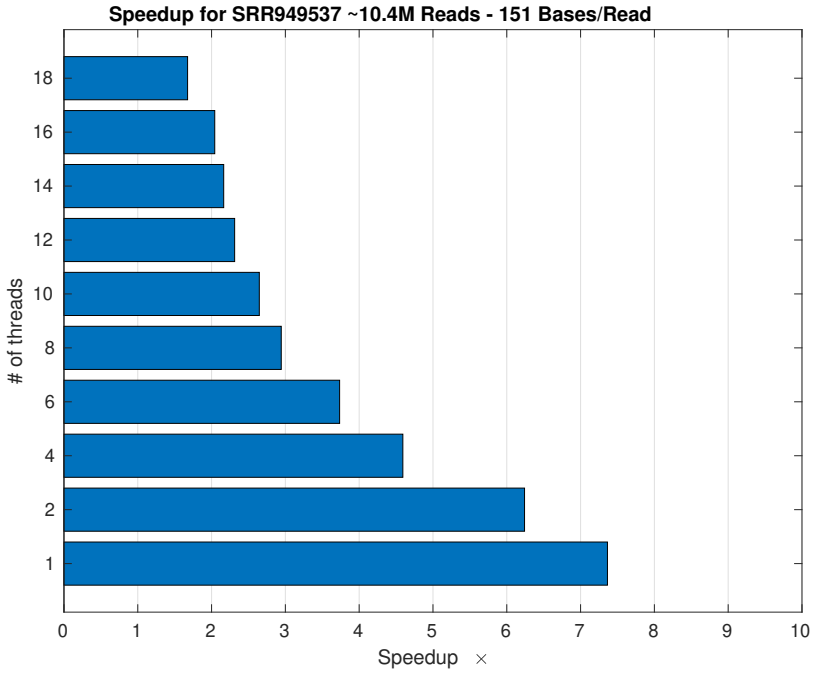


(b) Speedup over Baseline *BWA-MEM* for SRR921889 dataset.

Figure 6.4: Total execution time and speedup for SRR921889 dataset.

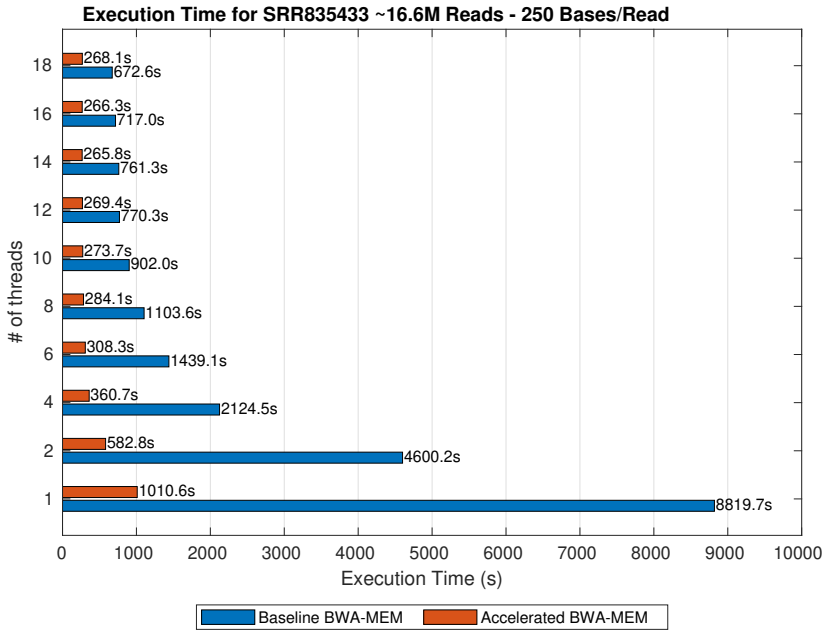


(a) Execution time Baseline vs. Accelerated *BWA-MEM* for SRR949537 dataset.

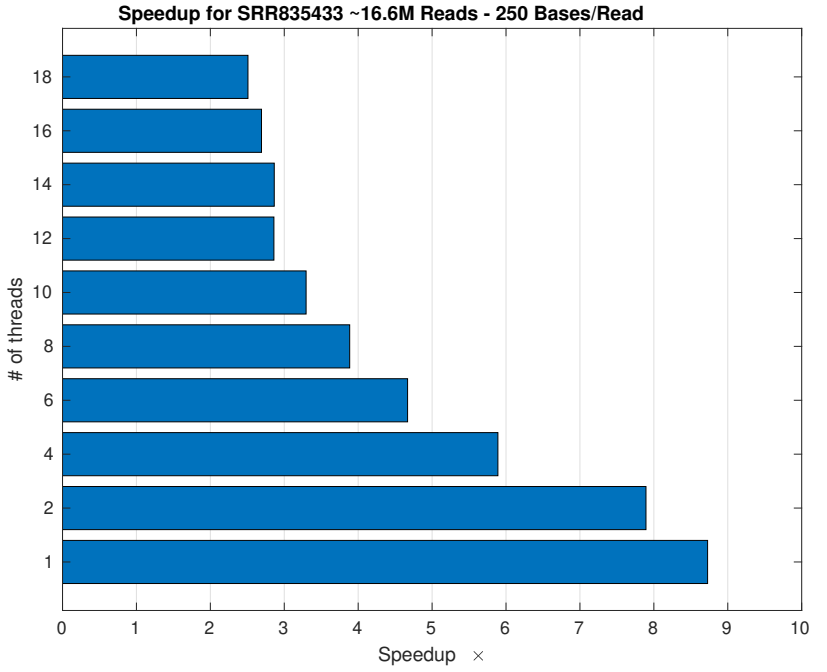


(b) Speedup over Baseline *BWA-MEM* for SRR949537 dataset.

Figure 6.5: Total execution time and speedup for SRR949537 dataset.



(a) Execution time Baseline vs. Accelerated *BWA-MEM* for SRR835433 dataset.



(b) Speedup over Baseline *BWA-MEM* for SRR835433 dataset.

Figure 6.6: Total execution time and speedup for SRR835433 dataset.

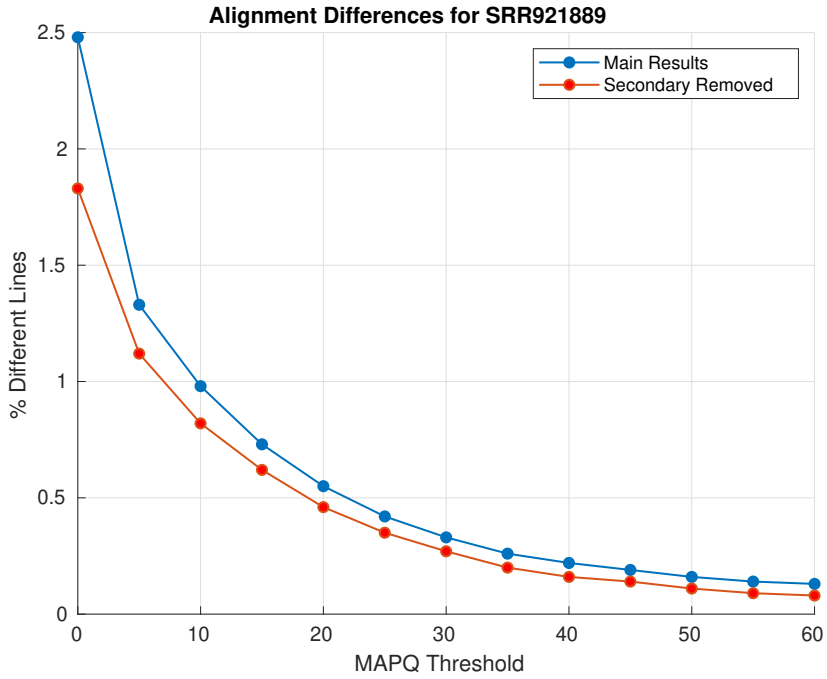


Figure 6.7: Alignment % of the different lines from the baseline for the SRR921889 dataset.

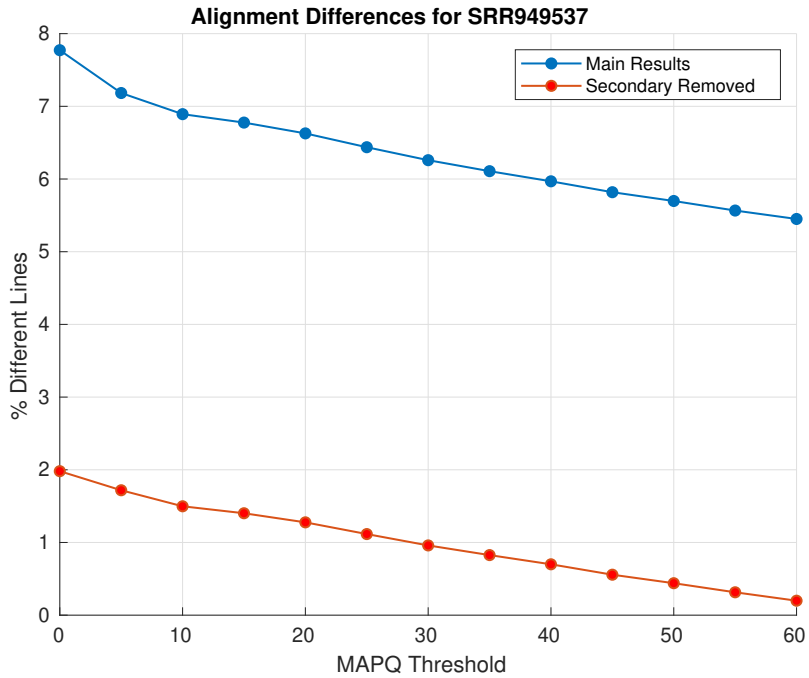


Figure 6.8: Alignment % of the different lines from the baseline for the SRR949537 dataset.

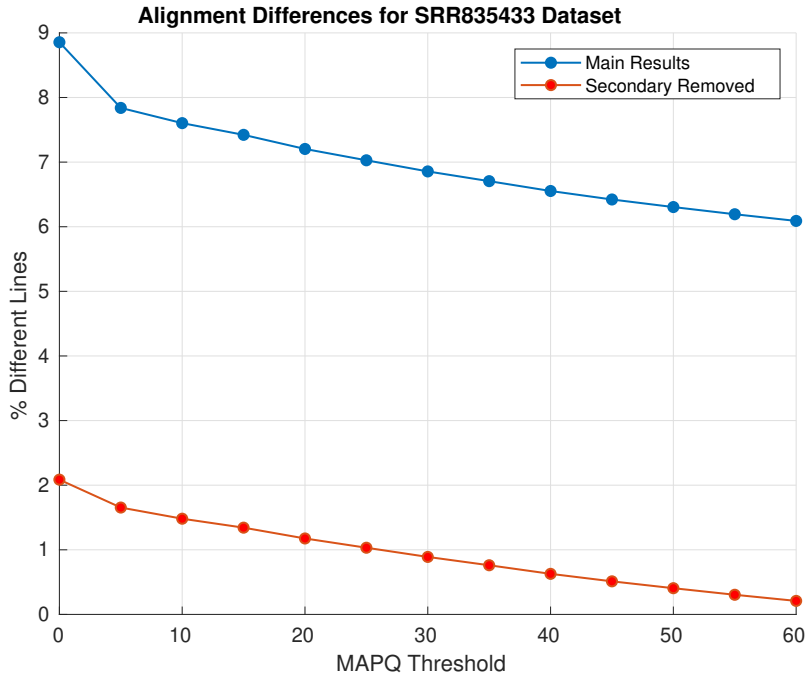


Figure 6.9: Alignment % of the different lines from the baseline for the SRR835433 dataset.

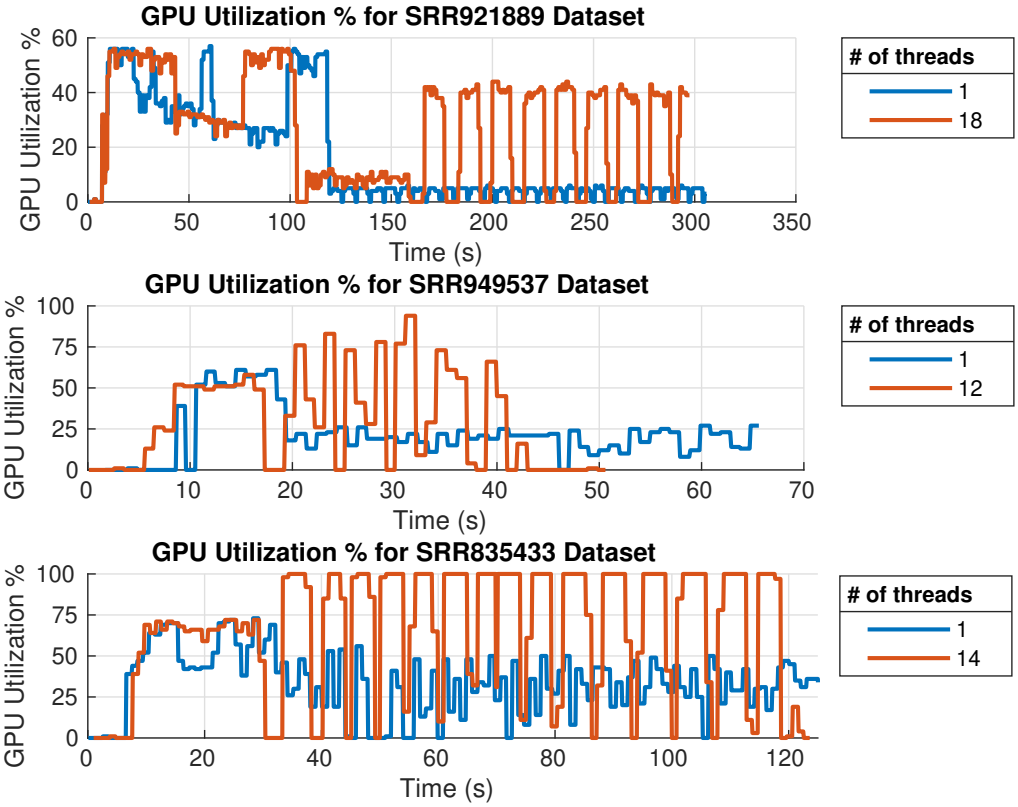


Figure 6.10: Integrated library GPU utilization % for all three datasets of Table 6.3.

7

Conclusions

In this chapter, we present a final overview of the thesis and continue by offering the main contributions of our work. Ultimately, in the last section of this chapter, we give some recommendations regarding future work which can be conducted using the outcome of this thesis.

7.1. Overview

7.1.1. Chapter 1

The journey of this thesis began by looking at what the modern-day study of DNA and genomics is and how it has evolved in recent times. The recent developments in bioinformatics have created an increased demand for better software and hardware implementations that can keep up with the high rate of growth of the field. Then, in Chapter 1, we briefly touched upon the importance of a piece in the puzzle of DNA analysis, namely, DNA alignment. Furthermore, we introduced the main focus of our work, the Burrows-Wheeler alignment tool (*BWA*), and how it is vastly utilized in the industry due to its good speed, accuracy and availability [3, 4]. The final sections of this introductory chapter presented our work's primary objectives and the general overview of what is to come in this thesis.

7.1.2. Chapter 2

Chapter 2 began by introducing in more detail the fundamental biological concepts behind DNA and then looked at a brief history of DNA sequencing. Recent DNA sequencing developments have resulted in vast market adoption of these techniques, making DNA analysis accessible to regular people. As DNA analysis applications increase and become more widespread, it becomes all the more important to keep improving current techniques and develop innovative solutions to enhance the quality of life of more people worldwide. In the context of this thesis, we focused on a small part of this challenge by addressing the DNA alignment problem and how it is solved nowadays. We presented some quick examples of how the alignment of DNA is performed and the reasons behind focusing on the *BWA* aligner.

Furthermore, we addressed the hardware on which genomics applications such as *BWA* are currently run, which are in the vast majority CPUs. Additionally, we saw the correlation between Amdahl's law and the rise of modern-day genomics, concluding that we should look at different computing architectures with higher efficiency in parallelizable tasks so that we are not limited by the slow CPU performance scaling of the last decade. The main discussion then shifted to consumer GPUs, specifically NVIDIA *CUDA* enabled devices. In the last section of Chapter 2, we presented the general architecture of such devices and typical execution models of programs using both a CPU and a GPU. Finally, we mentioned how the alignment of many independent NGS queries creates the perfect candidate for parallelization on GPUs.

7.1.3. Chapter 3

In Chapter 3, we presented the related work that was used in the context of this thesis. More specifically, this chapter was divided into three parts, each showing the theoretical background behind the three software components utilized in our work. The first part discussed the *BWA* aligner, and we described

how the aligner uses the seed and extend paradigm to align two DNA sequences. During the seed generation section, we presented how the *BWA* aligner uses the FMD index to find matching strings between the query and the reference in time relative to the query's size rather than the reference's. In addition, we presented the memory considerations for the data structures holding the FMD index and the exact algorithms used by *BWA-MEM* to find these matching strings and locate their position in the reference. During the alignment discussion, we presented the dynamic programming algorithm utilized to align two DNA sequences using the affine gap penalty model. Consequently, we expanded on that knowledge by presenting how *BWA-MEM* performs the extension phase using a few unique characteristics such as *Z-dropoff*, left and right alignments and tracking of the end-to-end alignment score, which is kept as the final alignment score under certain conditions. The first part of this chapter concluded by presenting the final output of *BWA-MEM*, which are the aligned queries in the *SAM* format.

The second part of Chapter 3 presented the *GPUSeed* library, which was previously developed to accelerate the seed generation phase. In this section, we introduced the unique stages utilized to perform the seeding phase on the GPU while highlighting how each stage has a specific goal that will enable the complete library to emulate the functionality of the CPU-based seeding as closely as possible.

This third chapter was finalized by presenting the *GASAL2* alignment library and discussing the two main kernels utilized in the context of the *BWA-MEM* integration, namely the packing and the alignment kernels. Similarly to *GPUSeed*, the *GASAL2* library has to make some assumptions in order to be able to utilize the GPU for the extension part, leading to some result differences compared to the baseline *BWA-MEM*.

7.1.4. Chapter 4

Chapter 4 focused on the analysis of the three software libraries, which we first introduced at a theoretical level during Chapter 3. The main goal of this chapter was to analyze and profile the three applications so that we have a clear overview of their capabilities and execution profiles moving forward into the integration phase. We used two real and four simulated datasets created from the GRCh37 complete human genome build to analyze all three libraries. These datasets enabled us to profile the applications using both real data and simulated reads, where we vary only the number of bases per query to see the effect of query length on our profiles. Similarly to the previous chapter, we began the analysis by looking at the latest baseline *BWA-MEM* version. From our experiments, we saw how a larger number of bases per read results in a higher percentage of time spent in the extension phase while the percentage of time spent finding *SMEMs* and locating them in the reference reduces. For this application, we determined that finding *SMEMs*, locating them in the text and then performing extension can amount to even 90% of the execution time. The last study of this section aimed at determining how the application's execution time scales with an increasing number of processing threads. This study showed an excellent scaling in performance up to 12 threads; after 12 threads, performance scaled much slower.

The second profiling study focused on the *GPUSeed* library. Since the parts of this application differ from the *BWA-MEM* seeding algorithm, we had to break down our program differently. The main portions of code followed the stages introduced in Chapter 3 when presenting the *GPUSeed* program, while for the profiling, we had to use some *CUDA* tools since we are mostly dealing with kernels executing on a GPU. The main outcomes of this profiling showed us that *GPUSeed* finds *SMEMs* around 24 to 33 times faster than *BWA-MEM*. At the same time, the *SMEMs* are not the same, with some redundant *MEMs* not being filtered properly and also, the achieved speedup does not take into account any seed storing data structure.

The final profiling study was conducted using the *BWA-MEM/GASAL2* program that has the main structure of the baseline *BWA-MEM* while performing alignment using GPU kernels provided by the *GASAL2* library. This section presents how a different batching strategy is implemented internally to utilize the GPU more efficiently for the alignment phase. The profiling of this application revealed some interesting insights regarding the behavior of this application when executed with our various datasets. For example, we concluded that a larger number of alignments does not necessarily result in a more significant amount of time spent in the alignment phase and that the length and quality of the query play an important role in the complexity of the alignment stage. Furthermore, we showed how this implementation achieves close to the theoretical maximum speedup over the baseline *BWA-MEM* by having an overlap of CPU and GPU execution during the seeding, chaining and alignment pipeline. The

performance scaling of this implementation using an increasing number of threads was also studied and compared to the baseline, concluding that it exhibits the same trend in decreasing execution time as the number of threads increases. Finally, the utilization of the GPU by the *GASAL2* library was discussed, and we saw how the GPU utilization is increased via the use of multiple GPU streams.

7.1.5. Chapter 5

In Chapter 5 we started our integration phase by taking a baseline *BWA-MEM* implementation and replacing its seeding part with the *GPUSeed* library. This exercise enabled us to quickly create an interface between the two libraries and have a naive program that finds seeds using a GPU. Using the knowledge regarding the modifications required to the baseline *BWA-MEM* acquired during the naive first integration in combination with the knowledge about the code and structure of *BWA-MEM* with *GASAL2*, we integrated the first complete library running seeding and extension on the GPU.

The second part of this chapter focused on optimizing this first integration to execute as fast as possible and remove potential bottlenecks that harm GPU performance. We identified and visualized the redundant operations using NVIDIA's profiling software, Nsight Systems. Due to the initial design of the *GPUSeed* library, which was meant to run as a standalone seeding library, we had a few operations such as reading the FM index from the file, the pre-calculate seed intervals kernel and the FM index memory transfers from host to the device that were occurring in each call to the GPU seeding wrapper function. As a result, removing all these redundant operations after performing them once during initialization enabled us to cut by almost half the total execution time for a short 1 million read query file.

The next round of optimizations focused on improving some of the memory operations performed by the *GPUSeed* library. First, we improved the query loading loop by using a variable that tells us how many bytes we need to jump in our query file to start loading the first query of the current batch the program should be processing. Secondly, the memory allocation for the FM index on the host side was changed to pinned memory to have a higher memory bandwidth during the transfer to the device. Finally, the last memory optimization altered the data structure where the seeds are stored to eliminate the need for host side computation and memory transfers for filling the previously used structures.

In the next set of optimizations, our focus shifted towards emulating the chaining function of the baseline program and improving the quality of the *SMEMs*. As a result, we altered the chaining algorithm of *BWA-MEM* in combination with the addition of a seed counting mechanism in one of the kernels of *GPUSeed* so that we replicate the functionality of the baseline *BWA-MEM* chaining while using the new seeds computed on the GPU. In addition, we adapted the *GPUSeed* pipeline so that it includes the unpublished changes of the previously developed *GPUSeed FMD* implementation and then we expanded even further by also adding the changes from the branch *GPUSeed FMD Fast Locate*. All these changes enabled us to match the seeds of the baseline *BWA-MEM* and improve performance since we have fewer seeds to process during chaining.

The move from the FM index to the FMD index drastically reduced the number of maximum threads we can use since the index is now larger and is kept in the GPU memory for the complete duration of the program's execution. As a result, we decided to alter the program's architecture by detaching the *GPUSeed* from the batching pipeline and moving it to the initial phase of our program. By relying on the RAM to keep our seeds after executing GPU seeding on all the queries, we freed valuable GPU memory enabling us to initialize again up to 24 threads (i.e., 48 GPU streams). The last part of this section discusses the coefficients used by *GASAL2* to pre-allocate memory for the host and device data structures. While testing the latest implementation, it was noticed that for some datasets, the allocated memory would have to dynamically be grown during the execution of the *GASAL2* pipeline. As a result, it was decided to alter the coefficients used for the initial memory estimation so that more memory is allocated from the beginning. This larger memory initialization enabled us to run all our datasets without seeing any memory re-allocations during execution, while the cost was that our program was limited to running with a maximum of 18 threads instead of 24. Furthermore, we altered the batching sizes of the *GASAL2* library, improving GPU occupancy from 10% to 30%.

7.1.6. Chapter 6

In Chapter 6, we studied the performance of our final integrated library against the latest version of a baseline *BWA-MEM* implementation using three real datasets of DNA queries, each with a different number of bases per query. The performance study presented two different measurements for each

dataset. The first measurement showed a breakdown of the execution time for each critical stage we defined in our application compared to a baseline execution where we measured the same stages. For this scenario, we ran both the accelerated library and non-accelerated base *BWA-MEM* using one processing thread. From this study, we determined that for all three datasets, the most dominant stage becomes the chaining phase, especially the function used to prepare and fill the *GASAL2* structures for the alignment kernel.

Similarly to our observations from Chapter 4, we found that even for the accelerated library, larger query sizes increase the percentage of time taken by extension. For the first two datasets with query lengths of 100 and 151 bases, we found that for single thread execution, our program achieved a speedup of 6.82× and 7.36× while the theoretical maximums were 6.94× and 7.93× for each respective dataset. The third dataset with queries of 250 bases achieved a speedup of 8.72× with the theoretical maximum being 9.8× bases on the baseline *BWA-MEM* measurements.

The second study looked at a realistic scenario where both accelerated and baseline libraries are executed using an increasing number of threads. Furthermore, we noticed that for the datasets with 151 and 250 bases per query, there is a thread threshold point where the benefits of using multiple threads for the chaining operations do not outweigh the negative impact of having too many stalled alignment kernels waiting for available GPU resources. As a result, we found that for the datasets with queries of 151 and 250 bases, execution with 12 and 14 threads yields the best performance, while for the dataset with 100 bases per read, 18 threads provide the lowest total execution time. The best execution time of each dataset was compared to the best execution of the same dataset using the baseline *BWA-MEM* application, and we found that for the 100, 151 and 250 bases query datasets, the speedup is 2.8×, 2× and 2.34× respectively.

Additionally, we found that the results of the alignments using our integrated library had 2.48%, 7.77% and 8.85% of lines different compared to the results coming from *BWA-MEM* for each of our datasets. When we removed the secondary scores, this number was reduced to around 2% for all datasets. Further filtering alignments with low mapping quality below 20 brought down the different line percentages to 0.46%, 1.27% and 1.17% for our three 100, 151 and 250 bases per query datasets.

The final analysis conducted in Chapter 6 looked at the resources utilized by our application. In the first section of this last investigation, we looked at how the RAM currently limits the number of queries we can process since we keep all the *SMEMs* in memory for the complete duration of our execution. We concluded that for our current platform which contains 128GB of RAM, we could compute up to 300 million queries using single thread execution or 250 queries using our application's maximum of 18 threads. In addition, we discussed the GPU global memory utilization during the seeding phase and the GPU memory utilized by the *GASAL2* library. Finally, we concluded the measurements and performance chapter by presenting the GPU utilization as the NVIDIA driver reports it for all three datasets using one thread and the number of threads corresponding to their best execution time. The main conclusions drawn from this analysis were that the GPU utilization is lower for both the seeding and extension phases in the case of queries with smaller lengths. For the *GPUSeed* library, we found between 60% and 75% utilization reported over the execution phase of this part, while the utilization during extension depended on the number of GPU streams used (i.e., number of threads). For the 100 bases per query dataset, we found utilization during extension increased eight times from around 5% to 40% while using 18 threads instead of 1. For the dataset with 151 bases per query, we saw the same utilization go from 25% to 75%, with peaks reaching close to 95% over the execution time span. The 250 bases per read dataset had the best utilization overall, with the 14 thread execution having consistent 100% utilization during extension. A final interesting observation made for this last dataset is that compared to the utilization results presented in Chapter 4 during the *GASAL2* analysis, our integrated library has higher as well as more consistent GPU utilization for both the single thread and 14 thread execution runs.

7.2. Contributions

Our work in this thesis made the following contributions.

- The *BWA-MEM* algorithm was analyzed in detail, with the seed and extension phases being explained in detail at the theoretical level and presenting the execution profile of the implementation.
- The public *GPUSeed* library was analyzed, and we presented its shortcomings with respect to

the quality of the results of *BWA-MEM*.

- The *GASAL2* library was presented at a theoretical level and also profiled to determine its execution profile and differences in alignment results from the baseline *BWA-MEM*.
- The complete integration of *GPUSeed* and *GASAL2* into the latest version of *BWA-MEM*.
- Optimization of the complete integration in incremental steps by changing individual kernels, functions or algorithm stages to extract better performance and alignment results.
- To our knowledge, at the time of writing this report, one of the only complete libraries that follow the paradigm of the *BWA-MEM* algorithm while running both seeding and extension on a GPU.
- An accelerated alignment library which can run complete human genome alignments on *CUDA* enabled GPUs with as low as 8GB of VRAM.
- Intermediate pipeline architecture allows the execution of our accelerated library on consumer hardware platforms where RAM is more limited.
- Final pipeline architecture can utilize more threads and exhibit better performance, targeting higher-end professional hardware platforms with more RAM and better compute resources.

The final integrated library implemented in this thesis can be found on GitHub [1].

7.3. Recommendations for Future Work

To conclude, we present the main recommendations for future work. We order the recommendations by the priority in which we consider these should be explored.

1. **Improve the GASAL2 extension kernel**. Improving the performance and quality of the alignments is a critical task for the future of our integrated library. The performance of the extension kernel should be improved since the extension kernel time is substantial. In the context of our pipeline, this kernel time is hidden by CPU-GPU execution overlap and multi-stream GPU execution resulting in the much lower visible extension time we discussed in Chapter 6. At the same time, the occupancy achieved by the extension kernel is quite low, ranging between 10% and 30%, and as we saw in Chapter 5 increasing the chunk of batches does improve occupancy, but execution time remains largely unchanged. As a result, most improvements using the current extension kernel architecture should focus on improving memory transfers or working on different sizes of tiles in order to see whether we can gain any significant speedup. At the same time, the heuristics of the *GASAL2* acceleration do not allow us to perfectly match the alignment results of *BWA-MEM*, meaning that we should either change the methodology of how we build our alignment batches or explore a different acceleration approach altogether. A possible solution would be to implement a systolic array-based GPU extension kernel such as the one presented in [4].
2. **Emulate BWA-MEM re-seeding in GPUSeed**. Our current implementation relies only on the *SMEMs*, and all of our work evolved around the comparison to *BWA-MEM* with its re-seeding functionality disabled. In order to completely emulate the *BWA-MEM* seeding functionality, we would have to adapt *GPUSeed* to perform the two re-seeding rounds of the baseline program correctly. This problem could be solved by adjusting our pipeline so that the kernels that find the seed intervals in the forward and backward directions are called again using the re-seeding logic of *BWA-MEM* for *SMEMs* which are longer than $1.5 \times min\_seed\_len$. The second option would be to fix the current *MEM* filtering solution to capture the intrinsics of the *BWA-MEM* re-seeding.
3. **Improve GPUSeed pipeline**. Several improvements can be applied to the GPU seeding pipeline in its current state. First, we could hide the seed memory transfers in each internal batch execution by overlapping the GPU memory transfers with the execution of the next batch of queries. Secondly, we could limit the number of seed transfers from device to host from every internal batch iteration to every x iterations. As we saw in Chapter 6 during the execution of *GPUSeed*, we only used around 5.4GB of GPU memory so we could keep the seeds corresponding to more batches of queries in the GPU memory and perform the transfer back to the host only once every x batches.

4. **Improve seeds memory accesses**. The current data structure for storing the seeds uses single arrays to store all the elements for billions of seeds. As a result, during chaining, the memory accesses to these large arrays are quite inefficient, and since we have different CPU threads accessing the same array, a more efficient memory access model could be implemented.

5. **Improve chaining functions**. As we saw in Chapter 6, the chaining functions, specifically the chain to align function, become the most dominant phase in our accelerated program. Although this part of the code is already parallelized to some extent as it is executed on different threads, and we also overlap it with the GPU extension execution, there is still a case to be made that we could integrate the seed chaining function into *GPUSeed* and improve the chain to align function. The addition of seed chaining in *GPUSeed* would be most beneficial if we would like to execute our library using one CPU thread. The addition of seed chaining into *GPUSeed* is considered a lower priority improvement.

Bibliography

[1] Stefanos Florescu. *BWA-MEM with GPUSeed and GASAL2 Repository*. https://github.com/sflorescu/BWA-MEM_GPUSeed.git. 2022.

[2] H. Li and R. Durbin. “Fast and accurate short read alignment with Burrows-Wheeler transform”. In: *Bioinformatics* 25.14 (May 2009), pp. 1754–1760. DOI: 10.1093/bioinformatics/btp324. URL: https://doi.org/10.1093/bioinformatics/btp324.

[3] Mohammed Alser et al. “Technology dictates algorithms: recent developments in read alignment”. In: *Genome Biology* 22.1 (Aug. 2021). DOI: 10.1186/s13059-021-02443-7. URL: https://doi.org/10.1186%5C%2Fs13059-021-02443-7.

[4] Ernst Joachim Houtgast et al. “GPU-Accelerated BWA-MEM Genomic Mapping Algorithm Using Adaptive Load Balancing”. In: *Architecture of Computing Systems – ARCS 2016*. Springer International Publishing, 2016, pp. 130–142. DOI: 10.1007/978-3-319-30695-7_10. URL: https://doi.org/10.1007/978-3-319-30695-7_10.

[5] St George’s University of London. *The clinical applications of genomic technologies — futurelearn.com*. https://www.futurelearn.com/info/courses/the-genomics-era/0/steps/4911. [Accessed 05-Sep-2022].

[6] Razib Khan and David Mittelman. “Consumer genomics will change your life, whether you get tested or not”. In: *Genome Biology* 19.1 (Aug. 2018). DOI: 10.1186/s13059-018-1506-1. URL: https://doi.org/10.1186/s13059-018-1506-1.

[7] Zachary D. Stephens et al. “Big Data: Astronomical or Genomical?” In: *PLOS Biology* 13.7 (July 2015), e1002195. DOI: 10.1371/journal.pbio.1002195. URL: https://doi.org/10.1371/journal.pbio.1002195.

[8] James M. Heather and Benjamin Chain. “The sequence of sequencers: The history of sequencing DNA”. In: *Genomics* 107.1 (Jan. 2016), pp. 1–8. DOI: 10.1016/j.ygeno.2015.11.003. URL: https://doi.org/10.1016/j.ygeno.2015.11.003.

[9] Mehdi Kchouk, Jean Francois Gibrat, and Mourad Elloumi. “Generations of Sequencing Technologies: From First to Next Generation”. In: *Biology and Medicine* 09.03 (2017). DOI: 10.4172/0974-8369.1000395. URL: https://doi.org/10.4172/0974-8369.1000395.

[10] National Cancer Institute. *Bioinformatics Pipeline: DNA-Seq Analysis - GDC Docs — docs.gdc.cancer.gov*. https://docs.gdc.cancer.gov/Data/Bioinformatics_Pipelines/DNA_Seq_Variant_Calling_Pipeline/. [Accessed 15-Aug-2022].

[11] Ernst Joachim Houtgast et al. “Computational Challenges of Next Generation Sequencing Pipelines Using Heterogeneous Systems”. In: *12th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems*. 2016.

[12] Heng Li. *Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM*. 2013. DOI: 10.48550/ARXIV.1303.3997. URL: https://arxiv.org/abs/1303.3997.

[13] Amit Kawalia et al. “Leveraging the Power of High Performance Computing for Next Generation Sequencing Data Analysis: Tricks and Twists from a High Throughput Exome Workflow”. In: *PLOS ONE* 10.5 (May 2015). Ed. by Christophe Antoniewski, e0126321. DOI: 10.1371/journal.pone.0126321. URL: https://doi.org/10.1371/journal.pone.0126321.

[14] Nauman Ahmed et al. “Heterogeneous hardware/software acceleration of the BWA-MEM DNA alignment algorithm”. In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2015, pp. 240–246. DOI: 10.1109/ICCAD.2015.7372576.

[15] Ernst Joachim Houtgast et al. “Hardware acceleration of BWA-MEM genomic short read mapping for longer read lengths”. In: *Computational Biology and Chemistry* 75 (2018), pp. 54–64. ISSN: 1476-9271. DOI: https://doi.org/10.1016/j.compbiolchem.2018.03.024. URL: https://www.sciencedirect.com/science/article/pii/S1476927118301555.

[16] Shanshan Ren, Koen Bertels, and Zaid Al-Ars. “Efficient Acceleration of the Pair-HMMs Forward Algorithm for GATK HaplotypeCaller on Graphics Processing Units”. In: *Evolutionary Bioinformatics* 14 (Jan. 2018), p. 117693431876054. DOI: 10.1177/1176934318760543. URL: https://doi.org/10.1177/1176934318760543.

[17] Shanshan Ren et al. “GPU accelerated sequence alignment with traceback for GATK HaplotypeCaller”. In: *BMC Genomics* 20.S2 (Apr. 2019). DOI: 10.1186/s12864-019-5468-9. URL: https://doi.org/10.1186/s12864-019-5468-9.

[18] National Human Genome Research Institute. *Deoxyribonucleic Acid (DNA) — genome.gov*. https://www.genome.gov/genetics-glossary/Deoxyribonucleic-Acid. [Accessed 16-Aug-2022].

[19] MedlinePlus. *What is DNA?: MedlinePlus Genetics — medlineplus.gov*. https://medlineplus.gov/genetics/understanding/basics/dna/. [Accessed 01-Sep-2022].

[20] National Human Genome Research Institute. *Chromosome — genome.gov*. https://www.genome.gov/genetics-glossary/Chromosome. [Accessed 01-Sep-2022].

[21] MedlinePlus. *What is a gene?: MedlinePlus Genetics — medlineplus.gov*. https://medlineplus.gov/genetics/understanding/basics/gene/. [Accessed 01-Sep-2022].

[22] National Human Genome Research Institute. *Genome — genome.gov*. https://www.genome.gov/genetics-glossary/Genome. [Accessed 01-Sep-2022].

[23] PacBio. *Sequencing 101: The Evolution of DNA Sequencing Tools — pacb.com*. https://www.pacb.com/blog/the-evolution-of-dna-sequencing-tools/. [Accessed 05-Sep-2022].

[24] Mehdi Kchouk, Jean Francois Gibrat, and Mourad Elloumi. “Generations of Sequencing Technologies: From First to Next Generation”. In: *Biology and Medicine* 09.03 (2017). DOI: 10.4172/0974-8369.1000395. URL: https://doi.org/10.4172/0974-8369.1000395.

[25] James M. Heather and Benjamin Chain. “The sequence of sequencers: The history of sequencing DNA”. In: *Genomics* 107.1 (Jan. 2016), pp. 1–8. DOI: 10.1016/j.ygeno.2015.11.003. URL: https://doi.org/10.1016/j.ygeno.2015.11.003.

[26] “Finishing the euchromatic sequence of the human genome”. In: *Nature* 431.7011 (Oct. 2004), pp. 931–945. DOI: 10.1038/nature03001. URL: https://doi.org/10.1038/nature03001.

[27] National Human Genome Research Institute. *The Cost of Sequencing a Human Genome — genome.gov*. https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost. [Accessed 05-Sep-2022].

[28] Illumina. *NGS Workflow Steps | Illumina sequencing workflow — illumina.com*. https://www.illumina.com/science/technology/next-generation-sequencing/beginners/ngs-workflow.html. [Accessed 05-Sep-2022].

[29] National Cancer Institute. *NCI Dictionary of Genetics Terms — cancer.gov*. https://www.cancer.gov/publications/dictionaries/genetics-dictionary/def/somatic-variant. [Accessed 05-Sep-2022].

[30] MedlinePlus. *What are single nucleotide polymorphisms (SNPs)?: MedlinePlus Genetics — medlineplus.gov*. https://medlineplus.gov/genetics/understanding/genomicresearch/snp/. [Accessed 05-Sep-2022].

[31] J. M. Mullaney et al. “Small insertions and deletions (INDELs) in human genomes”. In: *Human Molecular Genetics* 19.R2 (Sept. 2010), R131–R136. DOI: 10.1093/hmg/ddq400. URL: https://doi.org/10.1093/hmg/ddq400.

[32] Medhat Mahmoud et al. “Structural variant calling: the long and the short of it”. In: *Genome Biology* 20.1 (Nov. 2019). DOI: 10.1186/s13059-019-1828-7. URL: https://doi.org/10.1186/s13059-019-1828-7.

[33] Tanveer Ahmad, Zaid Al Ars, and H Peter Hofstee. “VC@Scale: Scalable and high-performance variant calling on cluster environments”. In: *GigaScience* 10.9 (Sept. 2021). DOI: 10.1093/gigascience/giab057. URL: https://doi.org/10.1093/gigascience/giab057.

[34] Hao Ye et al. “Alignment of Short Reads: A Crucial Step for Application of Next-Generation Sequencing Data in Precision Medicine”. In: *Pharmaceutics* 7.4 (Nov. 2015), pp. 523–541. DOI: 10.3390/pharmaceutics7040523. URL: https://doi.org/10.3390/pharmaceutics7040523.

[35] Nauman Ahmed et al. “GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data”. In: *BMC Bioinformatics* 20.1 (Oct. 2019). DOI: 10.1186/s12859-019-3086-9. URL: https://doi.org/10.1186/s12859-019-3086-9.

[36] Andrey D. Prjibelski, Anton I. Korobeynikov, and Alla L. Lapidus. “Sequence Analysis”. In: *Encyclopedia of Bioinformatics and Computational Biology*. Ed. by Shoba Ranganathan et al. Oxford: Academic Press, 2019, pp. 292–322. ISBN: 978-0-12-811432-2. DOI: https://doi.org/10.1016/B978-0-12-809633-8.20106-4. URL: https://www.sciencedirect.com/science/article/pii/B9780128096338201064.

[37] Stephen F. Altschul et al. “Chapter 20.1/Sequence Alignment”. In: *Handbook of Discrete and Combinatorial Mathematics*. 2nd. CRC Press/Taylor & Francis.

[38] Saul B. Needleman and Christian D. Wunsch. “A general method applicable to the search for similarities in the amino acid sequence of two proteins”. In: *Journal of Molecular Biology* 48.3 (1970), pp. 443–453. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(70)90057-4. URL: https://www.sciencedirect.com/science/article/pii/0022283670900574.

[39] T.F. Smith and M.S. Waterman. “Identification of common molecular subsequences”. In: *Journal of Molecular Biology* 147.1 (1981), pp. 195–197. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(81)90087-5. URL: https://www.sciencedirect.com/science/article/pii/0022283681900875.

[40] Ryan Musich, Lance Cadle-Davidson, and Michael V. Osier. “Comparison of Short-Read Sequence Aligners Indicates Strengths and Weaknesses for Biologists to Consider”. In: *Frontiers in Plant Science* 12 (Apr. 2021). DOI: 10.3389/fpls.2021.657240. URL: https://doi.org/10.3389/fpls.2021.657240.

[41] M. Burrows and D. J. Wheeler. *A block-sorting lossless data compression algorithm*. Tech. rep. 1994.

[42] P. Ferragina and G. Manzini. “Opportunistic data structures with applications”. In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE Comput. Soc. DOI: 10.1109/sfcs.2000.892127. URL: https://doi.org/10.1109/sfcs.2000.892127.

[43] Mark D. Hill and Michael R. Marty. “Amdahl’s Law in the Multicore Era”. In: *Computer* 41.7 (July 2008), pp. 33–38. DOI: 10.1109/mc.2008.209. URL: https://doi.org/10.1109/mc.2008.209.

[44] PassMark. *PassMark CPU Benchmarks - AMD vs Intel Market Share — cpubenchmark.net*. https://www.cpubenchmark.net/market_share.html. [Accessed 06-Sep-2022].

[45] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. “Timing Predictability in High-Performance Computing With Probabilistic Real-Time”. In: *IEEE Access* 8 (2020), pp. 208566–208582. DOI: 10.1109/access.2020.3038559. URL: https://doi.org/10.1109/access.2020.3038559.

[46] Yifan Sun et al. “Summarizing CPU and GPU Design Trends with Product Data”. In: *CoRR* abs/1911.11313 (2019). arXiv: 1911.11313. URL: http://arxiv.org/abs/1911.11313.

[47] Kimberly Powell. *World Record-Setting DNA Sequencing Technique Uses Clara Parabricks | NVIDIA Blog — blogs.nvidia.com*. https://blogs.nvidia.com/blog/2022/01/12/world-record-genome-sequencing-parabricks/. [Accessed 07-Sep-2022]. 2022.

[48] NVIDIA. *How GPU Acceleration Works*. URL: http://www.nvidia.com/docs/IO/143716/%20how-gpu-acceleration-works.png.

[49] NVIDIA. *Cuda C++ Programming Guide - Nvidia Developer*. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[50] Pradeep Gupta. *CUDA Refresher: The CUDA Programming Model | NVIDIA Technical Blog — developer.nvidia.com*. https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/. [Accessed 06-Sep-2022]. 2020.

[51] Mark Harris. *GPU Pro Tip: CUDA 7 Streams Simplify Concurrency | NVIDIA Technical Blog — developer.nvidia.com*. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/. [Accessed 07-Sep-2022]. 2015.

[52] Nauman Ahmed, Koen Bertels, and Zaid Al-Ars. "Efficient GPU Acceleration for Computing Maximal Exact Matches in Long DNA Reads". In: *Proceedings of the 2020 10th International Conference on Bioscience, Biochemistry and Bioinformatics*. ACM, Jan. 2020. DOI: 10.1145/3386052.3386066. URL: https://doi.org/10.1145/3386052.3386066.

[53] Heng Li. "Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly". In: *Bioinformatics* 28.14 (May 2012), pp. 1838–1844. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bts280. eprint: https://academic.oup.com/bioinformatics/article-pdf/28/14/1838/582806/bts280.pdf. URL: https://doi.org/10.1093/bioinformatics/bts280.

[54] Heng Li. *Burrow-wheeler aligner Repository*. https://github.com/lh3/bwa.git. [Accessed 26-Sep-2022]. 2022.

[55] Nauman Ahmed. "High Performance Seed-and-Extend Algorithms for Genomics". English. PhD thesis. Delft University of Technology, 2020. ISBN: 978-94-6384-108-5. DOI: 10.4233/uuid:7e916f03-09cc-4510-9914-03a44b339462.

[56] *GRCh37 - hg19 - Genome - Assembly - NCBI — ncbi.nlm.nih.gov*. https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.13/. [Accessed 23-Aug-2022].

[57] Heng Li. *Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM*. URL: http://lh3lh3.users.sourceforge.net/download/mem-poster.pdf.

[58] Srinivas Aluru. *Handbook of Computational Molecular Biology*. Chapman & Hall/CRC, 2006.

[59] J. Lévy. "Acceleration of Seed Extension for BWA-MEM DNA Alignment Using GPUs". MA thesis. TU Delft, 2019. URL: http://resolver.tudelft.nl/uuid:bd22471f-058a-4071-95bb-5126e263124b.

[60] T.S.F.S.W. Group. *Sequence Alignment/Map Format Specification*. [Accessed 23-Aug-2022]. Aug. 2022. URL: https://samtools.github.io/hts-specs/SAMv1.pdf.

[61] Nauman Ahmed. *GPUSeed Repository*. https://github.com/nahmedraja/GPUseed.git. [Accessed 26-Sep-2022]. 2019.

[62] J. Lévy. *BWA-MEM with GASAL2 Repository*. https://github.com/j-levy/bwa-gasal2. [Accessed 26-Sep-2022]. 2019.

[63] Intel. *Intel® Xeon® Processor E5-2620 v3 Product Specifications — ark.intel.com*. https://ark.intel.com/content/www/us/en/ark/products/83352/intel-xeon-processor-e52620-v3-15m-cache-2-40-ghz.html. [Accessed 23-Aug-2022].

[64] NVIDIA. *Graphics Reinvented: NVIDIA GeForce RTX 2080 Ti Graphics Card — nvidia.com*. https://www.nvidia.com/en-me/geforce/graphics-cards/rtx-2080-ti/. [Accessed 23-Aug-2022].

[65] Heng Li. *Wgsim Repository*. https://github.com/lh3/wgsim.git. [Accessed 26-Sep-2022]. 2011.

[66] National Library of Medicine. *SRR921889 SRA Archive: NCBI — trace.ncbi.nlm.nih.gov*. https://trace.ncbi.nlm.nih.gov/Traces/index.html?view=run_browser&acc=SRR921889&display=metadata. [Accessed 23-Aug-2022].

[67] National Library of Medicine. *SRR835433 SRA Archive: NCBI — trace.ncbi.nlm.nih.gov*. https://trace.ncbi.nlm.nih.gov/Traces/index.html?view=run_browser&acc=SRR835433&display=metadata. [Accessed 23-Aug-2022].

[68] NVIDIA. *User Guide :: Nsight Systems Documentation — docs.nvidia.com*. https://docs.nvidia.com/nsight-systems/UserGuide/index.html. [Accessed 07-Sep-2022].

[69] Mark Harris. *How to Optimize Data Transfers in CUDA C/C++ | NVIDIA Technical Blog — developer.nvidia.com*. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/. [Accessed 07-Sep-2022].

[70] National Library of Medicine. *SRR949537 SRA Archive: NCBI — trace.ncbi.nlm.nih.gov*. https://trace.ncbi.nlm.nih.gov/Traces/index.html?view=run_browser&acc=SRR949537&display=metadata. [Accessed 01-Sep-2022].