Sequence Alignment of Long Noisy Reads on Graphics Processing Units

Sean Christian Dutch

Thesis submitted for the degree of
Master in Informatics: Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2017

Sequence Alignment of Long Noisy Reads on Graphics Processing Units

Sean Christian Dutch

Sequence Alignment of Long Noisy Reads on Graphics Processing Units

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

New DNA sequencing technologies such as those produced by Pacific Biosciences (PB) and Oxford Nanopore Technologies (ONT) make it possible to sequence much longer reads than with previous sequencing technologies.

These long reads come with a high error rate compared to earlier NGS (Next-Generation Sequencing) technologies such as Illumina, and therefore, new specialised sequence alignment algorithms such as MHAP, BLASR, Minimap and DALIGNER have been created to overcome this.

Graphics Processing Units (GPUs) offer immense processing power, and in this thesis, methods for utilising these devices for sequence alignment of long noisy reads have been explored.

In this thesis I present a fast parallel sequence overlap algorithm based on Minimap using GPUs. The program was written in CUDA C/C++ for Nvidia's Kepler architecture.

Results showed a speedup over a serial CPU implementation for all configurations and data sets tested, with speedups up to 22,5x using ultra-long reads. The application was also shown to be efficient with all configurations, suggesting it may be well suited for researchers and other practitioners needing a fast way to overlap long noisy reads to reference genomes using configurations aimed at higher sensitivity.

Acknowledgements

First and foremost, I want to thank my supervisor Professor Torbjørn Rognes for his patience and guidance while working on this thesis. My thanks also goes out to my family for all of their kind support. I would also like to thank all my good friends that I have acquired during my five years at IFI, for making it such a wonderful time. Last, but not least, I want to thank my dearest Marte for her infinite love and support.

Contents

Acronyms

ONT: Oxford Nanopore Technologies

PB : Pacific Biosciences

SMRT : Single Molecule Real Time Sequencing

CPU: Central Processing Unit

GPU: Graphics Processing Unit

SIMD: Single Instruction Multiple Data

SIMT: Single Instruction Multiple Threads

GPGPU: General Purpose Graphics Processing Unit

SM: Streaming Multiprocessor

API: Application Programming Interface

ILP: Instruction Level Parallelism

IPC: Instructions Per Cycle

FLOP/s : Floating Point Operations/second

Chapter 1

Introduction

Graphics Processing Units (GPUs) are processing units specialised in rendering computer graphics. To be able to render real-time graphics for applications such as computer games and other simulations, these devices have been designed to have massively parallel architectures. These architectures consist of thousands of cores able to manipulate large quantities of data in parallel at a blazing speed.

Due to the ever increasing demand for computational speed that is required in solving complex and time-consuming calculations, researchers and other practitioners have recognised the potential of GPUs, and applied them to general purpose applications that traditionally have been solved using Central Processing Units (CPUs).

GPUs have shown great promise in many research areas, from accelerating deep neural networks used in machine learning for tasks such as image analysis and language processing [1], to bioinformatics applications [2, 3]

Single-molecule sequencers such as the those produced by Pacific Biosciences [4] and Oxford Nanopore Technologies [5] are capable of producing longer reads of up to several thousand base pairs compared to previous sequencing technologies such as Illumina. The problem with these reads however, is that they tend to have a considerably high error rate compared to traditional sequencing methods.

Previous alignment tools have been designed for aligning shorter reads with much lower error rates. As a result, new tools that are better suited for aligning long noisy reads have been created.

In this thesis I have implemented a parallel long read overlapper, based on mapping tool called Minimap, created by Heng Li[6]. Minimap is a tool inspired by many of its predecessors, lending many of its concepts from DALIGNER [7], MHAP [8] and Graphmap [9].

1.1.1 Goals and Research Questions

Programming GPUs is a complex and challenging activity that requires the developer to have good knowledge of computer architecture and programming.

Ultimately, my aim for this thesis is to investigate whether GPUs are suitable for long read alignment, and to make an assessment of what kind of limitations and problems can appear in doing so.

In this thesis, I wish to study existing algorithms for long read alignment, and to make an assessment of their performance and accuracy. I also wish to see whether any of these tools have characteristics that will enable parallelisation.

My goal for this thesis therefore is to pick the most appropriate of these tools, utilise its core algorithmic components, and implement these on a GPU platform such as CUDA (Compute Unified Device Architecture). Further, I wish to compare my own implementation to its existing CPU implementation, make an assessment of whether there is any considerable performance to be gained, and under which conditions.

1.1.2 Structure of Thesis

Chapter 2: Background

The purpose of this chapter is to give readers a good understanding over the topics covered in this thesis. This includes giving an introduction to some basic biology and bioinformatics. Later I will give an introduction to parallel and GPU programming, as well as describe the core concepts of GPU architectures in general.

Chapter 3: Implementation

First I will give my reasons for choosing Minimap over the other tools. Next, I will describe the design and implementation of the parallel GPU version of Minimap. Finally, I will also discuss some its potential shortcomings and limitations.

Chapter 4: Methods and Materials

This chapter gives an overview of hardware and software used for development and testing. It also includes data sets and methods used for data gathering.

Chapter 5: Results and Discussion

In this chapter I will present and discuss the results of my research.

Chapter 6: Conclusion and Future Work

This chapter will conclude the thesis, and I will discuss whether the goals of the thesis were met. Finally, I will propose some future work, including any potential improvements to the application, as well as suggestions on how to do so.

Chapter 2

Background

2.1 Bioinformatics

The aim of this section is to introduce the readers to some basic biology, as well as to discuss the development of sequencing technology over the past decades. I will present the two most popular long read technologies and finally I will discuss different tools designed for long noisy reads.

2.1.1 Biology

Deoxyribonucleic acid (DNA) encodes the building blocks of all living things [1]. DNA code is made up of four *nucleotides*: *adenine* (A), *thymine* (T), *cytosine* (C) and *guanine* (G). In the case of ribonucleic acid (RNA), thymine is substituted for *urasil* (U).

DNA encodes *amino acids*, of which there are 20 different types [10]. These amino acids are chained together into what we know as *proteins*. In Figure 2.1, we can see how DNA is connected together with their *Watson-Crick* complements in a *double-helix* structure, where adenine complements thymine, and cytosine complements guanine. The sugar phosphate acts as the backbone holding the *basepairs* together.

Whenever a cell requires a new protein to be made, an RNA copy, called the *messenger RNA* (mRNA) is made and sent to the *cytoplasm* of the cell. This RNA copy is stripped of *introns*, which are parts of the DNA that don't carry instructions for the protein, and we are left with the *exons*, that do. In the cytoplasm another protein known as a *ribosome* reads the RNA-strand and uses this as a "recipe" to create a protein. This process is commonly known as *the central dogma* [12].

The entire set of DNA in an organism is called the *genome*. The genome itself consists of *chromosomes*, each containing some DNA. When we talk about *genes*,

[1] apart from viruses

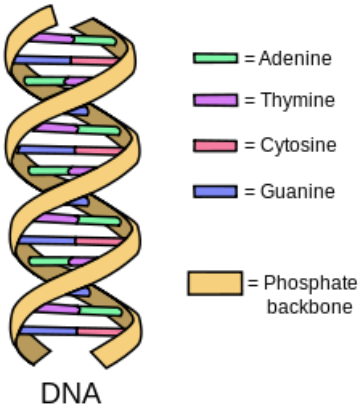


Figure 2.1: **Double-Helix DNA**. Complimenting molecules are connected together in double-helix structure. [11]

we are referring to any DNA sequence that encodes a protein or an RNA-molecule [10].

2.1.2 Sequencing Technology

DNA sequencing concerns methods that are used to sequence DNA, into what we call *reads*. Being able to do this is important for many reasons including classification and detection of pathogenic organisms, diagnosing and treating medical conditions and gathering genetic data across populations and analysing variations and changes in them [13].

Historically, the first methods for DNA sequencing were introduced by Frederick Sanger in 1977 [14], with his introduction of the "enzymatic dideoxy DNA sequencing technique based on the chain-terminating dideoxynucleotide analogues and the Allan Maxam and Walter Gilbert's chemical degradation DNA sequencing technique in which terminally labeled DNA fragments were chemically cleaved at specific bases and separated by gel electrophoresis [15]." [16].

The "first generation" DNA sequencers were based off of the "Sanger Method". This method had some shortcomings. First of all because sequencing one million base pairs cost $2400 using the *Sanger 3730xl* sequencer, in addition to having very long running times, producing only 1.9 to 84 Kb in 20 min to 3 hours [17].

Around the the mid 2000s, "next generation" sequencers (NGS), such as *454 life science*, *Solexa*, and *ABI SOLiD* emerged [10]. These methods offered so-called *High-Throughput Sequencing* methods, that unlike sequencers based on the Sanger method, were capable of not only sequencing one DNA fragment at the time, but many in parallel. Such NGS sequencers are capable of producing several gigabytes of biological data, for only a fraction of the price of first generation methods. As a result of this, sequencing the entire human genome in 2015 cost below $1500, compared to around $100 million in 2001 [18].

The "Third generation", also called "long-read" sequencers, started emerging around 2010, when *Pacific Biosciences* (PB) presented the *Single Molecule Realtime Sequencing* (SMRT) method, and some years later, *Nanopore Sequencing* methods developed by *Oxford Nanopore Technologies* (ONT) were introduced. It is these long-read technologies that are the subjects of this thesis, and I will be discussing them further below.

Illumina

Although this thesis focuses on reads produced by long read technology, it is necessary that we mention the Illumina sequencer. The reason for this is the Illumina sequencer still is one of the most popular sequencers, due to its high speed, low cost and highly accurate reads.

The first Illumina sequencer was introduced in 2006. Illumina is a short read sequencer that operates by clustering DNA along *flowcells*. *Fluorescent primers* are washed over the flowcell, attaching to a complimentary molecule. This gives each molecule an unique colour. Finally a laser is used to activate the primers, and a camera records the colours they emit [19]. Due to the short reads produced by Illumina sequencers, repetitive regions in DNA quickly become a problem as read lengths are shorter than the repeats [20].

Over the years, several different Illumina sequencers have been introduced, including different series of instruments with different price tags and performance statistics, including, but not limited to the *MiniSeq*, the *HiSeq* and the the *NovaSeq* sequencers.

Regarding Long Reads

It is useful to think of reads as pieces a puzzle. Lets also imagine that this image depicts a some mountains and a blue sky. The smaller the pieces are, the harder it is to place them quickly and correctly next to the other pieces. This also gets increasingly hard if the puzzle one is trying to put together contains a lot of similar looking pieces, ie. blue sky. If the pieces were larger however, the task would become easier as these similar looking pieces would now be more likely to contain some parts of the mountain, or maybe some clouds, and thus be more distinguishable. This is much how the difference of short and long reads work.

DNA often contains many *repeats*, much like the image in the puzzle analogy contains a lot of blue sky. A read is a string of nucleotides, and the shorter this string is, the more likely it is that this combination will occur in another read, and also the more likely it will occur several times in the genome we might be trying to align it to. By using longer reads, these repeats become increasingly easier to detect, and it becomes easier to know where to position the read in relation to the genome.

The problem with the sequences produced by Pacific Biosciences and Oxford Nanopore Technologies is that there is a high level of noise in these reads . By noise we are talking about discrepancies between the read and the actual source. There are three types of errors in sequences. *Inserts* and *deletes* (*indels*), and

substitutions. An indel occurs when molecules are inserted or deleted from the sequence, and a substitution occurs when a molecule is substituted by another one.

The errors in long read technology have been found to be randomly distributed [7]. We can use this to our advantage by sampling the same area of the genome several times, resulting in higher *coverage*. Coverage, also referred to as the *depth*, is "the average number of times a nucleotide is represented by a high-quality base in a collection of random raw sequence." [21]. With sufficient coverage, it is possible to compare the reads over the same region, and determine a more accurate representation of the sequence using consensus based methods.

Previous algorithms aren't really designed to handle long noisy reads, which is why complex and computationally intensive algorithms have been proposed [6]. Some of these are BLASR [22], DALIGNER[7], HMAP[8], GraphMap[9] and Minimap[6]. All of which we will be going to present in Section 2.2.

Single Molecule Realtime Sequencing

SMRT technology works by making use of *Phospholinked Neuclotides* that emit a specific colour for each nucleotide. *DNA polymerase* is observed as it cleaves the DNA lowered into a *Zero-Mode Waveguide* (ZMW) which is a small cylindrical chamber, where a laser is used to excite the nucleotides, causing them to emit a light which is recorded by a camera, thus determining which nucleotide it is [23, 24], this is illustrated in Figure 2.2. The ZMWs are placed in array, so that multiple reads are produced in parallel.

PB currently offers two machines, namely the *PacBio RS II* and *The Sequel System*. The sequel system being the newest, promises "more scalability, a reduced footprint and lower sequencing project costs" [25] and has a total of one million ZMWs compared to the RS II's 150 000.

The error rate of PB reads is around 16%, where approximately 1% are caused by substitutions, and the remaining 15% are indels [26]. The erroneous PB reads can be remedied by using *Circular Consensus Sequencing* (CCS) reads [27]. CCS reads are created by placing *hairpin adaptors* on both sides of a linear DNA sequence [23], allowing the DNA to be sequenced multiple times, consequently sacrificing some of the read length. One study claims to have produced CCS reads with an accuracy of 97,7% [28].

Nanopore Technology

ONT currently offers several different sequencing instruments, some are currently on the market, and others are still in development. The most researched sequencer is the *MinION*, which is a small portable sequencer that can be connected to a USB 3.0 port of a computer. The MinION is possibly most known outside the field of bioinformatics as the device used when the first DNA was successfully sequenced in space, on board the international space station [29].

ONT sequencers operate by measuring the current as molecules pass through an array of *protein nanopores* that are placed on a membrane with high electrical

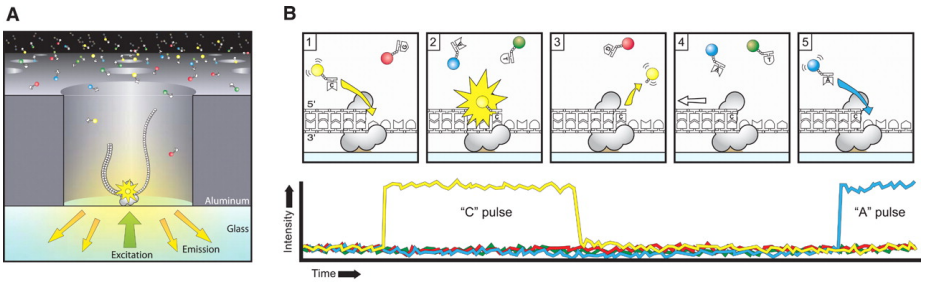


Figure 2.2: **SMRT Sequencing**. A:Zero-Mode Waveguide. B: Recording of light emissions from excited nucleotides. [24]

resistance, on a flowcell. The current passing through creates *nanopore signals* that are used to determine the correct molecule, see Figure 2.3 .*Base calling* is performed via a network connection to a cloud-based service called Metrichor, where it utilises a *Hidden Markov Model* (HMM) to predict the correct bases based on each possible state for a k-mer with $k = 6$, by default [23, 9].

Due to its portability, the MinION has promising applications other than biological research, including medical and pharmaceutical research [13]. Other ONT devices are currently in development, such as the *PromethION* sequencer, described as a "High-throughput, high-sample number bench top system" [30]. One of the notable differences being that the PromethION contains 48 flowcells, as opposed to the MinIONs one. It is "projected to produce up to 6 Tb of sequencing data each day. This equates to about 200 human genomes per day at 30x coverage" [31]. ONT are also working on the *SmidgION*, which is even smaller than the MinION and is designed for use with smart phones and other mobile devices.

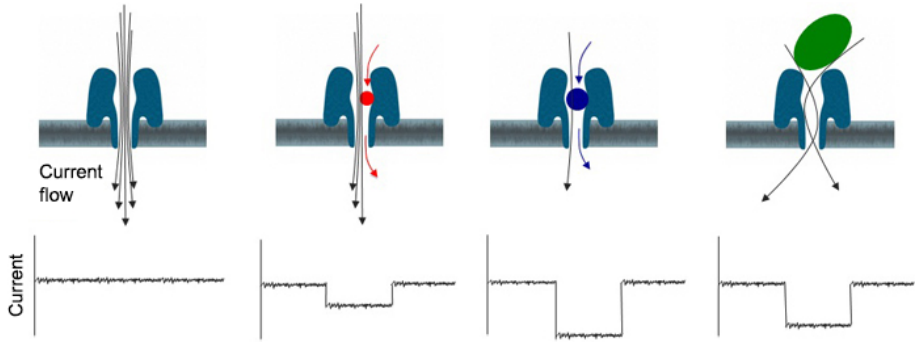


Figure 2.3: **Nanopore Sequencing**. The current changes as molecules of different sizes pass through the nanopore [5]

ONT reads are much more prone to indels than substitutions, much like PB reads [32]. Also much like CCS for PB reads, the high error rates can be reduced by generating two-directional reads, called *2D-reads*. This is done with a hairpin adaptor, allowing for forward (*template*) and reverse (*compliment*) [33] strands of the sequence. By combining these, it has been shown to produce a mean average of 85% accuracy [34]. This is a significant improvement compared

to the initial one-directional *1D reads*, having 62% accuracy [35].

Improving the sequencing chemistry is an active field of research, and it is shown that the accuracy of high quality 2D reads will improve as the chemistry does [35]. Current single read accuracy has been reported as 92% for MinION reads [36], and using consensus based base calling methods, reads have been shown to have up to 99.5% accuracy at 30x coverage [37]. Although average nanopore reads tend to be approximately 10 thousand base pairs, there are so called "ultra-long" reads are being reported using newer library preparation kits [38, 39].

2.1.3 Properties of Sequencing Technology

There are several useful properties of interest when studying sequencing technologies, most of them are outlined below.

Read length Read lengths determine the number of bases produced in one read.

Error rate The error rate represents the percentage of incorrectly classified nucleotides.

Cost The price of sequencing a certain number of bases.

Number of reads per run How many reads the technology produces in one run. Sometimes referred to as *throughput*.

Time per run How much time it is required to run the DNA sequencer.

In Table 2.1, I have compiled this information related to the Illumina HiSeq 2500, PacBio RS II and the Oxford Nanopore MinION [40, 26, 41, 25, 36, 35, 30, 42, 39]. It should be noted that long read technology is a fast moving field of research, and specifically error rates may depend heavily on use of chemistry and base calling methods. With regard to the MinION sequencer it is not an "apples-to-apples" comparison, as read lengths and number of reads depends on library preparation done beforehand, and the only limitation of the read length really depends on the molecule breaking. ONT read lengths for example are approaching one million base pairs, and have currently been reported of up to 900 Kbp [2] [39]. Figure 2.4 illustrates the current progress of sequencing technologies, in regard to read length and number of reads per run.

Despite the long read lengths, high error rates disqualify ONT and PB reads from certain applications, for example, detecting individual nucleotide substitutions in a single mitochondrial genome, would require error rates less than 0,01% [31]. However, it has been shown that ONT reads can be used to assemble bacterial genomes despite high errors using an error correction stage [37]. Moreover, PB and ONT produce much longer reads than Illumina sequencers, and because of this they are helpful in solving problems in genome and epigenetics research [40], including detecting *Structural Variants* (SVs) [44].

[2] Kilo-basepairs

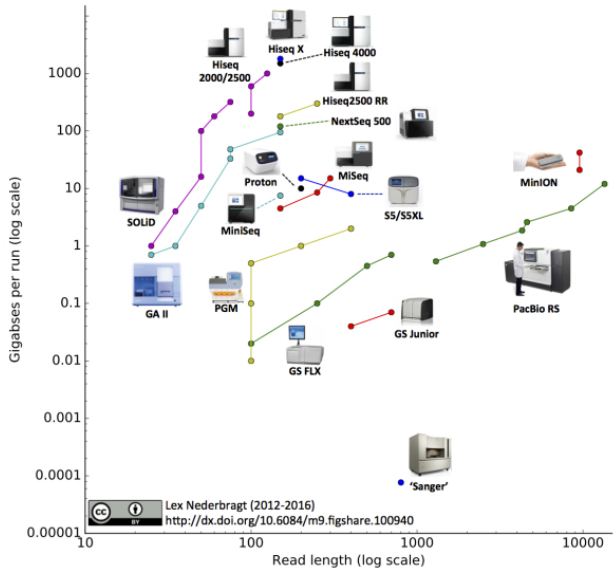


Figure 2.4: **Developments in sequencing technologies**. Different sequencing technologies with varying read lengths and throughput [43]

Technology	Illumina (HiSeq 2500)	PacBio RS II	ONT MinION
Error Rate (%)	0.1	13-16	8 - 38
Read length	2x150-2x250 bp	10 - 15kbp	up to 900kbp
Time	7 hrs-6 days	30 min-4 hrs	1 min-2 days
Reads/run	1-8x10^9	50K per SMRT cell (16 on RS II)	selected by user
Cost (USD)	0.03-0.4/mbp	0.4–0.8/mbp	500/flowcell

Table 2.1: Sequencing technology

2.1.4 Data

There are different formats for containing sequence data such as text-based formats like *FASTA* and *FASTQ*. The first line of each sequence in a FASTA file starts with a ">" (greater-than) sign, and is followed by a template description usually determined by the database in which the file is found. Following the description comes the actual sequence, encoded either as amino acids or DNA-nucleotides such as 'A', 'T', 'C' and 'G', for bases or other specialised characters such as 'B', which means anything but 'A', and 'N' which can be any nucleotide. FASTQ is similar to FASTA, but also includes a quality score for each base.

2.2 Long Read Overlap Algorithms

At the heart of many biological applications lies complex computational methods for processing and analysis of biological data. A common problem in bioinform-

atics is the *assembly* of reads into a complete genome. If the reads are collected from a new organism, these reads must be joined together, in a process called *de novo assembly*. If there already is a completed *reference genome*, one can map the reads to it (*mapping*). Common for both these problems is finding overlaps between reads, or between reads and a reference genome.

Most overlap algorithms try to find shared *seeds* between reads and to construct overlapping regions based on these. The difference in these algorithms comes down to how this task is solved. Chu et. al describes overlaps as "a broad term referring to a sequence match between two reads due to local regions on each read that originate from the same locus within a larger sequence (e.g. genome)." [23]. Where a *locus* is the position.

Chu et. al proceeds to give an explanation of overlaps, that I will give in summary. Overlaps are partial or full, and are either *contained* or *dovetailed*. Contained means that one read overlaps within the other, and dovetailed means they overlap on their edges. The first step of an overlap algorithm is to localise *overlap candidates*. This is done by pairing two-and-two matching seeds, which for the case of algorithms for long noisy reads usually consist of short k-mers. *Overlap distance* is the relative positions between two overlapping reads. This gives a sense of how much overlap is achieved, and can be inferred based on few shared seeds. *Overlap regions* are the end result of the overlap, and gives us a start and stop location in both reads.

In the upcoming sections I will present some of the current innovations in algorithms for long reads, using mainly their respective literature as reference. Common for all the algorithms is that they employ some kind of seeding and overlap detection strategy to reduce the number of regions to align later. Some algorithms offer the ability to perform local alignment on the overlap regions directly, such as BLASR and DALIGNER, while some only provide the overlapping regions, such as Minimap and MHAP [23]. ONT tailored algorithms such as GraphMap and Minimap forgo the computationally demanding error correction phases utilised in previous in some of the other long read overlappers in exchange for other methods, while still retaining the *sensitivity* [23, 6, 13].

2.2.1 K-mers

K-mers are k length sub-strings used in bioinformatics and computational genomics, and are much akin *n-grams* used in computational linguistics and probability [45]. K-mers are useful in bioinformatics as a way of providing small seeds in sequences, and are frequently used in sequence alignment and assembly [46].

An example of a k-mer with $k = 3$, of the word "apples" results in 'app', 'ppl', 'ple', and 'les'. We can determine the number of k-mers K using the formula:

$$K = L - k + 1$$

Where L is the length our original word and k is the length of each k-mer.

One of the challenges with k-mers is the storage capacity. By example, if we have a sequence of 500 bp, and $k = 15$, the total number of k-mers is 486. the

amount of storage required on a computer is then 486*15*2 = 1823 bytes. For larger sequences, this will potentially require a large amount of storage space.

Using k-mers as seeds will not only result in a high use in storage, but comparing each individual k-mer in a query to a target will also affect the runtime of the algorithm the larger the sets become. Overcoming this obstacle therefore becomes one of the main design challenges when creating new bioinformatics algorithms.

2.2.2 MHAP

MinHash Alignment Process (MHAP) maps long noisy reads by using probabilistic, locality-sensitive hashing [8]. Probabilistic means that accuracy is traded for more efficiency. MinHash is a dimensionality reduction method. Basically what this means is that it "reduces the dimensions" by getting rid of some features that are not as helpful, in terms of solving the problem.

MinHash was originally created to determine the similarity of two sets, but in this case it is used to "create a more compact representation of sequencing reads" [8].

The first step of the algorithm is to create a sketch of each read by funnelling all k-mers through a series of hash functions. A sketch in this scenario is a H length set of hash values, corresponding to the number of hash functions, thus determining the sketch size. The minimal hash value, called the *min-mer* for each hash value is stored in the sketch. When the sketches are done, they are compared to each other by measuring their likeness using the *Jaccard index*. Jaccard index is found by using the following formula, where A and B are sketches.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

In terms of computational costs, it is preferable to keep the sketch sizes small, as the sketch size is proportional to the time required to hash the k-mers, and to store and compare the sketches, but by reducing the sketch size it also decreases the sensitivity [8].

The sketches that pass the criteria (based on jaccard similarity) move on to the second part of the algorithm. Here the algorithm filters based on the k-mer count for the overlapping region using a sort-merge algorithm. Outputs are finally presented in a MHAP output format [23].

MHAP is designed primarily for SMRT but it is likely suited for nanopore sequences as these are expected to have similar length and error characteristics [8].

2.2.3 BLASR

Basic Local Alignment with Successive Refinement (BLASR) is a search algorithm proposed by Chaisson et. al, for aligning short matches to a target genome [22]. The algorithm makes use of a *BWT-FM Index* or *Suffix Arrays*

to search the target genome for short matches, ie. seeds. These seeds are then extended using methods that will be described below.

The "successive refinement" has three phases: "(1) detecting candidate intervals by clustering short, exact matches; (2) approximate alignment of reads to candidate inter- vals using sparse dynamic programming; and (3) detailed banded alignment using the sparse dynamic programming alignment as a guide" [22]. The three steps are all illustrated in Figure 2.5.

Detecting candidate intervals by clustering short exact matches, is done using a BWA-FM index or suffix array generated on a genome.

BWA-FM index, and suffix arrays are different data structures that are useful for compressing large data, especially strings.

"A suffix array is a sorted array of all suffixes of a string" [47]. Taking for example a string "*apples$*", this gives us in sorted order: "*$*", "*apples$*", "*es$*", "*es$*", "*ples$*", "*pples$*", "*s$*".

A *Burrows-Wheeler Transform* (BWT) is generated by rotating the input string and generating separate strings for each character, this step is proceeded by sorting the new strings lexicographically, and extracting the last column, ie. the last character of each string. This technique groups the same characters together, which which is useful for compression later on.

The *FM-Index* (Full-text index in Minute space), is somewhat based on the BWT, but in addition to extracting the last column, it keeps track of a table $C[x]$, which is the total amount of occurrences of a symbol less than x lexicographically. The second table $Occ(c, i)$, gives the number of occurrences of the symbol c, in the prefix $BWT[1 .. i]$.

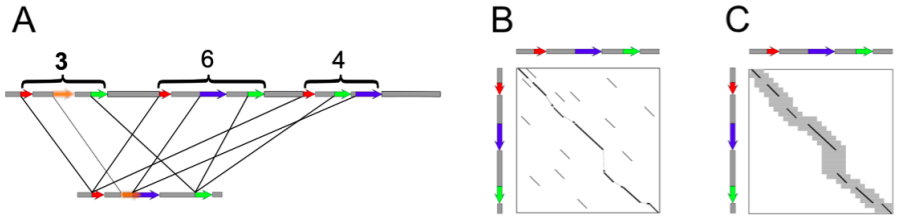


Figure 2.5: **Three steps of BLASR**. [22]. A: Detect candidate intervals B: Approximate alignments C: Detailed banded alignment.

Using these structures, BLASR builds a set of exact matches of sub-strings of a specified minimum length, between the query and the genome. These matches are referred to as *anchors*. Later, these anchors are clustered using *global chaining* [22]. This is done by sorting the anchors based on the following function

$$\sum_{a_i \in C_i} log(1/Freq(a_j))$$

Where $Freq(a_j)$ is the frequency of a_j occurring in the genome [22].

The clusters that satisfy this threshold are then aligned using sparse dynamic programming. Sparse, in this context means that one uses a subset containing

the matching parts of the two sequences to align the entire sequences.

The last step consists of aligning the matches again using banded dynamic programming based on the alignment scores created in the previous step. Finally, the results are presented in a *Sequence Alignment Map* (SAM), or similar format [23].

2.2.4 BWA-MEM

BWA-MEM is a mapping algorithm that was originally created for second generation sequencing, but as long read technology emerged, changes were done to adapt the algorithm to these reads. The main points of the algorithm are described briefly below, based on the article on BWA-MEM, by Li [48].

This algorithm uses an FM-index based structure to find seeds known as *super-maximal exact matches*.

Co-linear groups of seeds are called *chains*. Short chains that occur in longer chains are filtered out during the seeding process.

The seeds are extended by ranking each seed by its length, and the length of the chain it belongs to. Running through the seeds, in ranking order, any seed that has already occurred previously will be removed.

BWA-MEM keeps track of the score of each extension reaching the end of the query sequence. The alignment is dismissed if the difference between this score and the best local alignment is below some threshold, allowing the algorithm to choose between local and end-to-end alignments [48].

Finally, the last step of the algorithm is *pairing* two hits i and j belonging to two reads.A score S_{ij} is computed for each pair, and the highest scoring pairs are used.

2.2.5 DALIGNER

DALIGNER is another SMRT algorithm used for aligning long noisy reads. It takes a subset of reads, finds all k-mers along with their positions, sorts them lexicographically and identifies matching pairs between them. As each read is compared to every other read, this gives rise to a quadratic time complexity [7]. Nevertheless, by efficiently exploiting hardware architecture, the algorithm still runs at up to 20 to 40 times faster than BLASR, depending on the data set [7].

In the first phase, DALIGNER seeks to find seeds. This is achieved by taking two blocks of reads, A and B and finding read pairs with an optimal alignment with no more than a certain amount of differences, specified as

$$2\epsilon \cdot len(P)$$

Where ϵ is a parameter determining the level of accuracy, and $len(P)$ is the length of the seed [7]. Algorithms such as BLASR and BWA-MEM use BWT to find matches, but DALIGNER makes use of a threaded *radix sort*. Radix

sort is a sorting algorithm that sorts numbers by moving them into bins based on some key size, starting with the least significant digit (LSD).

In DALIGNER the implementation of radix is threaded, allowing the task to be split into evenly sized components. By encoding the components into 64-bit integers, the algorithm exploits the memory hierarchy on modern processors. Most CPUs usually have three layers of cache memory: L1, L2 and L3, with L1 having the fastest transfer rate. On modern processors, each core has a seperate L1 cache, thus allowing each thread to keep data on a seperate L1 cache, gaining a significant increase in efficiency, compared to storing it in any lower levels, like RAM [7]

When the sorting is done, the next step is to find the local alignment on a seed-hit between two reads. This is done using the $O(nd)$ difference algorithm [49]. The idea of this algorithm is to find the *furthest reaching points*, where given some point p in the edit graph between two sequences, the goal is to find the longest path starting from that point, with increasing amount of differences (horizontal or vertical edges) d, each of which are referred to as *d-waves* (or *d-path*). DALIGNER uses a slightly modified version of this algorithm, where it not only locates *d-waves* in the forward direction, but also the reverse. When this stage is finished, the output is presented as local alignments or in a LAS-format [23].

For two sequences A and B, an edit graph is an NxM grid, where N and M are the lengths of A and B respectively. We can use these graphs to determine the *edit distance* or Levenshtein distance between two strings. This distance is reflected by the number of changes, ie. indels and substitutions necessary to turn sequence A into sequence B or the reverse. There are several algorithms in bioinformatics for aligning two sequences by finding the best path in an edit graph. Examples being the *Smith-Waterman* algorithm for *local alignment* and the *Needleman-Wunsch* for *global alignment*.

2.2.6 GraphMap

GraphMap is a mapping algorithm proposed by Sovic et. al, that is made up of a "five-stage read-funnelling approach" [9].

Stage one makes use of a seeding strategy based on *gapped Q-grams*, that are like k-mers with "don't care" positions. The use of q-grams provides fast and sensitive look-up of inexact matches [9], and also reduces the search space for the next step of the algorithm.

Next, matching seeds are localised using an approach inspired by the *Hough Transform* (HT) algorithm. Hough transform is an image processing algorithm, usually used to recognise lines, circles and ellipses in images [50]. This is done by giving each edge pixel (often generated by an edge detection algorithm such as *Canny edge detection* etc.) in the image a "vote", which effectively increments a bin in something called the *Hough space*.

In the case of line detection, the hough space is parameterised by an angle/or slope, and the distance from the origin/the intercept, which means the hough

space can represent all lines

$$c = -mx + y$$

which is the dual of

$$y = mx + c$$

[9], which we recognise as a regular parameterised line, where m is the slope, c is the intercept and (x, y) is a point in *Cartesian space* (ie. pixel). Lines that score over a certain threshold, ie. intersections in the hough space, indicate a candidate line.

This approach is adapted by GraphMap, where each point is represented as (q, t) in 2-D space, where q (query) and t (target) are the indices in a seed hit. The algorithm operates under the assumption of a known slope m = 1, because ideally the slope of the regions should be a $\frac{\pi}{2}$ angle, and anything else would indicate the presence of indels. The goal is to find the intercept parameter c. Resulting in the equation: [9]

$$c = t - q$$

This is analogous to placing each hit into a bin corresponding to a diagonal c, as seen in figure 2.6.

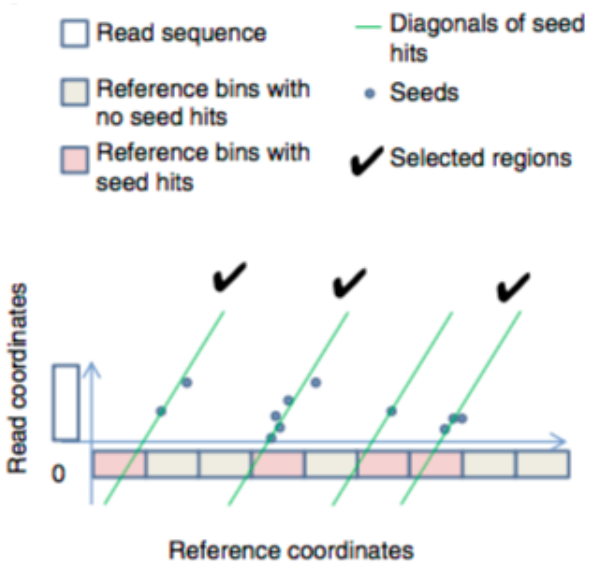


Figure 2.6: **Binning of hits**. Selected regions are chosen for bins with a certain number of hits.[9]

Where the intercept c is synonymous with the *diagonal*, and $c = 0$ being the *main diagonal* ($t = q$). The hough space in GraphMap is represented by an *accumulator array*, which is incremented for each c. The highest scoring c in this array is exact mapping position of the read and the reference [9].

The second stage, called *Graph-based vertex-centric construction of anchors*, refines the candidate regions from the previous stage. This is done by creating a *k-mer mapping graph* created on the target sequence. Here, vertices are k-mers, with l outgoing edges to proceeding vertices, ie. v_i to $v_{i+1}, v_{i+2}, ...v_{i+l}$.

By using $l > 1$, the edges serves as bridges between the vertices that alleviate some of the sequence errors by jumping over poor k-mers [9]. The goal of this stage is to identify *maximal walks*, resulting in an *alignment graph* based on a (q, t)-pair. Walks that score lower than a certain threshold are filtered out. This stage allows one to create longer alignments, despite higher substitution errors [9].

The third stage attempts to extend the anchors from the previous stage by performing the *Longest Common Subsequence in k length Substrings* (LCSk) [9]. The reason for this is that the vertex-centred anchors are not necessarily increasing monotonically. The *monotonicity requirement* basically means that the set of anchors must be either only increasing or decreasing, not both. This stage has shown to be very important, and without it GraphMaps *precision* and *recall* (measures of relevance) decrease by 10-30% [9].

The fourth stage performs L_1 *Linear Regression* to refine the alignment by filtering out outliers caused by repeats or sequence errors [9]. After the regressions step, LCSk is performed once more.

The fifth stage consists of producing the final alignment, in addition to a *Mapping quality*, given by

$$Q = -10logp$$

where p is the probability of a read being mapped to the wrong position. And finally, the output also consists of an *E-Value*, which is "... a parameter that describes the number of hits one can "expect" to see by chance ..." [51].

The output of the GraphMap algorithm is presented in either SAM or MHAP format.

2.2.7 Minimap

Minimap is an algorithm described by [6] as an algorithm for fast approximate mapping designed for long read technology.

The algorithm is influenced by Myers' DALIGNER and it's sorting schemes [7], and the idea of sketching a reduced representation like in MHAP [8], but by using *Minimizers* instead of Locality Sensitive Hashing.

In this section, I will give a summary of minimap based on Li's article [6].

Minimap, much like other mapping algorithm consists of three phases: indexing the target sequences and identifying seeds; querying the index for overlapping reads; and finally extending these overlaps.

Indexing

Indexing the target sequences is done by creating a sketch of minimizers M. The sketches are found by looking at both the forward and *reverse compliment*, choosing the smallest k-mer in the window in both directions. The reverse compliment consists of the reverse string of watson-crick complements, for example, "ATCG" turns into "CGAT".

One can describe this formally: let $\phi : \sum^k \Rightarrow Z$ be a hash function. A *double-strand* (w, k, ϕ)*-minimizer* of a string s, where $|s| \geq w + k$ *-1* is a three-tuple (h, i, r), is a minimizer such that there exists $max(1, i - w + 1) \leq j < min(i, |s| - w - k + 1)$ having

$$h = \phi(\pi(s_{ik}, r)) = min\phi(\pi(s_j + p_k, r')) : 0 \leq p < w, r'0, 1$$

[6]. As a result we end up with a set M, consisting of the triplets (h, i, r), where h is the minimizer, i is the index of the k-mer in the string, and r is the strand on which it was found.

When all minimizers are found, the set is sorted using the triplets as keys. The point is not to insert these values directly into a hash-table, but instead, the table contains the indices into the index array, using the minimizer as key. This is similar to the technique by Myers in DALIGNER. Figure 2.7 illustrates how the index table is used to reference the minimizers.

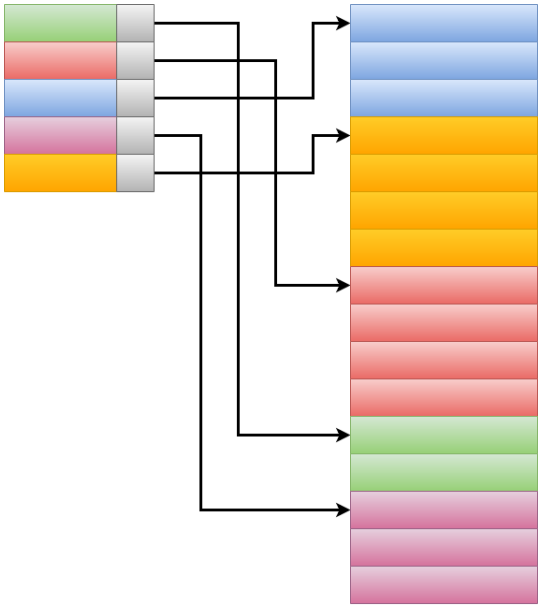


Figure 2.7: **Index Table**. Instead of searching the entire minimizer table, only the indexing table is queried to retrieve the position of the queried minimizer.

Hash Function

The goal of the hash function is to map a k-mer into an unique *2k* integer value. Often, one will encode the string having A, C, G, T transform to 0, 1, 2, 3 respectively. For example, a 3-mer "CCT", is encoded into "010111". The problem with using this scheme however, is that regions containing long repeated sections of adenine (A), known as *poly-A*, tend to be oversampled [6], because these result in zero. To overcome this, minimap uses an invertible integer hash function ϕ that maps each k-mer to a $2k$-bit integer [6]. This hash function reduces the number of hash values that map to regions with low complexity [23].

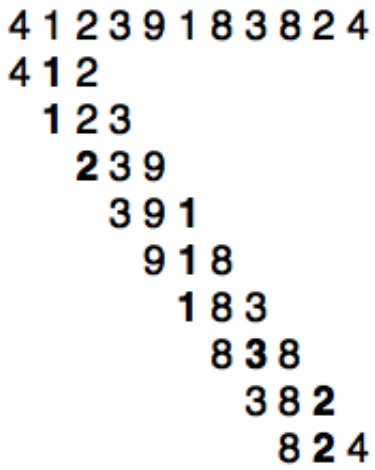


Figure 2.8: **Collecting Minimizers**.Identifying minimizers with w=3. Bolded numbers indicate the minimizer. This illustrates how there can often be subsequent minimizisers of the same value. This example results in the final output = {1, 2, 1, 3, 2}

Minimizers

One of the core concepts used in the minimap algorithm are so-called minimizers [52]. Minimizers provide a way of reducing the amount of storage required when storing large amounts of biological data used for sequence comparison, without loss in sensitivity.

In Section 2.2.1, I discussed the large amount of storage required to store all k-mers. Baring this in mind, it is easy to see why we would want to reduce the amount of k-mers we store in our database.

The minimizer algorithm functions by locating the smallest value in a surrounding window of length w. In the case of bioinformatics applications, these values are numerically encoded k-mers. This is best illustrated in Figure 2.8.

The number of windows in a sequence of length l, and with window size w is, $L = l - w + 1$. The number of k-mers stored is reduced by a factor of $\frac{2}{w+1}$ [52].

Minimizers have been shown to reduce the amount of storage required quite significantly, depending on input parameters k and w [52]. One can expect a k-mer to occur every b^k places, where b is the base, ie. four for DNA. for a total of $\frac{L}{b^k}$ matches [52]. Therefore, choosing a higher k will result in higher specificity, but lower sensitivity, and by having an $\frac{L}{b^k}$ ratio less than one decreases the chance that matches of two k-mers has occurred randomly [52].

Overlap and Mapping

The next step of the algorithm is to find overlapping regions between the target sets and the queries. The query sequence is sketched similarly to the targets, but this time without the sorting.

Any overlaps between the query s and the target s' is known as a *minimizer-hit* (h, x, i, i') and can be described formally as any minimizer (h, i, r) $\in M(s)$ and (h, i', r') $\in M$(s') [6]. The x is the result of $x = r$ XOR r', which essentially is a boolean value representing whether the two minimizers are on the same strand or not. These minimizer-hits are appended to a set A. Which consists of a 4-tuple (t, r, c, i'), where t is the target sequence, r is the strand, c is the diagonal and i' is the index of the target sequence. The diagonal c, is set as either (i + i') or (i - i') depending on whether the two minimizers are on the same strand or not. When all overlapping hits have been found for the query, A is sorted.

Finally, the last step is to cluster the minimizer hits. To do this, minimap employs a clustering of minimizer hits similar to the Hough Transform method used in Sovic et. al's GraphMap.

Seeing as A is sorted, the set is already grouped by their reference id and strand, so any hits within these two groups that have hits with diagonals c within an ϵ distance are said to be approximately co-linear, and are stored as one interval. ϵ is set to 500 base pairs by default. This distance is defined as the absolute difference between the diagonals, or formally as [6]:

$$|c_1 - c_2| < \epsilon$$

These intervals are similar to binning hits in GraphMap, but here we use a sorted array, "binning" hits within an ϵ band, instead of incrementing an accumulator array. Within the intervals, the longest increasing subsequence (LIS) is applied to find the longest chain of co-linear matches [53]. This step is similar to the LCSk stage in GraphMap that satisfies the monotonicity requirement. If the cluster contains chains with gaps that are longer than g, these are split into seperate ones.

To increase the sensitivity of the algorithm, one can decrease the window size, thus yielding more minimizers in the index. Additionally, decreasing the k-mer length will serve the same purpose, but Li reports that doing so may have influence on the overall performance on mammalian genomes [53].

Finally, mappings which satisfy a certain chain length C are printed out in the *PAF-format*. *Pairwise Mapping Format* (PAF) is a tab-delimeted output format proposed by Li as a means of illustrating the mapping results [6]. It consists of the following fields in order:

Query Name Name of the query sequence.

Query length Length of query sequence.

Query start Start index in query.

Query end End index in query.

Strand '+' or '-' indicating whether mapping occurs on the same strand.

Target name Name of the target sequence.

Target length Length of target sequence.

Target start Start index in target.

Target end End index in target.

Matching bases Number of matching bases in the mapping

Number of bases Number of bases in the mapping, including gaps.

Mapping Quality Mapping quality, given as a value between 0 - 255.

2.2.8 Comparison of Algorithms

One of the main differences between each algorithm is how they generate seeds and how they are overlapped and clustered. In this section I present a comparison on BLASR, DALIGNER, MHAP, GraphMap and Minimap based on research gathered by Chu et. al [23].

The five algorithms were originally tested on four separate *E. coli* data sets, two PB and two ONT. For each technology, one set was based on real sequencing data, and one on synthetically generated sets using *PBsim*[54] and *NanoSim*[55] [23].

The algorithms were tested based on sensitivity and precision, as well as computational performance. The *gold standard* data sets were generated using BWA-MEM alignments, and although these are not necessarily 100% accurate, they provide a good estimate because the rate of mismatch is much lower with read-to-genome overlapping than read-to-read overlapping [23]. This is because in read-to-genome overlapping, we only need to take the read errors into consideration.

Sensitivity and *precision* are measures often used in pattern recognition and classification tasks. Sensitivity is calculated as:

$$Sensitivity = \frac{TP}{TP + FN}$$

and precision as:

$$Precision = \frac{TP}{TP + FP}$$

The *F1-Score* is a way of combining precision and sensitivity together using a *harmonic mean* between them [23]. The general formula is given as:

$$F_1 = 2 \cdot \frac{Precision \cdot Sensitivity}{Precision + Sensitivity}$$

In Table 2.2 I have adapted Chu et. al's model to include results based on the average results of the non-synthetic and synthetic data sets. The results were based on the parameters giving the best F1-Score for each algorithm [23].

	Pacbio			Oxford Nanopore		
Tool	Sensitivity(%)	Precision(%)	F1-Score(%)	Sensitivity(%)	Precision(%)	F1-Score(%)
BLASR	78,5	89,2	82,25	92,55	74,05	82,3
DALIGNER	88,1	88,85	88,45	93,9	94,3	93,9
MHAP	85,65	83,9	84,80	93,15	84,25	88,45
GraphMap	80,9	95,25	87,25	90,5	94,7	92,55
Minimap	74,25	89,3	80,75	92,9	97,2	94,95

Table 2.2: **Comparison of Algorithms**. Table summarises the average results of runs using both synthetic and non-synthetic reads.

The table shows that DALIGNER gave the best F1-Score for PB reads, whereas Minimap performed best on ONT reads. The results also show that the different algorithms perform differently on different data sets. For example, Minimap performs relatively poorly on PB reads (F1-Score= 69,7%), but well on ONT reads (F1-Score=93,2%). Chu et. al hypothesise that this may be due to differences in read length distributions and error type frequencies [23].

To profile computational performance, tests were done on a "64-core Intel Xeon CPU E7-8867 v3 @ 2.5 Ghz machine with 2.5TB of memory" [23]. Running on default and best F1-Scoring parameter setups, and on different number of threads (4, 8, 16 and 32) [23]. Minimap was shown to be the most computationally efficient tool, running at least 3 to 5 times faster than all the other methods, regardless of parameterisation [23]. The rest of the algorithms varied heavily on the parameterisation and some performed many times faster than others depending on the parameters and data sets. Memory usage was also dependant of parameterisation, but Minimap used less memory than almost all other methods [23], which could possibly be credited to the use of the minimizer method.

2.3 Parallel Programming

Parallel programming is a programming paradigm that focuses on solving tasks on multi-cored CPUs as well as computer clusters and other parallel machines. With parallel computing, each thread or process is executed individually on a separate processing core, making each thread run in parallel. Parallelism must not be confused with *concurrency*, which is when multiple threads or processes are be executed seemingly in parallel, but what really is happening is that the operating system performs *context switching* between these tasks at such a high rate that one perceives them as running simultaneously.

In 1966, Michael J. Flynn proposed what has come to be known as *Flynn's Taxonomy*, a system that classifies different parallel programming models [56]. The model classifies combinations of single or multiple instructions paired with single or multiple data, namely *Single Instruction Single Data* (SISD), *Single Instruction Multiple Data* (SIMD), *Multiple Instruction Single Data* (MISD) and *Multiple Instruction Multiple Data* (MIMD).

For my work, the focus will be on MIMD and SIMD architectures, as these represent most common CPUs and GPUs respectively.

MIMD is the architecture most commonly used in regard to modern personal computers. MIMD allows for multiple different instructions to be executed simultaneously on separate data. Typically, this means we can have one process adding up some numbers, while another process searches through a text file etc.

SIMD is a parallel model that as the name indicates, executes the same instruction on multiple streams of data. By doing so, we achieve something called *Data parallelism*. An example is *Streaming SIMD Extension* (SSE) created by intel for the x86 architecture. When working with SIMD, data is distributed onto separate registers and executed. Intuitively, SIMD allows us to take two sets of data, say two arrays 1, 2, 3, 4 and 5, 6, 7, 8 and with the multiply operation, this gives us 5, 12, 21, 32. What happens is that the entire instruction is executed over the entire datasets at the same time, and not over a for-loop as one might normally do.

In addition to Flynn's Taxonomy, another class known as *Single Instruction Multiple Threads* (SIMT) exists. SIMT can be seen as an extension of SIMD combined with *Multithreading*. Multithreading is an execution model that allows for multiple threads to exist in one process. This model was introduced by Nvidia, and is what is used on their GPUs. SIMT will be discussed further in the GPU programming Section 2.4.3.

2.3.1 Measuring Speedup

When working with parallel algorithms, we often measure their performance based on the *speedup* gained compared to another implementation, for example a serial (one-threaded) one.

We can measure the speedup of one implementation compared to another based on the *latency*, ie. time. Latency is basically the amount of time units spent from the execution starts, until it ends. Thus, we can calculate our achieved speedup with the following formula [57]:

$$Speedup = \frac{L_1}{L_2}$$

Where L_1 and L_2 is the latency of the two implementations.

Ideally, we would like to have linear speedup, meaning that for each computational resource we added, ie. add one processor, we would gain a proportional speedup. However, realistically, there are some limitations. There are two common ways to predict speedup, namely *Amdahl's Law* and *Gustafson's Law*, also referred to as strong and weak scaling, respectively [58].

Amdahl's Law

Amdahl's law, named after computer scientist Gene Amdahl. Is a formula for calculating speedup. The law is based on the observation that when we speed up a portion of a system, the overall effect on the system performance depends

on how big the portion is, and how much that part is sped up [59]. Resulting in the following formula [59]:

$$Speedup = \frac{1}{(1-\alpha) + \frac{\alpha}{k}}$$

Here α is the portion which has been sped up, and k is a factor determining how much it has been sped up by. When used in the context of parallel programming, k is often synonymous with p, the number of processors or processing cores. What this model shows is that there is an upper limit to how much speedup we can achieve in total, which is based on the fraction of the portion we have improved, and that increasing the number of processors has diminishing returns. A simple way of showing this is by taking the limit of the function:

$$\lim_{k\to\infty} Speedup = \frac{1}{1-\alpha}$$

In words, the theoretical minimum latency of our implementation can be no less than $\frac{1}{1-\alpha}$.

As an example, one can consider a standard CPU containing eight cores, and the amount that is sped up is 30% of the implementation. Speedup is then calculated by:

$$Speedup = \frac{1}{(1-0.30) + \frac{0.30}{8}} = 1.36$$

if we double the amount of cores to sixteen, we get:

$$Speedup = \frac{1}{(1-0.30) + \frac{0.30}{16}} = 1.39$$

In other words, doubling the the number of cores yields roughly 8.3% better speedup.

Gustafson's Law

Gustafson's Law is a slightly more optimistic formula, in the sense that it makes no assumptions about the total amount of work required for the task. Instead, it assumes that the amount of work the implementation can solve, scales with the amount of available computational resources, ie. the problem size scales with the number of processors [60]. The formula looks like this:

$$Speedup = 1 - p + sp$$

Where p is the portion which is sped up, and s is its speedup.

2.4 GPU Programming

In the mid-1960s Gordon Moore made the observation that the number of components, specifically transistors, in integrated circuits would double annually [61]. This was adjusted to to bi-annually in 1975 [62], and has been accurate for quite some time. Increasing the clock rate at which processors run is a stable way to increase the performance of our computers. However, although transistors continue to get smaller and faster, there are certain physical limitations. Increasing clock rates requires a great amount of power, which in turn produces high temperatures, meaning that we simply cannot keep our processors cool. Therefore, increasing the number of processors instead of each processors individual speed is gradually becoming a better option.

Originally, Graphics Processing Units (GPUs) were specifically designed for rendering computer graphics, but over the recent years these devices massively parallel architecture have allowed for great developments in many scientific research areas ranging from artificial intelligence to,–of course– bioinformatics, of which is the subject of this thesis. This has lead the way for more General Purpose GPUs (GPGPUs), which are specifically designed for solving other tasks than rendering.

2.4.1 Performance Measures

As stated in the previous section, GPUs are originally designed for rendering computer graphics. We will not be going into great detail on what this entails, but it is important to understand what kind of tasks these devices are designed to handle, and in doing so, it will become easier to understand why GPUs are well suited for parallel problems.

A GPU is basically hardware that is designed to determine what each pixel on your computer screen will display at each given moment. As an example, if one has an 1920x1080 display, this is 2073600 pixels, in addition to this lets say that the monitor has a refresh rate of 60Hz. This means there are 124416000 pixel updates each second. To handle these requirements, GPUs must therefore be *throughput oriented*.

Two fundamental measures of performance that are particularly important when comparing CPUs to GPUs are latency and throughput. Latency, as mentioned in section 2.3.1, is the amount of time units spent from initialising a task and until it ends, ie. "waiting" for operations to finish. This includes, but is not limited to disk and memory accesses, idle time (time waiting while other tasks are running) and time spent on computation, ie. work done by the *Arithmetic Logic Unit* (ALU). The ALU is basically a circuit responsible for executing instructions.

Throughput is the total amount of work completed in a certain amount of time. There are two main measures of throughput. The first is the amount of bytes transferred per time unit, called *Memory Bandwidth*, and is measured in $Nbytes/second$. The second is called Computational Throughput, and is defined

as the number of instructions executed per time unit, and is often measured in *Floating Point Operations Per Second* (FLOP/s) [63].

FLOP/s is an extension on *Instructions Per Second* (IPS), and FLOP/ can be calculated with the following formulae [64].

$$FLOPS = Sockets * \frac{Cores}{Sockets} * Clockspeed * \frac{FLOPs}{Cycle}$$

For calculating IPS, the $\frac{FLOPs}{Cycle}$ part can be swapped for the number of *Instructions Per Cycle* (IPC).

CPUs are *latency oriented*. Meaning that they are optimised for minimising latency. This is achieved by having an architecture consisting of large caches for increasing cache hits, so that less time is spent on long memory accesses. CPUs also have highly elaborate control units giving good *branch prediction* and better pipeline control. Lastly, CPUs have powerful ALUs, reducing the amount of time spent per operation [65].

GPUs, on the other hand, are *throughput oriented*. Meaning that they maximise throughput. These devices make the assumption that the tasks in which the they will execute are parallel. In contrast to CPUs these devices have small, but many caches, split among the cores, thus increasing memory bandwidth.

GPUs differ from CPUs in the sense that they are specifically designed for processing large amounts of data and not for data caching and flow control, in contrast to ordinary CPUs thay are very good at running one task at the time, very quickly, GPUs can solve many tasks at once, relatively quickly.

To use an analogy, we can compare this to having one person paint a wall with a large paint brush. Alternatively, you could have five people paint the wall with paint brushes half that size. Obviously, in this scenario one person is the CPU, and 5 people are the GPU. The result would be for each time unit, the five people would complete two and a half times more work than the person painting with the large paint brush. Assuming they all stroke the paint brush at the same rate.

GPUs have no branch prediction and do not have as powerful ALUs, but are much more efficient in terms of energy. On the other hand normal CPUs usually have somewhere between 4-16 strong cores, but GPUs often have have several thousand, less powerful ones. Figure 2.9 illustrates the differences in the architectures.

With this kind of architecture it is important to keep the number of working threads high, to hide the latency, to ensure that as many cores are working at all times. GPUs are very good at efficiently launching many threads, and running these in parallel. Fortunately, GPU threads are extremely lightweight, and there is very little overhead in the creation and scheduling of them, compared to CPU threads [66].

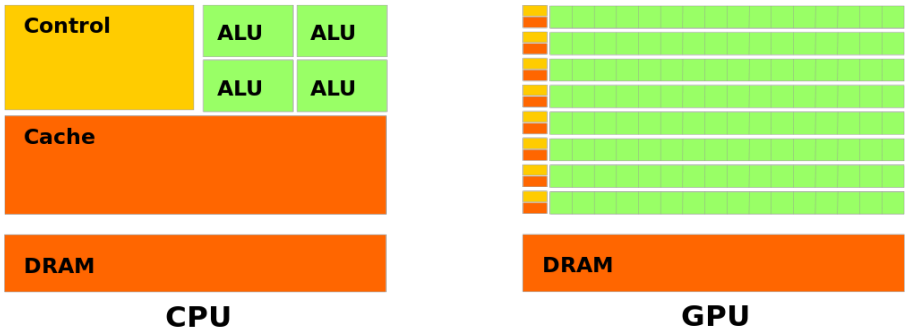


Figure 2.9: **Latency vs. Throughput Oriented Architectures** [66]. This illustrates how CPUs prioritise a larger common cache, control structures and few powerful ALUs. GPUs on the other hand divide these resources among many less powerful ALUs.

2.4.2 CUDA

Previously, writing programs for GPUs required some knowledge of graphics programming, where the programmer would have to "hack" the rendering pipeline, fooling the GPU to think the problem it was solving was a graphics one. Over the last decade or so, several APIs specialised for GPGPU programming have emerged, some of which are CUDA, OpenCL and openACC. For the work in this thesis, I will focus on CUDA, as it is the most documented and it is tailored for the hardware that I will be using. Unfortunately, this also means that it can only be run on Nvidia GPUs.

CUDA presents us with a programming model that "easily" allows us to write programs for both the CPU and GPU together. When working on computer systems that use more than one kind of processor, or technically; different Instruction Set Architectures (ISA), this is called *Heterogenous computing*. In our case, the GPU ("Device") is acting as a *co-processors* to the CPU ("Host").

CUDA supports many programming languages, but for this work I will be using CUDA C/C++. When a CUDA program is compiled, the host code is compiled using the local C/C++ compiler. Device code is compiled using its very own *NVCC-compiler*. To get access to the nvcc-compiler, one must download the Nvidia CUDA toolkit from Nvidia's own homepage [67]

Grids, Blocks and Kernels

When writing GPU code, it is initially the CPUs job to create tasks for the GPU. These tasks are called *Kernels* and can be considered as device functions. Kernels run within a *grid* of *thread blocks*, in turn a thread block consists of a grid of threads. These grids can be of either 1, 2 or 3 dimensions and it is up to the programmer to choose the layout.

Typically, one uses a 1-dimensional layout for 1-dimensional data such as vectors or strings, whereas a 2-dimensional layout is used for 2-dimensional data such as

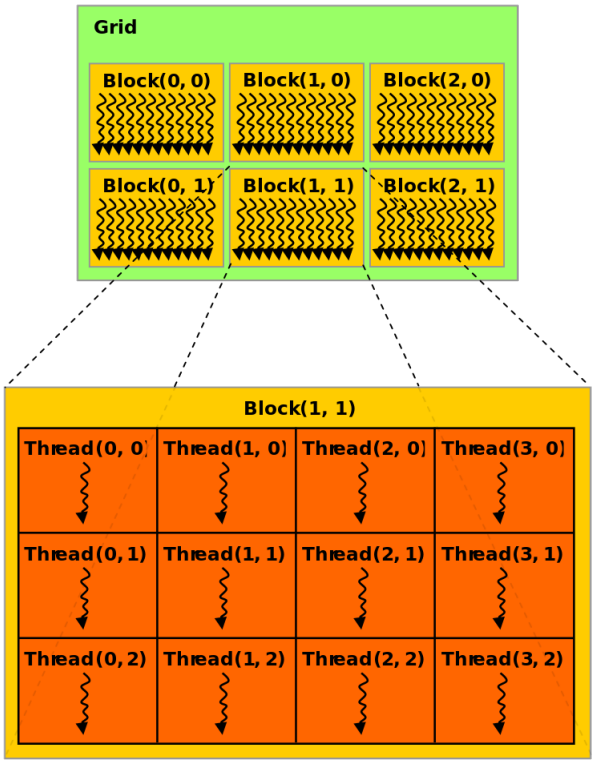


Figure 2.10: **2-Dimensional Grid of Thread Blocks**. Threads are contained within a block, contained within a 2-dimensional grid [66]

images. An illustration of how the grids and thread blocks are tied together can be seen in Figure 2.10. Choosing the correct grid and block structure is essential for gaining maximal performance, and we will delve deeper into these details later when we have a better understanding of the underlying architecture.

One of the goals when writing CUDA programs is to expose as much data parallelism as possible. A simple example of how we can do this in CUDA, is with array multiplication. Given two vectors of a given length L, we can use a 1D-layout with L threads. This way, each thread is responsible for adding together the elements at its index. This is best illustrated with an example, seen in Listing 2.1.

Listing 2.1: CUDA Example Part 1

```
1 __global__
2 void add(int * v1, int * v2, int * result, int L) {
3
4     tid = blockDim.x * blockIdx.x + threadIdx.x;
5
6     if (tid >= L) {
7         return;
8     }
```

```
9
10      result[tid] = v1[tid] * v2[tid];
11  }
```

In this example, *__global__* lets the compiler know that this is a kernel. The *thread index* is calculated using built-in variables *blockDim*, *blockIdx* and threadIdx. To clarify I have provided the complete example in Listing 2.2.

Listing 2.2: CUDA Example Part 2

```
1   import <stdio.h>
2   import <stdlib.h>
3
4   int main() {
5
6       /* Variables */
7       int L = 1024; // Length of arrays
8
9       int host_arr1[L];
10      int host_arr2[L];
11      int * host_result;
12      int * device_arr1;
13      int * device_arr2;
14      int * device_result;
15
16      /* Initialise arrays with random values */
17      for (int i = 0; i < L; i++) {
18          arr1[i] = rand();
19          arr2[i] = rand();
20      }
21
22      /* Allocate space on device */
23      cudaMalloc((void**)&device_arr1, L);
24      cudaMalloc((void**)&device_arr2, L);
25      cudaMalloc((void**)&device_result, L);
26
27      /* Copy host arrays to device */
28      cudaMemcpy(device_arr1, host_arr1, L, cudaMemcpyHostToDevice);
29      cudaMemcpy(device_arr2, host_arr2, L, cudaMemcpyHostToDevice);
30
31      /* Run kernel */
32      add<<<1, L>>>(device_arr1, device_arr2, device_result, L);
33
34      /* Copy back to host array */
35      cudaMemcpy(host_result, device_result, L,
            cudaMemcpyDeviceToHost);
36
37      /* Print results */
38      for (int i = 0; i < L; i++) {
39          printf("Value: \%d, Index: \%d", host_result[i], i);
40      }
41
42      /* Free device pointers */
43      cudaFree(device_arr1);
44      cudaFree(device_arr2);
45      cudaFree(device_result);
46
47  }
```

Above we see the host code. In most cases, one cannot use the same pointers on the host and the device, therefore we must initialise them individually. It is

normal to distinguish these with host/device prefixes. To allocate memory on the device, we use the cuda function *cudaMalloc* as seen on line 23 and then the contents of the host arrays are copied to the device via *cudaMemcpy*, on line 28. It is important to note that when using this function, one must clarify whether the copy is being done from host to device, or vice versa.

Next, the kernel is run with what is called a *launch operator*, as seen on line 32. In this example, our launch operator takes two arguments. The first argument signifies how many blocks to run on the grid, and the second how many threads per block. In this case the number of threads is 1024, equal to the number of elements in the arrays. The maximum number of threads in a block depends on the architecture, and newer architectures support up to 1024 threads [68]. Finally, we must free up the memory, as we would with an ordinary C program.

Another important concept in CUDA programming is that running a kernel is *asynchronous*. This means that the CPU does not wait for the device to finish its work, but carries on with its host code. And so by launching another kernel, it will not run before the previous kernel is done.

CUDA Memory Model

Designing an efficient program for GPUs requires the programmer to have a good understanding of the CUDA memory model. In this model, each thread has access to its own *local memory*. In addition to this, it has access to *shared memory* that it shares with each thread in its block. Finally, *global memory* is memory that is accessible to every thread on the grid. There is also *constant* and *texture* memory, used as read-only memory. An overview of the memory layout is illustrated in Figure 2.11.

When working with shared or global memory, the threads may read and write from a common memory area. This also means that the application is subject to something called *race conditions*. A race condition occurs when two threads access a shared variable at the same time [69]. This can cause problems if threads for example are attempting to increment some shared integer value by one. A scenario of this is when two threads increment the value simultaneously, then both threads may end up reading the same value, and thus incrementing the value by one instead of two. There has been extensive research dedicated to handling race conditions and other concurrency related problems over the past decades, such as mutexes, semaphores and atomics, whats common for these is that the goal impose some sort of *critical section*, where only one thread can access a certain area the time [70]. In the example, this would ensure that the first thread could read, increment and write back the new value before the second thread could do so.

There are multiple ways of handling race conditions in CUDA, the easiest of which is calling the cuda intrinsic *__syncthreads*. This acts as a barrier at which all threads in a block must wait before any is allowed to proceed [66]. Using synchronisation comes at a price, as it will cause threads to block causing some latency, however this can be reduced by having enough threads ready, so that the device can deploy new work while other threads are waiting.

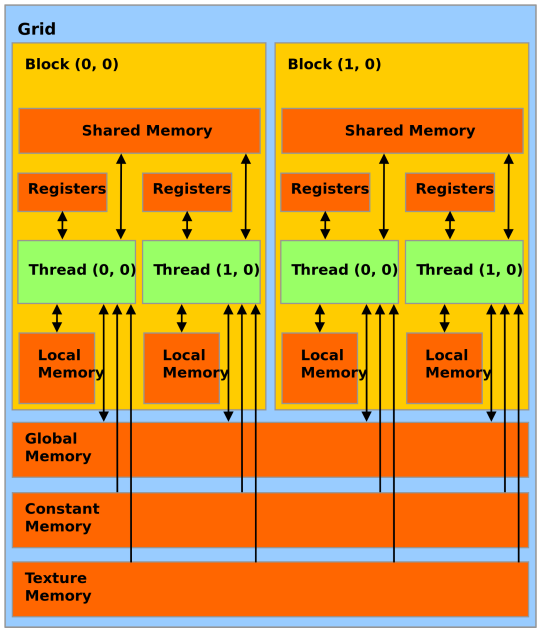


Figure 2.11: **CUDA Memory Layout**. Arrows signify the scope of each individual thread. For example, each thread has its own individual set of local memory and registers etc. [66]

CUDA also offers a wide array of highly optimised *atomic functions*, which may serve as a better alternative to synchronisation if the situation permits it. Usually when calling any standard function such as *add* or *max*, we perceive this as one single operation. In reality, the system sees it as a combination of three; read, modify and write. This leaves our program susceptible to race conditions, which may pose some issues. Atomic functions locks down the area for the entirety of these operations, until the thread is done with it.

2.4.3 Threads, Warps and Execution

Threads are are grouped in sets of 32, into what is called a warp. The two halves of a warp is called *half-warp*, and each 8-thread quarter is called a quarter warp [66]. The index of a thread within a warp is called the *lane id*.

All threads in a warp execute the same instruction, this execution style is SIMT, which is closely related to SIMD. This means is that if threads in the same warp choose a different branch (through an if predicate etc.), then the warp must execute each branch serially, as seen in Figure 2.12.

When this occurs, we get something called *branch divergence*, and the programmer must try to avoid this. Given that the branching is non-random, ie. the control flow is based on some predetermined values, and not on for example the data, it is possible to avoid such divergence by making sure that all threads

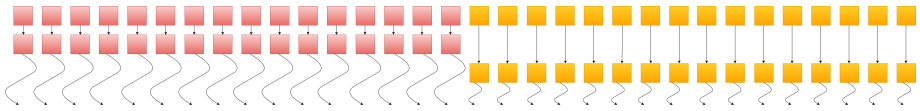


Figure 2.12: **Branch Divergence**. Illustrates how thread divergence causes a warp to execute each branch in seperate steps. Red being one branch, yellow the other.

in a warp follow the same code. An example of how this may be achieved is illustrated in the snippet in Listing 2.3 below.

Listing 2.3: Code example showing how we may avoid thread divergence

```
1  #define WARP_SIZE 32
2
3  __device__
4  void divergence_example_func() {
5
6      /* Causes thread divergence */
7      if (threadIdx.x < 10) {
8          // Do something
9      } else {
10         // Do something else
11     }
12
13     /* Doesn't cause thread divergence */
14     if ((threadIdx.x / WARP_SIZE) < 3) {
15         // Do something
16     } else {
17         // Do something else
18     }
19
20 }
```

In the first example, branch divergence will occur as threads 0 to 9 will follow one path and all other threads another. In the other example however, by dividing by the WARP_SIZE, this ensures that all warps follow the same code (first three warps in this case).

2.4.4 Hardware Architecture

The goal of this section is to give an overview of the Nvidia GPU-architecture. Nvidia graphics cards are divided intro three series, depending on their use: *Geforce* cards are mainly for "gaming"; *Quadro* is for professional workstations, often used by CAD (Computer Aided Design) users etc.; and *Tesla* which is targeted at the *High Performance Computing* (HPC) community [66]. During the work of this thesis, mainly the Nvidia Tesla K20Xm cards located on the Abel computing cluster will be used [71]. These cards belong to the *Kepler* architecture. An architecture with regard to hardware refers to the structuring and capabilities of hardware components on the GPU. Any specific hardware or implementation details we discuss from here on out, will be detailed mainly for these cards. The cards in question were launched in 2012, so there have been some significant hardware and functional improvements since then. The

functionality of each GPU is decided by what is called the *Compute Capability*. The Tesla K20Xm supports compute capability 3.5.

The Tesla K20Xm accelerator consists of one GK110 GPU, and has access to 6GB of GDDR5 on-board memory and supports PCI Express Gen2 [72]. Its *Clock rate* is 732Mhz and it has a *bus width* of 384 bit. The peak throughput for single precision floating point operations is 3,935 TFLOP/s, and the maximum memory bandwidth is 249.6 GB/s [73]

Figure 2.13: **NVIDIA GK110**. Note that the chip used for K20Xm has 14 SMXs. [68]

Streaming Multiprocessors

At the heart of the GPU architecture lies the Streaming Multiprocessor (SM). The SM was re-introduced as an SMX on Kepler hardware. The reason for which is that the Kepler architecture has the base and core clock running at the same frequency, whereas Kepler's predecessor, the *Fermi* architecture, did not. For simplicity, I will be referring to the SMXs as SMs for the rest of the thesis.

Figure 2.13 shows an overview of the GK110 architecture. Readers should note that the image shows a chip containing 15 SMs, but the Kepler K20Xm card has 14. Furthermore, the device consists of six memory controllers, responsible for handling memory transactions between the device and the main memory. Lastly, the *GigaThread engine* is a component that schedules blocks to the SMs.

The SM is given blocks to execute, and it is the job of the SM to control the warps in a process. This includes creation, scheduling, management and execution of threads [66]. Because of this, selecting the number of threads in a block should be a multiple of the warp size, to fully utilise the SM. Another thing to take into

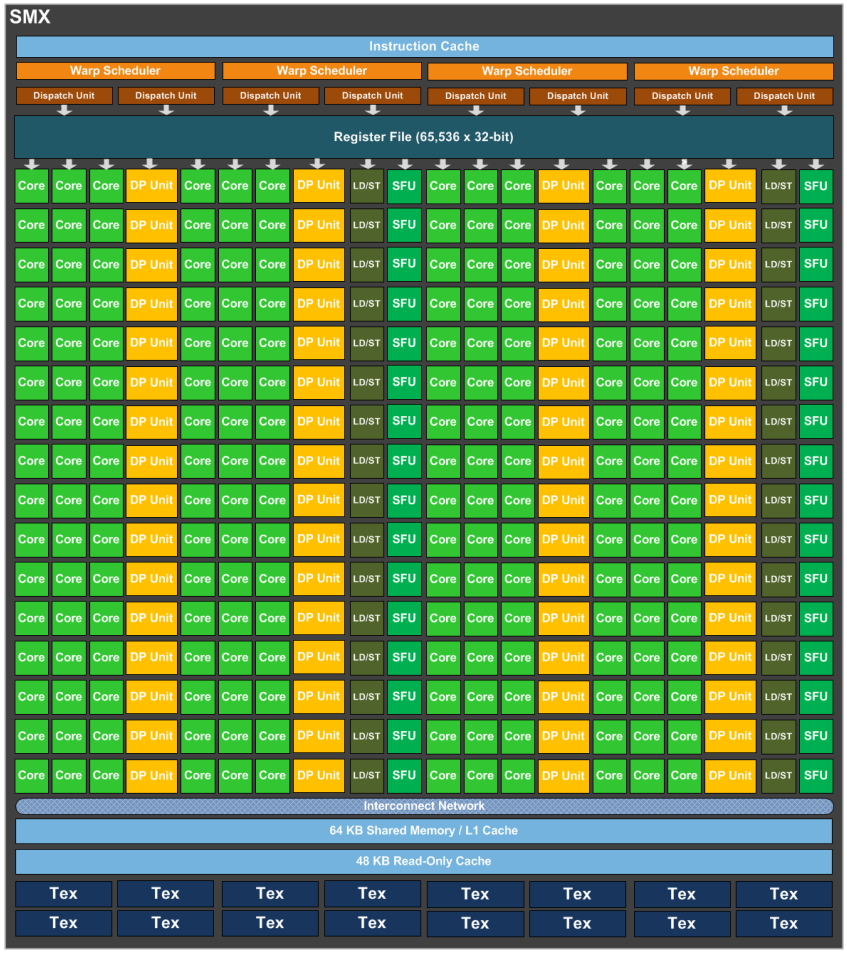


Figure 2.14: **GK110 SMX Layout**. 192 cuda cores, 64 DP units, 32 SFUs and 32 LD/ST units. Four warp schedulers handle the allocation of warps to different units. [68]

consideration when choosing the block size, is something called the *tail effect*. Which essentially is when all the thread blocks in a grid cannot fit evenly onto the multiprocessor. This means that the remaining thread blocks must run in a seperate *wave*, possibly utilising a very small fraction of the multiprocessors capacity. This performance penalty can be reduced by using smaller thread blocks [74].

One SM consists of multiple of *CUDA cores*. In Figure 2.14 we see that there is a total of 192 such cores. Additionally, there are 64 *Double Precision Units* (DP Unit), 32 *Special Function Units* (SFU) and 32 *Load/Store Units* (LD/ST).

CUDA cores are single precision, so when calculating thoughput, it depends on whether one will be using single or double precision operations. In the former case we base it on the number of cuda cores, but in the latter it is based on DP units. The SFUs are used for "fast approximate transcendental operations"

[68], such as mathematical operations like exponentiation and trigonometric functions. LD/ST units, as the name entails handle loading and storing.

Furthermore, GPUs are mainly designed for doing floating point operations. On Kepler, a single multiprocessor (SM) can execute 192 single precision operations per second (add, multiply etc.). With a warp size of 32, this translates to 6 IPC for single precision floating point operations on one SM [66].

One can calculate the peak 32-bit floating point throughput of the K20Xm device, using the formula presented in Section 2.4.1 with

$$14 * 6 * 32 * 2 * 732 * 10^6 * 10^{-9} = 3935,2\ GFLOP/s$$

Where $6 * 32$ is the full throughput of 32-bit floating point arithmetic (utilising 192 CUDA cores). The result is multiplied by 2 because CUDA cores support *Fused Multiply-Add* (FMA) operations that are capable of doing a combined multiply and add in one instruction, effectively doubling the throughput for these operations. The multiplication by 10^{-9} is to convert to Giga-FLOP/s.

However, integer operations are not capable of providing the same throughput as floating point operations. For example single precision integer addition has a throughput of 160 (5 IPC) operations per cycle, and single precision multiplication only 32 (1 IPC) operations per cycle. Moreover, there is no native support for 64-bit integer operations, and doing so will cause the operation to be emulated into two seperate 32-bit integer operations. Taking these things into consideration, it means that a integer based applications will not be able to utilise the full computational throughput of a GPU.

The SMs in the Kepler architecture contain four *Warp schedulers*. For each warp scheduler there are two *Instruction dispatch units*. This means that four warps can be issued and executed concurrently, and for warp scheduler, two independent instructions can be dispatched each cycle [68]. What this means is that there can be a total of $2 * 4 * 32 = 256$ instructions being executed in parallel, using something called *Instruction level parallelism*, meaning that the the unit identifies instructions that can be computed independently in a kernel, and executes these individually.

The warp scheduler performs context switching by choosing an active warp, ie. warp with threads ready to run, each cycle. Because context switching is performed on hardware, it comes at no cost [66]. It is important to note that the number of *active warps*, which is the number of warps that are running concurrently, is not the same as the number of warps running in parallel. The number of active warps is limited by the maximum *occupancy*, ie. the maximum number of active warps per SM, whereas the maximum number of parallel warps possible is determined by the number of available CUDA cores and warp schedulers.

The *achieved occupancy* is a measure of how many active warps there are compared to the maximum number of active warps per SM. By having higher occupancy, one can hide some of the latency caused by threads stalling (by waiting for data, synchronising etc.), by switching to other active warps. An active warp is a warp that has threads that are ready to be executed [66]. A problem that may arise by having too high occupancy however, is that the number of available registers per thread is reduced, which may lead to something called *register pressure*, which I will be discussing in the next section.

When an SM is given a set of blocks to execute, these blocks cannot migrate to other SMs, and the SMs cannot be given new work before the blocks are finished. This is why it is so important to configure the number of threads per block optimally, so as to make sure the SMs always have enough warps to run.

SM Memory

The memory hierarchy of an SM is much like that of a CPU, having small amounts of fast memory such as registers closest to the cores, and large amounts of relatively slow memories such as RAM further away.

Register File Register memory is the fastest memory available on the GPU, but is only visible to the thread, and its life-time is the same as that of its thread. If a task requires more registers than are available, this leads to something called register pressure. Register pressure should be avoided, as it can cause data to *spill* down to the slower memories [66]. Each SM on the GK110 GPU consists of 65536 32-bit registers, allowing for up to 255 registers per thread [68].

Shared Memory and L1-Cache Each SM in the Kepler GK110 architecture has 64KB of configurable memory, that is split between shared memory and the L1-Cache. L1 caching in Kepler GPUs is reserved only for local memory accesses, such as register spills and stack data [75].

Read-Only memory The SMs also have 48KB of dedicated *read-only memory*. This memory essentially makes it easier to access texture memory. What's useful about this is that it is cached [66], and it allows the programmer to use this memory for read-only data, thus freeing up more space in the shared memory for writable memory.

L2-Cache A 1536KB L2-cache is shared between the SMs. "The L2 cache is the primary point of data unification between the SMX units, servicing all load, store, and texture requests and providing efficient, high speed data sharing across the GPU." [68].

RAM 6GB of RAM is placed off-chip on the card.

2.4.5 Efficient Memory Accesses in CUDA

Knowing how to utilise the CUDA memory hierarchy is crucial for gaining any significant performance, and in this section I will present some of the basic techniques used when handling memory transactions in CUDA.

Global memory bandwidth is relatively low, when compared to other on chip memories. Because of this, one should keep these transactions at a minimum. One should keep the ratio of arithmetic instructions to words of memory transferred, called *Arithmetic intensity* [76] as high as possible. Tasks which have low arithmetic intensity, are called *Memory bandwidth limited* or *Memory bound* [77], as the amount of cycles spent solving the task is lower than the number of cycles spent retrieving the data. In the opposite case, the task is *Compute limited* or *Compute bound*.

Global Memory Accesses

When threads in a warp load data from global memory, it can access up to 128 bytes of data per transaction, this is also called the *load granularity*, and has to do with the fact that the L2-cache lines are 128 byte [66].

Given a warp makes a memory access, where each thread requests 4 byte each. This results in a single 128 byte memory transaction. However, the number of cycles required for each thread to retrieve its data, depends on how the requested data is arranged in memory. If each of the 128 bytes lie consecutively, this only requires one transaction, thus utilising our global memory throughput perfectly. This is called *Memory coalescing* [66]. If however, the requested data is scattered far apart, then these accesses cannot be combined and will require multiple cycles. When a thread accesses memory in a pattern where the data is located non-consecutively, it is called a *strided access* [66].

For example, with a stride of four, this would cause only one quarter of the memory retrieved to be used, whereas the remaining three quarters are *over-fetched*, so that a total of 512 bytes of data is transferred, although only 128 bytes are needed, resulting in 25% *load/store efficiency* (128/512).

Figure 2.15 shows the results of varying stride sizes on an older CUDA architecture, although this trend tends to be similar. As we can see, avoiding misaligned or strided accesses is crucial for our performance. In some cases, such as with 2 or 3 dimensional arrays, strided accesses cannot be avoided. In these cases using shared memory is often a better solution [78].

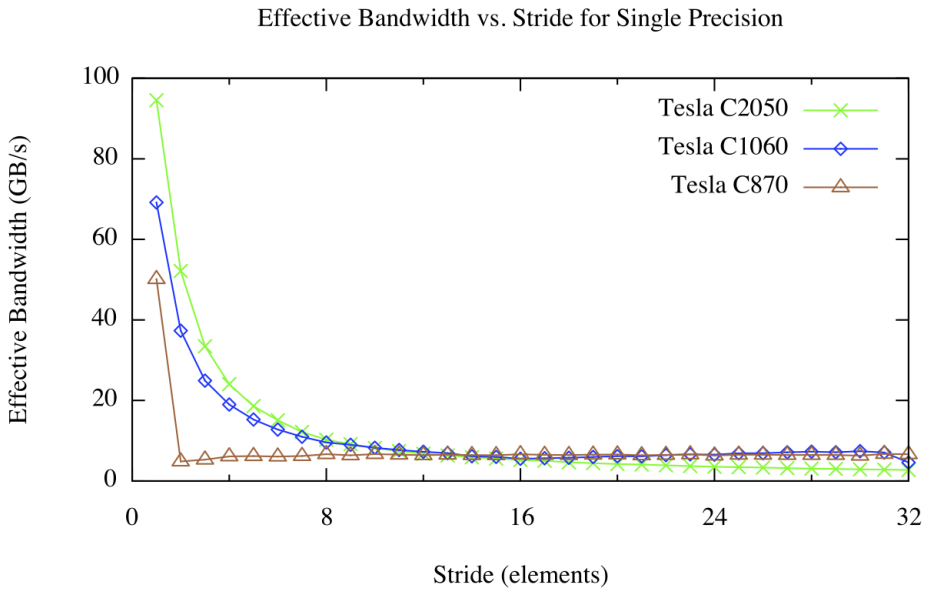


Figure 2.15: **Effects of strides Global Memory accesses**. Image shows how increasing strides cause severe loss of effective memory bandwidth. [78]

If the requested amount of data requested per thread exceeds four bytes, this requires multiple transactions. Also, it is important to note that the data types in global memory accesses must be aligned to 1, 2, 4, 8 or 16 bytes, or any multiple of these. As long as they are, the transaction will be compiled to one single instruction [66]. Unaligned data types can be solved by using *padding*.

Shared Memory

Shared memory has much lower latency and higher bandwidth than global and local memory, and it does not require the programmer to deal with memory coalescing. However, there are some potential problems to consider, to utilise shared memory correctly. Shared memory is split into 32 *memory banks* of 4 byte words each by default. As there are 32 threads in a warp, and 32 memory banks, each memory bank can service one memory request each simultaneously. The bandwidth is 4 to 8 bytes per bank per cycle, on each SM, depending on the access pattern [74].

However, if two threads from the same warp were to request different words from the same bank, this would cause a *bank conflict*. A memory bank can only handle one request at the time, so bank conflicts would cause the accesses to be serialised, thus reducing the effective memory bandwidth. Bank conflicts only occur within a warp however, and if the threads in the warp are requesting the same word, there is no conflict and the contents of that memory is broadcast to the entire warp. Kepler devices support both 4-byte (default) and 8-byte banks, and the 8 byte mode can be utilised to improve bandwidth for 8-byte accesses [75].

Shared memory, as with global memory, is susceptible to the pitfalls of strided accesses. Although in the case of shared memory, these pitfalls are bank conflicts. For example, if each thread requests four bytes with a stride of two, then every second bank is queried twice, and causes a 'two-way' bank conflict. Therefore, it is important to structure the data in such a way that one can avoid these scenarios.

2.4.6 CUDA Streams

Transferring data between the host and device is often a major bottle-neck in terms of performance. Because of this, it is desirable to minimise these transactions as much as possible, and try to do as much work on the data when possible on the device. In my case, I will be transferring biological data, which often consists of several gigabytes. These transactions often take a relatively long time compared to the computations. One way of decreasing this is by using *CUDA Streams*.

Streams can be considered as separate queues of device work [79], where the host places work in a queue and continues immediately. Operations within the same stream are ordered and cannot overlap, but operations on separate streams can [79]. All work that has not specified a stream is run on the default stream [66].

Streams enable work to be run on the GPU independently. Consequently, it may be easier to utilise the maximum capacity of the device, if kernels are small. If smaller kernels are launched on the same stream, then all this work will be done serially, and each kernel will only be taking up a little part of the GPUs capacity. By splitting the work into multiple streams, one can attempt to saturate the GPUs capacity to a fuller extent. There is no realistic limit to how many streams that can be created, but the Kepler architecture supports up to 32 connections between the host and device [80], and can be altered by setting the environment variable *CUDA_DEVICE_MAX_CONNECTIONS* to the desired stream size (it defaults to 8) [75].

One should bare in mind the overhead involved in creating and launching each stream as well, so casually increasing the number of streams beyond that which is supported probably won't give any performance boost. Furthermore, although Kepler supports 32 concurrent kernels, it is hard to actually utilise them effectively as the hardware limits the number of threads that can run concurrently. But for applications planning to run many small kernels, this may be a good solution.

One of the most practical uses of streams is to overlap data transfers between the host and device, as discussed in a developer blog post by Harris [81].

The goal of overlapping transfers is to run many smaller transfers concurrently with streams, instead of one large one. This also means that kernels are able to start executing sooner, instead of waiting for the entire set of memory to be transferred over first. In Figure 2.16, I visualise how this works. The top bar is transferring data to the device, executing the kernel, and transferring it back, on the default stream. The set of four bars shows how overlapping the transfers may reduce the total amount of time spent. The reason there is a slight gap between each stream is that some overhead is incurred by the kernel launch on the CPU.

Streams are created using *cudaStreamCreate*, destroyed using *cudaStreamDestroy*, and to make memory transfers asynchronous, the *cudaMemcpyAsync* operation is used.

To be able to do asynchronous transfers between host and device memory, the requested section of host memory must be *pinned*. Modern computer systems use something called *Virtual Memory*. In short, this means that data stored in physical memory (RAM), divided into something called *pages*, that can be swapped out and onto disk. When memory is pinned, it means that it cannot be swapped out, and is effectively *page-locked* [70]. When a transfer is issued normally (ie. using cudaMemcpy), new pinned memory must be allocated first, then the requested data is copied into this memory, before it is transferred to the device. If the memory is pinned however, the GPU can interact directly with the something called the *Direct Memory Access* (DMA), without the involvement of the CPU, and will lead to higher bandwidth utilisation [82]. The downside of using pinned memory is that it may cause the host system to slow down if too much memory is being pinned.

To allocate pinned memory, we use *cudaMallocHost*. to pin/unpin already allocated host memory, *cudaHostRegister* and *cudaHostUnregister* is used.

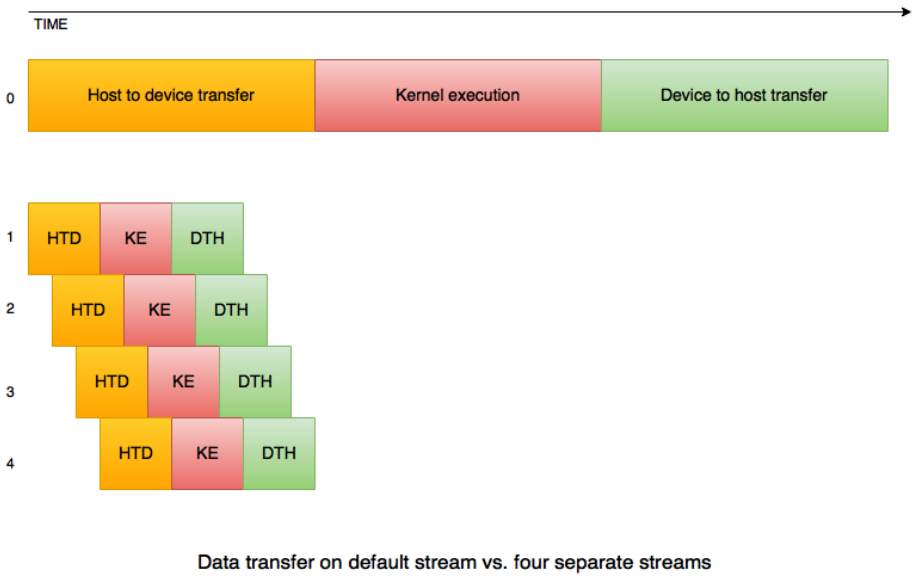


Figure 2.16: **Data Transfer With Four Streams**. The top bar (0) illustrates kernel execution running one (the default) stream. 1 through 4 shows how it is possible to gain a speedup by separating the work into multiple streams.

2.4.7 Libraries

There are several CUDA enhanced libraries available, some of which are included in the CUDA toolkit, such as the *Thrust Parallel Algorithms* library, which is a C++ template library for CUDA based on the *Standard Template Library* (STL) [83]. Other libraries such ash *CUDA Unbound* (CUB) are available for download online [84]. These libraries are all created to help the programmer solve various tasks, and include many helpful parallel algorithms, so that the programmer can avoid implementing many CUDA kernels. Also for every toolkit update, libraries such as Thrust are updated with any new optimisations of the algorithms, making sure that the best implementation always is available.

One thing worth noting about Thrust is that it uses different algorithms based on the input to the thrust call. So by invoking for example the *thrust::sort* function, thrust may use radix sort, or it may use merge sort, depending on what kind of data structure is passed.

Thrust and CUB are modular, and often allows the programmer to pass different predicates to functions, so long as it satisfies the *interface*. For example, when using a stream compaction function such as *thrust::remove_if* it is possible to specify a predicate such as "less than" or "equal to" etc. to signify which condition an element in an array should be removed on.

2.4.8 Profiling

The CUDA toolkit also includes some useful profiling tools. *nvprof* is a terminal based profiler much resembling *gprof* for ordinary CPU programming. The *Nvidia Visual Profiler* (NVVP), offers a visual representation of the profiling results. It also allows the user to examine individual kernels, and to analyse useful statistics such as occupancy, register pressure and memory bandwidth. This way it is easier to tweak and optimise the implementation to better fit the hardware, and it also makes it easier to recognise and manage any potential bottlenecks.

2.4.9 Parallel Patterns and Algorithms

Parallel patterns concerns how we distribute data among tasks. In this section the aim is to describe some of the most common parallel programming patterns and to give a brief introduction to some commonly used algorithms.

Map

The *Map* pattern uses a one-to-one pattern, where a thread maps a function onto data from one index, into another. An example of the map pattern in cuda was given in the code (Listing 2.1) provided in Section 2.4.2. An illustration of the map pattern is provided in Figure 2.17

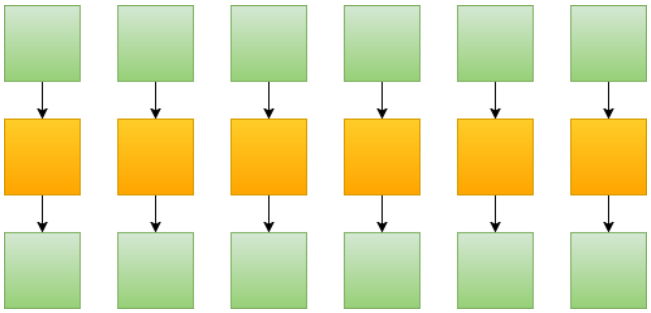


Figure 2.17: **Map Pattern**.Yellow blocks are threads, whereas green blocks are data entries.

Scatter and Gather

With *Scatter*, threads distribute a value based on one data entry into multiple. An example use of the scatter operation is in image processing where the value of one image pixel may be distributed onto neighbouring ones.

The *Gather* operation is somewhat the opposite of scatter, as it uses data from several entries to compute some result into one. Using image processing as an example again, taking the average of several neighbouring pixels and writing them into one is one use of the gather operation.

The access patterns of scatter and gather can be either *Dynamic* or *Static*. Dynamic is the case when the indices change, whereas static has some predetermined pattern. Static is often better, as it allows for some prediction in terms of optimising memory accesses [85].

Stencil

A *Stencil* is a special case of the gather operation using a neighbourhood pattern. One way to apply a *1-D Stencil* is when calculating k-mers. Each thread looks at k different indexes to calculate one k-mer. Figure 2.18 illustrates a stencil working on a neighbourhood of size three.

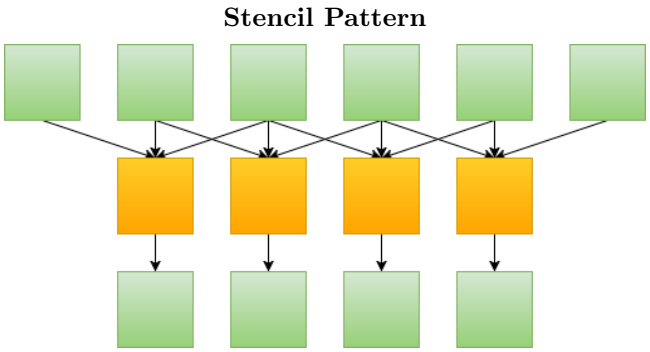


Figure 2.18: Threads acting on three neighbouring data entries.

Stream Compaction

Often we find ourselves in a situation where we want to remove or retain certain elements on a group of data, based on some predicate. For this we use a method called *Stream Compaction* or *Filters*. For ordinary serial programming, filtering can be done by copying elements from one array that satisfies the predicate over to another array. The program keeps control over the new index to place each element by incrementing a counter.

Doing this on GPUs poses some challenges, as each thread may be working on an unique index. Whenever a thread needs to copy an element over, it also needs to know where. One cannot simply have one counter shared by all threads, because this will undoubtedly lead to race conditions. A solution to this problem is by making sure that each time a thread adds a new element, it increments the counter using an atomic operation, such as *atomicAdd*. However, this may lead to poor performance, due to the *degree of collisions* on the atomic operation [86], ie. how many eligible threads trying to perform the operation at one given time.

If the application requires filtering, using Thrust or CUB functions may be a viable and simple solution. However, for uses where one can expect the degree of collision to be high, there is a better solution proposed by Adinet, in Nvidia's

developer blog [86]. This method shows a significantly higher bandwidth than libraries such as Thrust.

One important thing to note though is that this solution does not retain the relative ordering of the set, something Thrust and CUB does. The solution uses so-called "warp aggregated atomics". *Warp Aggregation* is "the process of combining atomic operations from multiple threads in a warp into a single atomic" [86]. Furthermore, the author describes the steps of the algorithm as follows: "

1. Threads in the warp elect a leader thread.
2. Threads in the warp compute the total atomic increment for the warp.
3. The leader thread performs an atomic add to compute the offset for the warp.
4. The leader thread broadcasts the offset to all other threads in the warp.
5. Each thread adds its own index within the warp to the warp offset to get its position in the output array."

Finally each thread adds its position to the new array [86].

Reduce

Reduce is an algorithm that takes a binary operation and applies it to a set. For example, an *add* operation will result in the sum of the set, while the *max* operation will result in the maximum value. Parallel reduction can be implemented using $\frac{N}{2}$ threads in O(log n) steps, as opposed to O(n) steps in a serial implementation. An example of how reduce may be implemented with the *min* operation is illustrated with the following code example in Listing 2.4.

Listing 2.4: Parallel reduce min in CUDA

```
1
2  __global__
3  void reduce_min(int * src, int result, int N) {
4
5      int tid = blockDim.x * blockIdx.x + threadIdx.x
6
7      /* Loop log n steps */
8      for (int i = N/2; i > 0; i>>=1) {
9
10         /* Valid threads continue */
11         if (tid < i)
12
13         /* Minimal value is stored in thread position */
14         src[tid] = src[tid] < src[tid+i] ? src[tid] : src[tid+i];
15
16         /*
17          * Synchronisation is necessary to avoid
18          * race conditions across loops
19          */
20         __syncthreads();
21     }
```

```
22
23      /* Result is stored in first index of the array */
24      result = *src;
25  }
```

Visually, reduce can be illustrated in Figure 2.19. Observant readers may recognise this as the gather pattern. One may also notice that half of the threads will turn idle after each iteration. Therefore, it is advised to split the reduction across many blocks as to reduce idle threads and increasing occupancy.

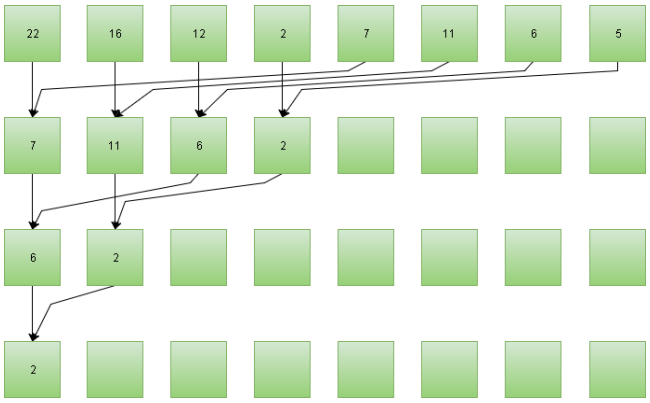


Figure 2.19: **Parallel Reduce**. Reduction on 8 data entries is done in 3 steps. Notice how the number of working threads is halved for each step.

Chapter 3

Implementation

In this chapter I present one serial and one parallel implementation of Li's Minimap algorithm [6]. The Minimap algorithm was chosen as it showed great promise in terms of both accuracy and performance.

The goal of writing a parallel implementation is to gain as much speedup as possible. In the results presented in Section 2.2.8, Minimap was shown to have the best computational performance [23]. I hypothesise that the same advantages that make Minimap efficient, may scale to a massively parallel architecture such as GPUs. Additionally, from a mere subjective point of view, I personally found the Minimap algorithm to be an interesting one as it combined some of the most efficient features of many of the other algorithms.

All the overlap algorithms discussed so far are all parallel on CPU, with Minimap being no exception. From this point of view, the question becomes whether adding more cores would speedup the implementation further, considering Amdahl and Gustafson's laws.

Based on some initial profiling done on the original implementation available on *Github* [53], I was able to make out roughly what parts of the algorithm that tended to use the most time. The part that used a considerably long amount of time per invocation was the sketching routine. Sketching is a seemingly *Embarrassingly Parallel* [87] problem, this is because sections can be split on the k-mer and window sizes, I therefore speculated that minimizers could be generated in a very data parallel fashion. Further, Minimap makes frequent use of sorting, making it possible to make use of the highly optimised radix sorting implementations provided by Thrust or CUB.

As it was found to be many discrepancies between the article's presentation of the algorithm, and the actual implementation done by Li, and because the code available was hard to understand due to highly optimised C code and little documentation, I decided to make my own serial implementation based mostly on the article. Further, many of the techniques employed in Li's own implementation, were not reflected by the article, thus making it hard to make any precise assessment of how these parts worked, and what their purposes were.

Also, it is easier to compare the parallel implementation to a purely serial imple-

mentation. Li's solution is not possible to run with only one thread, even when specifically telling the program to do so. By using any task manager such as *top* or *htop* on linux showed that there still were more than one thread executing, even if the number of threads were set to one. The reason for this seems to be that one can only determine the number of extra worker threads, and not the total number, as it seems to be at least two threads running regardless of the configuration. Writing my own serial implementation also made it easier to gain a better understanding of the algorithm details, which also helped create a better parallel solution later on.

First the serial implementation will be described, then a thorough documentation of the parallel algorithm and implementation will be presented. No specific development method was followed during the development of both the following programs, but it was always a goal to write optimised, yet clean and easily understandable code.

3.1 Parameters

Parameters for both algorithms stay the same. Here is a list describing those used in my implementations.

Target File FASTA-/FASTQ file containing target/reference data.

Query File FASTA-/FASTQ file containing query data (ie. reads).

k-mer size Sets k-mer size (k).

Window size Sets window size (w)

3.2 Serial Minimap

The serial version is implemented more or less per the description given in the article although some parts were adapted from Li's own implementation when possible. The program was written in C, as this allowed me to use both some of Li's code and to reuse some of the code written in this section in the parallel implementation if needed. In this section I give overview of the implementation details from a programmatic perspective. In section 2.2.7 I gave an overview of the algorithm from a relatively abstract perspective, mostly focusing on the article[6] description. Here I attempt to give some insight into how the algorithm is implemented into a program.

3.2.1 Implementation

In this section the aim is to give an overview over some of the design choices made in terms of data structures and other algorithms used when implementing a serial Minimap. This serial implementation was primarily made to have a reference point to compare the parallel one to, so this section will be relatively brief compared to the parallel section.

Reading Input Files Using KSEQ

All input files are read using a FASTQ/FASTA reader called *KSEQ*. KSEQ provides a structures containing useful sequence data such as sequence length, name and a pointer to the sequence itself.

Minimizers

Minimizers are stored in a 2x64-bit structures, similarly as in Li's implementation. The first 64 bits hold the minimizer value, and the next 64 is packed consisting of 32 bits used as the reference id, 31 bits for the sequence index and 1 bit for indicating what strand the minimizer is on. This allows k up to 32, maximum 2^{32} unique targets and sequences up to 2^{31} bases long. Having 2^{32} unique targets is very unlikely, but the memory saved by having smaller data types is likely outweighed by the performance gain of having the data types aligned to 64-bit in memory. Furthermore, it is likely that any optimisations done by the compiler will include padding of the data types to achieve this anyway. Figure 3.1 shows how the bits are set up in a minimizer data type.

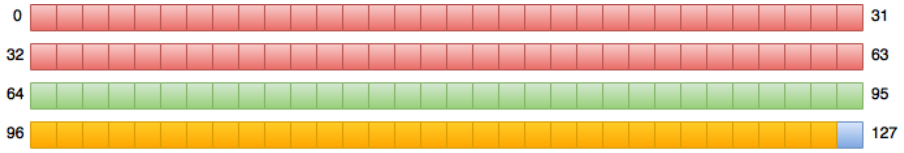


Figure 3.1: **128-bit minimizer**. Red: hash value. Green: reference id. Yellow: sequence index. Blue: strand.

Sketching

The sketching of minimizers was done using Li's sketching code, as it outperformed most of my own attempts, when tested during development. The algorithm is quite complex, but in summary it works by sliding a k-sized window through the sequence string using a loop, and continuously encoding and hashing the forward and backward strands of each k-mer. Ambiguous [1] or symmetric [2] k-mers are skipped, and a w-sized queue keeps track of minimal hash valued k-mers.

"Winners", ie. the smallest in the window are packed into minimizer data types and appended to the minimizer array, and any repeating minimizers are skipped, as the goal is to only keep one representation. The encoding is done using a look-up table, copied from Li's own code. This table takes one byte and returns an integer between zero and four, corresponding to DNA nucleotides. Any values greater than three are considered to be ambiguous and are not included.

[1] k-mers containing non DNA bases
[2] forward and reverse complement are identical

Minimizer Set

Minimizers are stored in an array located in a structure also including the total number of minimizers and additional data related to memory allocation. To avoid over-allocating memory, and to reduce the number of memory allocations, the dedicated memory for the minimizer set is increased using a block allocation scheme with a fixed block size.

When the sketching is done, the array of minimizer is sorted using radix sort, similarly to how k-mers are in DALIGNER [7]. For this application I wrote a 128-bit radix sort algorithm as I could not find any suitable programs available. Because of the structure of the minimizer data type, sorting the array gathers all the minimizers of same hash value together. Within these clusters the minimizers are subsequently ordered by the reference index, sequence index and strand. Only the 64 least significant bits need to be sorted on the minimizer table as 64 most significant bits are sorted due to the way the minimizers are inserted to the table.

Indexing

One way of indexing the minimizers is to insert them into a hash table based on the hash value. However, Myers [7] shows that by sorting and merging the arrays, we can achieve better cache efficiency [6]. Therefore, we instead insert the index of the beginning of each group of minimizers (minimizers with same hash value) into the hash table. By doing this, Li also claims that we reduce the number of heap allocations and cache misses. It is quite easy to convince ourselves of this, firstly because we do not need to allocate new arrays to hold the minimizers of each value in the entries, and secondly cache misses are reduced due to the fact that all minimizers lie in the same memory area, thus reducing the number of random accesses.

Ideally, we would like to use the hash value of the minimizer to index directly into the table, as this would eliminate the need for an additional hash function, and guarantee no collisions. Unfortunately, due to the large key space of these values ($2k$ bits), this could result in a very large hash table. Instead, the hash table is implemented using open addressing, with quadratic probing. The table size is set to a function of the total amount of minimizers, with a *load factor* of 75%. The hash function used is a simple *modulo* of the table size.

Insertion is done by looping through the array of sorted minimizers, keeping a temporary index into the beginning of each group of minimizers. For each minimizer of the same value, a counter is incremented. When a new minimizer occurs, the minimizer value, the index and the count are inserted into the hash table.

Overlap Detection

Overlapping is implemented by first sketching a read. Next, each minimizers hash value is used to lookup into the hash table. On each hit, we retrieve the index into the beginning of the minimizer group and its length.

For each hit, a new entry is appended to the overlap set per the rules described in section 2.2.7. The maximum number of occurrences for each minimizer can be limited by defining *MAX_MINIMIZER_OCC*, which currently defaults to eight. The overlap data type is also a 2x64-bit data type, similar to the minimizer. An overlap is structured as follows; 32-bit reference id ; 1-bit indicating whether overlap occurred on the same strand; 32-bit diagonal; 32-bit index into target. Figure 3.2 shows the layout of an overlap type.

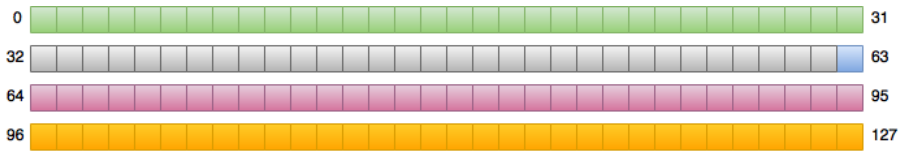


Figure 3.2: **128-bit Overlap Data**. Green: reference id. Grey: N/A. Blue: strand. Pink: Diagonal. Yellow: target sequence index.

Finally, the overlap set is sorted using the 128-bit radix sort. To avoid the diagonal becoming negative, if for example the query index is greater than the target index, a bias must be added to each diagonal when they are appended to the overlap set to avoid a faulty array when sorting. This is because the radix sort is sorting on digits and will therefore consider negate integers as large numbers.

Mapping

The mapping implementation is done more or less exactly as described in Li's article. For each array of overlaps, a loop locates the co-linear intervals, called chains, by finding the longest sequence of overlaps that have diagonals within a certain width, parameterised by r, the cluster width (synonymous to ϵ in the section 2.2.7). Another parameter C ensures that the the chain is of a certain length.

Chains that are long enough are sorted on the query index, by adding or subtracting the target index from the diagonal, depending on the direction.

The chain is then sorted using *qsort*. This is based on the assumption that the intervals rarely will have any significant length, and radix sort will likely be superfluous.

Next,the program proceeds by applying the *Longest Increasing Subsequence* (LIS) algorithm, as used in Li's implementation, and translated from a O(n log k) algorithm found online [88].

This LIS subset is split into new subsets on gaps in the query indices, over a certain length.

Finally, any chains that still satisfy all requirements are reported as a hit, using a modified format based on PAF, as the described in section 2.2.7. The difference is that I have omitted the mapping bases, number of bases and mapping quality. The reasons these were removed is mostly because the way these are calculated are poorly documented, and any effort I made to reproduce these measures based

on Li's own code alone, were not successful. Thus, I decided that including them would merely be based on guessing from my side.

3.3 Parallel Minimap

In this section I present a parallel GPU implementation of the minimap algorithm. The goal was to identify some of the bottlenecks in the serial implementation and attempt to improve these sections by running them on a GPU.

Seeing as the topic of this thesis is long read overlapping, the algorithm was designed primarily for this cause. However, given some small alterations reference-to-reference overlapping is achievable as well.

3.3.1 Implementation

The first step when making a parallel implementation of a program is to do some profiling on the serial version. Some varied testing involving small and large data sets showed that the sketching step, for the most part, was the most expensive step when considering the number of times it was invoked. Also, lookups on the hash table was shown to use the most time in total. This is mostly because each query minimizer must matched. However the actual time spent in each invocation is fairly short. Furthermore, radix sort is a relatively costly procedure, but it is to be expected considering the amount of data that is processed. For readers who are interested in reviewing these profiling results can skip to Section 5.4.1.

Based on this analysis, considerable effort was put into parallelising these sections of code, but seeing as data transfers between host and device are costly, my goal was to do as much computation on the GPU as possible.

The outline of the program is as follows:

1. Startup
   (a) Allocate processing and storage memory
   (b) Create Streams
2. Indexing Stage
   (a) Read target data (CPU)
   (b) Copy data from host to device memory (GPU)
   (c) Sketch minimizers (GPU)
   (d) Sort minimizers (GPU)
   (e) Create and insert minimizers into index table (GPU)
   (f) Sort index table (GPU)
3. Overlap Detection Stage

   (a) Read query data (CPU)
   (b) Copy data from host to device memory (GPU)
   (c) Sketch minimizers (GPU)
   (d) Lookup and add overlapping minimizers to overlap array (GPU)
   (e) Sort overlap array (GPU)
   (f) Copy overlap data back to host memory (GPU)
4. Perform Mapping (CPU)
5. End
   (a) Free processing and storage memory
   (b) Destroy streams

Minimizers

One of the first things to take into consideration when parallelising an algorithm with GPUs is to consider any possible hardware limitations, when compared to the serial CPU implementation. The GPU available at my disposal only had 6 GB of memory available, which is relatively little compared to the vast amounts usually available when solving other problems on CPU clusters.

Furthermore, modern GPUs have 32-bit architectures, as opposed to modern CPUs 64-bits, and running any 64-bit operations will result in the operation having to be emulated into seperate 32-bit operations (see Section 2.4.4 for more about this).

Because of this, we I had to make some alterations in terms of the data types, especially when it came to the size of the minimizers. That's why I decided to set the minimizer to 64 bits instead of 128.

This obviously introduces some restrictions. For example, it is now only possible to have k-mer sizes up to 16. Additionally, the remaining 32 bits are split into a different setup as follows: 28 bits for indexing into the sequence; 3 bits for sequence ids; 1 bit for strand. The 28 bits for sequence indexing is necessary as it allows us to index chromosomes up to the size of 2^{28} (268435456), the largest human chromosome is roughly 250Mbp [89]. Three sequence id bits only supports eight different references at once, but it should suffice for the scope of this work. An illustration of the 64-bit minimizer is illustrated in Figure 3.3.

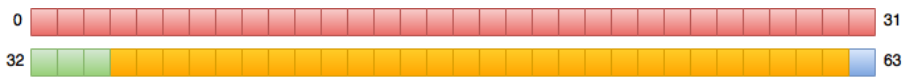


Figure 3.3: **64-bit minimizer**. Red: hash value. Green: reference id. Yellow: sequence index. Blue: strand.

Moreover, 128-bit minimizers double the amount of required memory for our index. ie. having 128-bit minimizers would allow for longer references, and larger

k-mers, but it would not actually be possible able to hold as many references as this data type can support in memory either way.

For example, given that we allocate a certain amount of data $MinimizerData$, to hold the minimizers, this means that it is possible to hold a total of

$$MaxMinimizers = \frac{MinimizerData}{8}$$

, 64-bit minimizers. Furthermore, by modifying the formula from Section 2.2.7, we can calculate approximately the biggest reference genome the memory could hold, ie. the maximum number of nucleotides the minimizer table can hold, with this setup:

$$MaximumSize = \frac{MaxMinimizers * (w+1)}{2}$$

This means that given 2GB of $MinimizerData$, $MaxMinimizers$ equals 2^{28}. Given $w = 5$, this means that the $MaximumSize$ is a little over 805Mbp. If we were to use 128-bit k-mers, this number would be halved.

This is hardly sufficient if we want to store the human genome, which is over 3 GB [89], in addition to having many chromosomes all of which would require more than the eight possible with the 64-bit minimizer, thus forcing us to use 128-bit minimizers, which isn't possible either.

Sketching

Probably the most central part of the minimap algorithm and especially this parallel solution is the sketching. Sketching is done in several stages, all of which will be described in detail in this section.

A string of nucleotides are passed as a byte array of length L. The first step of this to encode these bytes to their corresponding 2-bit values and to locate every minimizer.

Listing 3.1: encode_string

```
1  __global__
2  void encode_string(char * seq, unsigned char * table, minimizer_t *
       res, const int k, const int w, int rid, int SHIFT, int
     BIT_MASK, int length, int offset)
3  {
4    /* global index */
5    uint32_t g_idx = blockDim.x * blockIdx.x + threadIdx.x;
6
7    /* aligned index */
8    uint32_t a_idx = g_idx + offset;
9
10   /* shared index */
11   uint32_t s_idx = threadIdx.x;
12
13     extern __shared__ unsigned char s_seq[];
14
15   if (g_idx >= length) return;
```

```
16
17    s_seq[s_idx] = (uint8_t)table[seq[g_idx]];
18
19    /* Copy overlapping bytes */
20    if (s_idx < k - 1) {
21      s_seq[s_idx + THREADS_PER_BLOCK] = (uint8_t)table[seq[g_idx +
            THREADS_PER_BLOCK]];
22    }
23
24    /* need seperate for last one */
25    if (g_idx >= length - (k - 1)) return;
26
27    __syncthreads();
28
29    int i;
30    int kmer[2] = {0,0};
31    int c;
32    uint32_t z;
33
34    /* Make k-mer. */
35    for (i = 0; i < k; i++) {
36
37      /* Get encoded value */
38      c = s_seq[s_idx+i];
39
40      /* skip any Ambiguous k-mers */
41      if (c > 3) {
42          res[a_idx] = (minimizer_t)-1;
43          return;
44      }
45
46      /* Forward k-mer */
47      kmer[0] = (kmer[0] << 2 | c) & BIT_MASK;
48
49      /* Reverse k-mer */
50      kmer[1] = (kmer[1] >> 2) | (3UL^c) << SHIFT;
51
52
53    }
54
55     /* Symmetric kmers are eliminated */
56    if (kmer[0] == kmer[1]) {
57
58        /* Set to -1 */
59        res[a_idx] = (minimizer_t)-1;
60        return;
61    }
62
63    /* Get strand */
64    z = kmer[0] < kmer[1] ? 0 : 1;
65
66    /* Get hash value */
67    uint32_t h = invertible_hash(kmer[z], BIT_MASK);
68
69    /* Insert to minimizer table */
70    res[a_idx] = (((minimizer_t) h << 32) | (((rid << 29) | (a_idx))
          << 1) | z);
71
72  }
```

For this a kernel called *encode_string* was made where each thread is responsible for creating one k-mer. This means that the number of blocks required is equal

to the input length divided by the block size (1024 is used for all my kernels). The code for this kernel is presented in Listing 3.1

Each thread encodes one byte, using the lookup table, and maps (Section 2.4.9) it into shared memory, as seen on line 17 and 21. Using shared memory, I hope to utilise as much of the memory bandwidth as possible. Afterwards, a barrier ensures that each thread in the block has completed the transaction.

Next, each thread computes its k-mer using a stencil pattern (section 2.4.9), skipping any ambiguous nucleotides or symmetric k-mers by setting the value to -1, and returning (line 42 and 59). The code for calculating forward and complement k-mers was adapted from Li' code [53] (line 47 and 50). Finally, eligible k-mers are hashed with the invertible hash function and inserted into an array consistent with the minimizer data type described in section 3.3.1 (line 67 and 70). When all threads are done, the size of the array consists of a total of $K = L - k - 1$ elements.

There are three separate indices used for this kernel (lines 5, 8, 11). The first *g_idx* points into the global memory, where the sequence is stored as a string. This index is used to fetch a byte and encode it into the shared memory using the *s_idx*. When the results are ready, they are copied into the result array using an aligned index *a_idx* into the correct memory area corresponding to the stream.

One thing to note is that the number of threads in each block is less than the number of bytes of data needed, that's why an additional k - 1 overlapping bytes must be copied by the first warp in each block. This causes some bytes to be copied from global memory twice, by a factor of the number of blocks required in total, multiplied by the overlap. This is necessary due to k-mers inherently overlapping nature.

This solution exposes a good amount of data parallelism overall, where each thread creates a k-mer each in parallel. In regard to the access of byte arrays, this may seem to cause bank conflicts due to several threads accessing the same bank. However, for byte arrays, this is not the case, as the word is broadcast to each requesting thread [66].

Previously, I mentioned that symmetric k-mers or any containing ambiguous nucleotides are set to -1. The reason for doing this is that unlike processing input data linearly such as with a for-loop on CPU, where we can keep appending to the result array as we proceed. We have no way of re-sizing the array during this step, therefore we must filter out the -1 values in a seperate step afterwards. In this case I make use of Thrusts *thrust::Unique* for sketching during indexing and CUBs *cub::DeviceSelect:if* function for sketching during overlapping. By passing a predefined predicate, this essentially makes it filter out any -1 values. The CUB library was chosen over thrust for overlapping through some experimentation and also based on CUBs own documentation stating a significantly better performance on Kepler based architectures [90]. The reason Thrust was used for indexing is because unlike CUB, Thrust does not require an extra array to copy the results in to, something which would require a very large extra array just for these functions, and would lock down a large section of much needed memory.

Next up is possibly the more complicated part of the algorithm, that concerns creating the actual minimizers by finding the smallest of which in each w-sized window. In that sense can this algorithm that I present be considered a parallelisation of the minimizer algorithm. The goal was to create an algorithm that exposes as much data parallelism, with as little step complexity for each thread.

In Section 2.4.9 I discussed the parallel reduce algorithm that takes a binary operator and applies it on a set of N data in a total of O(log N) steps. One solution would be to apply reduce on each w-sized subset a total of $W = k$ - w - *1* times (number of windows to consider). However, this would require a vast amount of threads, in addition to many of which becoming dormant for each step.

Instead, I realised that if one instead uses an offset of $\frac{w}{2}$ this would consequently result in a O(log w) algorithm, and the resulting array would consist of the minimal value in w-sized windows. I call this step *partial reduction*, as we don't reduce on the entire set, as with the regular reduction algorithm. I was not able to find any similar algorithms solutions anywhere I looked, although I doubt this idea hasn't been explored before. To understand this method better, it is easier to visualise it with an illustration as seen in Figure 3.4.

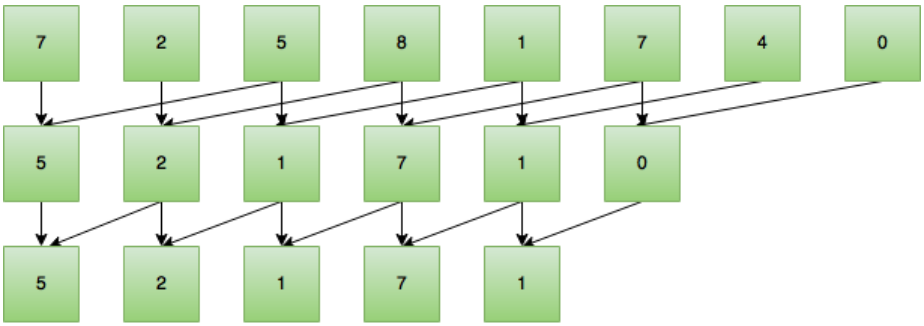


Figure 3.4: **Partial Reduction**. Partial Reduction with $w = 4$, starting with an offset of $\frac{w}{2}$. Eight data elements are reduced to five in two steps.

Now, considering a block size of T threads, it quickly becomes apparent that not only will there be some overlap between the blocks in relation to the data they process, but also the number of minimizers a block produces will be less than the number of threads.

Formally, we can define the number of data required with T threads as

$$D = T + \frac{w}{2}$$

This is the maximum amount of data T threads can process. Given D, the number of results R yielded with a block of T threads is

$$R = T + \frac{w}{2} - w - 1$$

or simply

$$R = D - w - 1$$

Consequently, the required number of blocks B is

$$B = \frac{W}{R} + \begin{cases} 0, & \text{if } W \mod R = 0 \\ 1, & \text{otherwise} \end{cases}$$

The partial reduction is implemented with a kernel I call *reduce_minimizers*. To better understand how this kernel works, I believe it could be helpful to include the code here, in Listing 3.2, proceeded by an explanation.

Listing 3.2: Partial Parallel Reduction

```
1  __global__
2  void reduce_to_minimizers(minimizer_t * minimizers, minimizer_t *
       results, int w, int length, int offset) {
3
4    /* this is the index into the global data, aligned by the offset
         */
5    uint32_t aligned_idx = (blockIdx.x * offset) + threadIdx.x;
6
7    /* block index (index used for shared memory) */
8    uint32_t s_idx = threadIdx.x;
9
10   /* Copy from global to shared */
11   extern __shared__ minimizer_t s_res[];
12   s_res[s_idx] = minimizers[aligned_idx];
13
14   /* Copy extra w/2 memory required. */
15   if (s_idx < (w >> 1)) {
16
17     s_res[s_idx + THREADS_PER_BLOCK] = minimizers[s_idx +
           THREADS_PER_BLOCK];
18   }
19
20   /*
21    * If its the last block we must calculate the needed threads.
22    */
23   int s = (gridDim.x == blockIdx.x + 1) ? ((length % offset) - (w
         >> 1)) : THREADS_PER_BLOCK;
24
25
26   /* Barrier */
27   __syncthreads();
28
29   /* The offset to compare */
30   int r;
31
32   /* window size */
33   int wc = w;
34
35   minimizer_t temp;
36
37   /* Reduce windows of w length */
38   for (r = wc >> 1; r > 0; r >>= 1) {
39
40     /* Return finished threads */
41     if (s_idx >= s) return;
42
43     /* min() function */
44     temp = cmp_min(s_res[s_idx], s_res[s_idx + r]);
45
```

```
46      __syncthreads();
47      s_res[s_idx] = temp;
48      __syncthreads();
49
50      /* readjust for odd numbers */
51      r += wc & 1;
52      wc = r;
53
54      /* Reset thread threshold to be consistent with reduce offset
            */
55      /* Important that this is done here and not before r update */
56      s -= (r >> 1);
57
58    }
59
60    /* Copy into correct offset memory area */
61    results[aligned_idx] = s_res[s_idx];
62  }
```

As with *encode_string* we utilise shared memory here as well. Seeing as we are working with 64-bit data types, I increased the memory bank size to eight bytes from four to improve the bandwidth. Some performance gains were achieved. The overlapping minimizers are copied to shared memory along with the rest, as seen on line 12 and 17. The number of active threads must be recalculated for the last block in coherence with the offset (same as R from earlier). To avoid race conditions on the shared memory, a synchronisation waits for all threads to finish the transfers before continuing.

Next, the actual reduction is done. This loop, as seen on line 38, much resembles that shown in Section 2.4.9, but w is used instead of the length. Because threads are dependant on each other, we must first copy the *min* result into a temporary variable, synchronise, copy result back into shared memory, then synchronise again.

Because window sizes can be odd, we must account for this by readjusting the offset on line 51. Thus, the exact number of steps each thread will take in the loop is technically $Ceil(\log w)$. Later the number of active threads is reduced for each iteration by subtracting the new offset divided by two. Finally the result is copied to results, on line 61.

An alternative to this solution would have one thread loop through w k-mers and report the smallest value over. This is would result in $O(w)$ steps. I believe that despite the synchronisation overhead, that the sub-linear complexity of this reduction method would outperform the linear solution, for a wider range of w. This makes it a more flexible solution if one were to extract only the minimizer sketching part of this implementation, which essentially is a parallel minimizer algorithm. However, for very small w, it is possible that looping can be a little bit faster.

The last step of the sketching algorithm is filtering out unique clusters of minimizers. As previously, and for the same reasons, the Thrust Library was used for indexing, with *thrust::Unique*. And CUBs *cub::DeviceSelect::Unique* was used for overlapping. Both functions do the same thing, and that is to collect unique values, thus removing any repeating minimizers, as illustrated by Figure 3.5.

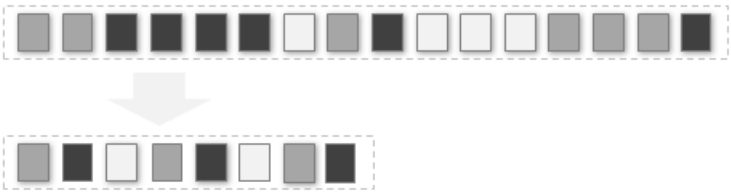


Figure 3.5: **Filtering Unique**. Filtering start of unique sequences. Different shades signify varying sequence contents. [90]

Index Table

The serial implementation of Minimap uses a hash table for its index table. However, implementing hash tables on GPUs is relatively tricky due to the possible race conditions occurring on insertion, causing an overhead on atomic instructions to correctly insert the entries. The positives, being a constant look-up speed, is outweighed by the negatives, being the complexity and time required generating the hash table. There are some implementations of hash tables for GPU possible, such as a *Cuckoo-Hash* based method proposed by Alcantara [91], but ultimately I ended up using a parallel *binary search* method using a sorted array, due to its simplicity and time constraints.

This method reduces the time spent on index table generation, but instead results in a possibly less optimal lookup performance. Binary search on GPUs essentially operates by having each thread perform regular binary search in a given area.

The index table structure used for parallel minimap is that of a sorted array of 64 bits. The first 32 bits being the "look-up" key, and the last 32 bits being the index into the minimizer table.

Indexing

The indexing step is essentially the first part of the algorithm, after we have finished initialising memory.

Due to the memory restrictions on GPUs, the amount of data processed must be divided into sections. The size of these sections can be tuned, but in the parallel minimap, these default to 2^{26} bytes.

First, the entire file is read into host memory using the KSEQ library, as done in the serial solution. Sections are copied from host to device using overlapping transfers using streams, as described in Section 2.4.6. As KSEQ copies data into un-pinned memory, we must pin (and later unpin) these areas using *cudaHostRegister*.

Next the sketching routine is applied, having *encode_string* kernels overlap using streams. It was found that utilising only 1 stream was sufficient for smaller target data (less than 50Mbp), and inputs larger than that are split on four streams. When the entire section has been transferred and encoded, the streams

are synchronised and the proceeding steps of the sketching are done on the default stream, as described in Section 3.3.1.

The resulting sketches for each section is appended to a large array of minimizers. When all the references have been transferred and sketched, the entire array is sorted using the Thrust library's *thrust::sort*, which uses a highly optimised parallel radix sort algorithm.

Followed by the sorting phase, the goal is to filter unique sequences of minimizers based on their hash value (first 32 bit) and insert them into the index table. Because it is not necessary to retain the ordering such as with the previous filtering we did, we can make use of the warp aggregated filter method described in 2.4.9, as this algorithm is shown to have better performance than the filtering methods provided by both Thrust and CUB [86].

I implemented this algorithm into a kernel called *compact_index* more or less with exactly the same code as provided by the author. One thread is deployed per minimizer in a map pattern, and its data entry is added to the index table by the thread if its entry does not equal the previous one, or if it is the very first element of the set.

This unique filter doesn't insert the minimizer as it is, but instead substitutes the most significant word with the index into the minimizer array. Finally, the index table is sorted using the parallel radix sort.

For large data sets, data will be divided into several sections. Along with multiple streams, which in turn consists of multiple blocks. This will introduce a layer of overlaps that we must keep careful track of. This can be illustrated in Figure 3.6.

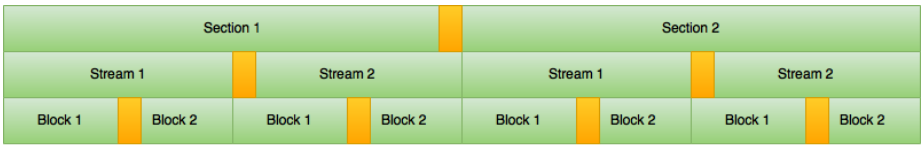


Figure 3.6: **Overlapping Data**. Layered data overlap (yellow) between sections, streams and blocks. A section of n data will produce n - w - k - 2, minimizers.

Overlap Detection

For the overlap stage, all reads are input using KSEQ. Reads are relatively short, even by "long read" standards. So unlike transferring reference data, which in most cases is sufficiently large to merit a seperate heterogeneous memory transfer per data entry, we would like to copy as many reads over to the device memory in one large bulk.

Seeing as KSEQ only processes one read per iteration, we must keep a seperate pinned memory location where we append each read. We do this by looping over each byte in a read and adding it to the pinned memory. The result is one large byte array. So to be able to locate each read, we also make use of two

seperate integer arrays storing the start position and read length/end position respectively.

When this stage is done, the array is copied with overlapping transfers using *NSTREAMS* streams.

Next, the program calculates the number of rounds required by dividing the reads into the number of streams. This parallel implementation of Minimap is mainly designed for overlapping long reads to reference genomes, so I have tried to design the structure of this algorithm so that multiple streams of concurrent kernels may work on seperate reads, all at once.

The level of concurrency, as discussed in Section 2.4.6, depends on the number of threads deployed by each stream, so it is possible that not much overlap between kernel execution is achieved for longer reads, but initial testing during development showed that there indeed was a significant overlap in most cases.

The reason that I wanted to use concurrent kernels for this solution, is that if I were to only process the reads on the default stream, ie. processing one read at the time, this would potentially cause very little utilisation of the GPUs processing power. Because subsequent kernel calls are handled serially, the device would be "locked down" to one kernel call at the time. By using streams, we can try to fit more work onto the GPU at once.

As there are limitations to the amount of working memory available, the processing memory allocated evenly among the streams. Therefore, if one wishes to do genome to genome overlapping, and the size exceeds the amount dedicated to the stream, some alterations must be made to the code. However the current setup has been tested on read lengths up to 70 Kbp.

Apart from the previously mentioned exceptions, is sketching performed using the same core operations as previously described, having each stream processes one read each at the time. This is achieved by a for-loop over the number of rounds. Within this loop, we dispatch each of the sketching and overlap steps by having small NSTREAMS sized loops, to dispatch each of the streams kernels, each of which working on a seperate area. So all the streams run *encode_string*, on their respective reads first. Then all streams run the unique filtering and so on for all the kernels.

Because many the library calls are dependent on values calculated by previous kernel, there are some cases where these values must be copied from device to host. CUB calls require input lengths to be passed by value. Seeing as the values we need to pass are calculated on the device, there is no way to dereference device pointers, meaning that these must be copied over to the host before they can be used as parameters.

Initially this was solved having NSTREAM sized arrays for each stream to copy asynchronously into. This also meant that synchronisation had to be done before accessing the value to avoid race conditions. It turned out however, that these small transactions, including the synchronisation step, slowed down the program significantly. Instead, an non-synchronous copy for all the values on the default stream turned out to be improve the performance considerably. Additionally, as this was on the default stream, this served as a barrier for the rest of the streams, thus removing the need for any explicit synchronisation.

It was also found that increasing the stream number improved performance, because this resulted in fewer and larger transactions. Suggesting that this may be a bottleneck in my implementation.

When the sketches have been created, these are used to lookup the index table using a vectorised version of the Thrust *thrust::lower_bound* function. This is essentially a parallel binary search algorithm that returns the position in the index set in which the query can be inserted first. It is therefore important to ensure that the query and target actually are a match when the minimizer entry is acquired. Because this is a vectorised version, we can pass the entire list of queries (the minimizer sketches) to the function, along with a pre-allocated output buffer, and the index table. The output contains an array of positions that we can use as indices into the index table.

From this point we call a kernel I have called *find_overlaps*, where each thread handles one of the previously fetched indices from the index. The thread computes the position into the minimizer set, using these. Next, the thread verifies that the keys are indeed the same, and then proceeds as seen in the following code snippet 3.3.

Listing 3.3: Snippet from *find_overlaps*

```
1  for (int i = 0; i < MAX_MINIMIZER_OCC; i++) {
2
3      target = M[key + i];
4
5      /* Not a hit, return! */
6      if (target>>32 != query>>32) return;
7
8      /* Same strand ? */
9      if (!((get_strand(query)) ^ get_strand(target))) {
10
11       diag = (get_index(query) - get_index(target)) + BIAS;
12       strand = 0x0;
13
14     } else {
15
16       diag = (get_index(query) + get_index(target)) + BIAS;
17       strand = 0x1;
18     }
19
20     int pos = atomicAggInc(nres);
21
22     uint64_t temp = ((uint64_t)get_rid(target));
23     temp = temp << 1 | ((uint64_t)bit);
24     temp = temp << 28 | ((uint64_t)diag);
25     temp = temp << 28 | ((uint64_t)get_index(target));
26     A[pos] = temp;
27
28 }
```

To reduce thread divergence caused by overlaps on different strands, we try to remedy this by computing the direction and the diagonal up front, before adding them to A (the overlap set), as seen on line 11 and 16. It is important that we also here remember to add the bias to the diagonal to avoid issues during the sorting stage later on.

Similar to the way we generated the index table, we also employ the warp aggregated filtering method when adding elements to the overlap set, on line 20.

An overlap data type is also reduced to 64 bits on line 22. The 64 bit overlap data type has a layout as illustrated in Figure 3.7

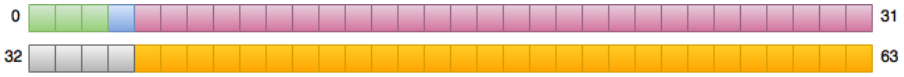


Figure 3.7: **64-bit Overlap Data**. Green: reference id. Blue: strand. Pink: diagonal. Grey: N/A. Yellow: target sequence index.

A limit MAX_MINIMIZER_OCC, can be set to limit the total number of overlaps per query.

The final step of the overlapping is to sort the overlap array using a parallel radix sort. Unlike the indexing stage, the input sizes of the overlapping stage makes it much more efficient to keep temporary data and therefore we are able to use CUBs radix sort, which should perform better for most input lengths based on a comparison of GPU libraries done by Merry [92].

To reduce the number of transfers done in total, and to avoid doing small transfers, ie. one transfer per read, each stream appends overlap results to its own overlap set, and only when the overlapping is complete, or when the remaining amount of memory in the streams overlap array is too little to add the next result, does it transfer all the current data asynchronously to its own seperate area of pinned host memory.

To keep track of the position and length of the results, we have two arrays to store start position and length for each read. To keep track of each individual read, they are given their own index ID. This can be calculated as $index = round\_number * rounds + stream\_id$.

A critical part of making the overlap phase perform well is optimising the use of temporary memory used by each library call (Thrust, CUB etc.). Initially, my program only used Thrust functions, instead of CUB, for each read. To my astonishment, doing so caused a depressing 4x "slow-down" compared to the serial implementation.

To my desperation, I obviously sought to find a better solution, and by doing some profiling using I was quickly made aware of the overhead caused by each Thrust call. Further experimentation showed that by calling a Thrust function, some temporary memory would be allocated, but was not cached for later use, as I had initially expected. As we already are aware of, are memory operations in CUDA are quite expensive, so considering a file containing possibly tens-of-thousands of reads, this quickly accumulates into a lot of unwanted overhead.

CUB on the other hand requires the programmer to allocate this temporary memory manually. By exploiting this, I was able to reduce the number of allocations to one single big allocation, which I divided evenly among each running stream. By doing so, each CUB call to a stream reuses its own section of temporary memory, thus improving the performance greatly.

Mapping

Profiling the serial minimap (section 5.4.1) indicates that the mapping stage takes up a relatively short amount of time, so due to the effectiveness of the serial mapping, was creating a parallel version was not a top priority. Furthermore, the mapping consists of many parts where balancing the amount of work between threads would be hard because of the varying size of the results. Also, because the mapping stage is the last, this means that we will not need to copy data back to the GPU, therefore it is fitting to do the mapping stage on the CPU exactly as it is done in the serial implementation. The only noticeable difference being that I have changed the minimizer size from 128 to 64 bits as discussed in the previous section.

Setup and Memory Requirements

In the current implementation, 32 streams are used, as this gave the best results all-over.

Because of the relatively expensive memory operations, such as allocations and transfers in CUDA, the best results are achieved if one tries to allocate all the memory that is required at one time. Also, because we have limited memory resources, it is vital that we try to make the best use of it, as well as reusing memory whenever we can.

The goal is to utilise as much of the available memory as possible, but only if needed. However, not all cases will require the 6GB of available memory, and doing so will cause unnecessary overhead. The configurations are therefore a trade-off between performance and flexibility.

The general design idea is to have a set block size *DEV_BLOCK_SIZE*, and allocations are multiples of this block size. It is important to note that these sizes are tunable, and this may be required depending on the input file sizes. The current setup is as follows:

Block Size 2^{26} bytes.

Indexing Section size 1 Block. This allows us to transfer and process approximately 67Mbp of DNA data per round.

Processing Memory 8 Blocks are used as working memory for the indexing stage and is also shared by streams during overlapping. It is a multiple of eight as to be able to hold the number of bytes in each indexing section. For overlapping, this means that each stream can hold a maximum of 2097152 minimizers.

Sequence Look-up Table 256 bytes are allocated for the look-up table used for the encoding of nucleotides.

Minimizer Table 16 Blocks are used to hold the minimizer data by default. This approximates to about $4,03x10^8$ bytes of input data, using $w = 5$.

Index Table 8 Blocks are used to hold the index table. The index table is usually a smaller fraction of the minimizer table, but this is simply a

"worst case" precaution.

Host/Device Read Data Up to 1GB of read data can be stored on host and device memory.

Device overlap set 8 Blocks used for overlap arrays stored on the device. Using four streams, this allows for $16,7x10^6$ entries before transferring.

Temporary Data 2MB plus 4B (return values) per stream, as temporary data for CUB library calls.

Binary Search Output 0.5MB per stream for storing the outputs of the binary search. Optimally, it would be great to use the already sufficient temporary data, but somehow thrust will only allow vector data types to be passed to its binary search function.

Host overlap set 4GB is used to store overlap results under default configurations.

Because of this method, memory usage is pretty much constant, apart from that which is based on input size, ie. read data set.

3.3.2 Optimisations

There wasn't much time to spend on optimising each kernel thoroughly, due to the sheer size and complexity of my implementation. However, some basic optimisation strategies, apart from those already mentioned in the implementation section, were used to improve the performance in general.

Where possible, shared memory and registers were used to avoid costly global transfers. I attempted to avoid as much branching code as possible, although this was not always possible. An example of this is in *find_overlaps*, where each thread will take one of two different branches based on the strands of the target and query minimizers.

Global memory transfers are coalesced for all kernels apart from *find_overlap*. The reason being that the the access pattern is based on the indices, which are non-deterministic.

Loop unrolling was applied on loops to achieve higher instruction level parallelism. Unrolling loops essentially tells the compiler to seperate each iteration into seperate sequential steps of machine code instructions instead of having jumps around the same section of instructions.

To achieve higher instructions per cycle, it was always a goal to minimise costly operations such as modulo and division. Where possible, division was substituted for right shifts, and modulo substituted for logical AND [66]. An example of where I have done this can be seen on line 51 in Listing 3.2 in Section 3.3.1. Note that these operations can only be done if the divisor is a power of 2.

The block size for all kernels is set to 1024, as this showed the best overall performance. I suspect however, that smaller blocks may increase occupancy for kernels with frequent synchronisation [66], such as *reduce_minimizers*.

3.3.3 Limitations

The implementation can be considered a prototype due to some limitations I will describe in this section. The reason I refer to it as such, is because although the core of the program works under the given conditions, there are some limitations to its flexibility and usability.

Further, some assumptions were made to be able to "fit" this application on a GPU. Meaning that there are limitations to the maximum number of reference genomes that can be indexed, as well as limitations to the maximum k-mer size and so on. This is driven by the reduced minimizer data type size. This is mostly due to the the hardware restrictions, limiting the amount of memory at our disposal, and arguably the extended performance cost of performing 128 bit operations.

The parallel implementation currently is configured for overlapping reads to genomes. The program is able to support genome-to-genome overlapping as well, but users must actively modify the source code to enable this by reducing the number of threads so that each of the stream's section of memory is large enough to handle the size of the input.

Chapter 4

Methods and Materials

4.1 Software

During the development stage, many different platforms were utilised. In the beginning, a combination of my own desktop personal computer running Windows 10, and one of the machines at UIO running Linux Red Hat 7 were used. As the implementations took form, their requirements were not met by these machines, so all further development moved over to ABEL, running Linux 64 bit Centos 6 operating system [93]. ABEL was accessed by terminal through *Secure Shell* (SSH), and files were handled and edited over *Source File Transfer Program* (SFTP) using *Cyberduck*. This has also been the platform on which I have done my testing and benchmarking of the algorithms.

The serial C implementation was compiled using *gcc* with the -O5 optimisation flag.

The CUDA implementation was compiled with the *nvcc* compiler. Nvcc essentially uses the g++ compiler for host code, and its own compiler for device code. Compilation was done using -O5 optimisation for host code; using -Xptxas -O5 optimisation for GPU; -gencode arch=compute_35,code=sm_35 to signify to the compiler what architecture it should compile to; Also -I. must be added to link the CUB library located in the same folder.

The parallel implementation was tested using GPU driver 367.48 and CUDA toolkit version 8.0.44.

Source code for both implementations are availble at `https://github.com/uio-bmi/GPU-Minimap`.

4.2 Hardware

Testing was done on the following hardware.

CPU Intel Xeon CPU E5-2670 0 @ 2.60GHz.

GPU Nvidia Kepler K20Xm [1]

It should be noted that every time you request a GPU on ABEL, you are likely to get a different one, so it may have minuscule effects on performance.

4.3 Data and Analysis

As references, the *E.coli K-12* bacterial genome and *H.sapiens, Chromosome 1* were used. Having around 4,6Mbp and 250Mbp sizes respectively. This made it possible to use both a small and a relatively large reference for the programs to overlap with.

Minimap is designed primarily as an ONT read overlapper, and due to time restrictions, it should be sufficient to use only ONT read data sets to test my implementation.

For *E.coli* overlapping, Loman et. al's [94] read data set from 2014 was used. Although using the older R7 chemistry, this one is well documented by Quick et. al [33] in their article. And has been used thoroughly by other researchers. Also, the data set is helpful as it contains both template and 2D reads. Which is helpful to compare how this affects my implementation.

Table 4.1: ONT Read Data sets

#	Filename	Chemistry/Type	Reads	Average Read Length	Size (MB)
1	ForwardReads.fasta	R7/1D	58450	6081	360
2	NormalTwoDirectionReads.fasta	R7/2D	20750	6597	139
3	rel3-nanopore-wgs-4178605061-FAB42395.fastq	R9.4/1D	72605	3942	585
4	rel4-nanopore-wgs-4244782843-FAF15630.fastq	R9.4/1D	9663	33400	647

For Human chromosome overlapping Jain et. al's [38] read data set was used. This is a relatively new data set providing data from the newer R9.4 chemistry [38]. The data sets also include ultra long reads, which I will include in my testing. A description of each file included used in my research is described in Table 4.1.

A full list over where to download the test data is provided below.

E.coli str. K-12 substr. MG1655 `https://www.ncbi.nlm.nih.gov/nuccore/U00096.2?report=fasta&to=4639675`

H.sapiens, Chromosome 1 `http://hgdownload.cse.ucsc.edu/goldenPath/hg38/chromosomes/`

E.coli ONT Reads `http://gigadb.org/dataset/100102`

H.sapiens ONT reads `https://github.com/nanopore-wgs-consortium/NA12878`

For accuracy, all timing results were collected as the mean average over 10 separate runs per experiment. For accurate results, *CUDA events* were used to record the CPU time of the implementations. Basically, one records a start and end event on the sections in the code of interest and record the time elapsed

[1] More about specs in section 2.4.4

between the two events up approximately half a microsecond precision [63]. This solution was found to be the most accurate, and to have fair comparison, this scheme was used on both implementations.

For each implementation individually, it is also interesting to do some profiling to see what sections of the algorithms that use the most time, and to also identify possible bottlenecks. For serial minimap, gprof was used. For the parallel implementation I used nvprof to export profiling data from ABEL to the Nvidia visual profiler on a host computer. Because I did not have sudo access on ABEL, there were some limitations to what metrics I could record.

Chapter 5

Results and Discussion

5.1 Pre-processing of Data

To adequately fit my implementations, and to be able to carefully decide the conditions of my experiments, some pre-processing of the data was performed. Generally, the data sets were filtered around a read length interval. For example, the default data sets, 1A and 1B, were both sampled around the mean read length to obtain a new subset, having approximately the same amount of reads, number of bases, and mean read length.

A complete table of the data can be reviewed in table 5.1. For the upcoming experiments discussed in this chapter, this table will be referenced by the experiment codes, ie. "2A" etc.

Table 5.1: Data used for experiments

Experiment	Source File Number	Interval	Reads	Mean Read Length	Bases
1A	2	4000 - 12000	12420	7523	93444033
1B	3	4000 - 12000	18325	6918	126780953
2A	1	4000 - 12000	12420	7358	91391162
3A	4	13000 - 50000	2605	35936	93614649
3B	1	15000 - 70000	5318	18145	96495634

5.2 Validation and Accuracy

Making a one-to-one comparison with Li's original implementation was done with caution, as Li has included some additional parts into his program, that haven't been documented in the article. However, Li's results were used as a guideline to validate the programs accuracy.

The parallel implementation and the serial/Li's implementation was shown to have some differences in number of results at a rate of 0,04%. It is likely that

Table 5.2: Total run time of all experiments

	E.coli 2D Reads (1A)					H.sapiens 1D Reads (1B)					E.coli 1D Reads (2A)				
(k, w)	Serial	SD	Parallel	SD	Speedup	Serial	SD	Parallel	SD	Speedup	Serial	SD	Parallel	SD	Speedup
(15, 5)	9,682	1,355	1,672	0,021	5,791	46,321	2,810	9,408	0,135	4,924	8,422	0,969	1,580	0,075	5,330
(15, 10)	4,925	0,022	1,564	0,043	3,149	27,949	1,193	8,628	0,051	3,236	4,694	0,075	1,580	0,027	2,971
(12, 4)	10,431	0,225	5,114	0,080	2,040	87,51	4,687	9,115	0,110	9,601	9,555	0,091	5,091	0,049	1,877
(12, 8)	6,095	0,15	4,750	0,034	1,283	51,092	3,430	8,908	0,148	5,736	5,523	0,056	4,373	0,033	1,263

	H.sapiens Ultra-long Reads (3A)					E.coli Longer Average Reads (3B)				
(k, w)	Serial	SD	Parallel	SD	Speedup	Serial	SD	Parallel	SD	Speedup
(15, 5)	39,62	1,204	3,705	0,279	10,694	8,152	0,062	0,991	0,110	8,226
(15, 10)	23,335	1,432	2,058	0,079	11,339	4,692	0,010	0,923	0,024	5,083
(12, 4)	73,134	4,109	3,256	0,084	22,461	10,377	0,319	2,505	0,019	4,143
(12, 8)	42,602	3,486	3,053	0,128	13,954	5,54	0,020	2,413	0,031	2,296

this is caused by miscalculation in one of the many offsets during the sketching stages. I did however find, by manually reading through the results of a small sequence about 1Kbp long, that the serial implementations were not able to report some of the minimal hash values in windows correctly. This may suggest that the code which Li uses to calculate minimizers, (and as I have also used for my serial implementation), may act differently from what has been reported in his article. Or it may very well have an error itself. Either way, the error rate is so small, and there is nothing fundamentally wrong with the way any of the programs solve the problem. Although it may cause some minor changes in sensitivity it should not reflect on the overall performance reported in the upcoming sections.

5.3 Runtime Analysis

In this section my goal is to compare and analyse the running time of my two implementations, and to see how, and if the performance alters under different conditions. The total runtime of each experiment, along with the standard deviation over the ten runs can be seen in Table 5.2

Because the initialisation cost of the parallel algorithm is constant, and because time is almost entirely dominated by the allocation of pinned host memory for the overlap results, this initialisation time, which is approximately 1,71 seconds for 4GB, has been omitted from the results. Also because for longer runs, the significance of this is expected to diminish. The mapping stage was not included in the results, as this stage is the same in both implementations. Timing was recorded from the when the indexing procedure is called, and ended when all overlaps are ready to be mapped.

All tests have been run using four different (k, w)-configurations. The default setting for overlapping reads in minimap, per Li's description is (15, 5) [6]. So the results for this configuration should be considered the most significant. For brevity, three more configurations have been added; (15, 10), (12, 4) and (12, 8).

5.3.1 Comparing Long and Short Reference Genomes

My first experiment sought to discover how varying the size of the reference genome would affect the runtime. For this, data set 1A and 1B were overlapped with their respective genomes. The reason that these two sets were compared for this experiment is that the 1A set is relatively old, so although 1B are 1D reads, the difference in error rates should be more or less the same. I will be looking further into these differences in Section 5.3.3.

Figure 5.1a and 5.1b depict bar charts illustrating how the implementations perform when overlapping with E.coli and H.sapiens reads for different (k, w)-configurations. We can observe that the parallel solution achieves a speed-up on both the small and the larger references. The amount of speedup gained varies between each data set and configuration, with the greatest overall speedup achieved on the 1B data set with (k, w) = (12, 4) gaining 9.6x speedup. The lowest on the 1A data set with (k, w) = (12, 8) with 1.3x speedup.

We can observe that the serial implementation tends to follow a similar pattern for equal configurations in both data sets. The GPU solution however, seems to vary heavily between configurations on the 1A set, while having a relatively stable performance for 1B. It is likely that the relatively poor performance on 1A is a consequence of the way my algorithm is designed. I will investigate this further in the next section.

Generally, these initial results seem to suggest that parallel minimap performs better on larger data sets. Which is to be expected.

5.3.2 Comparing Configurations

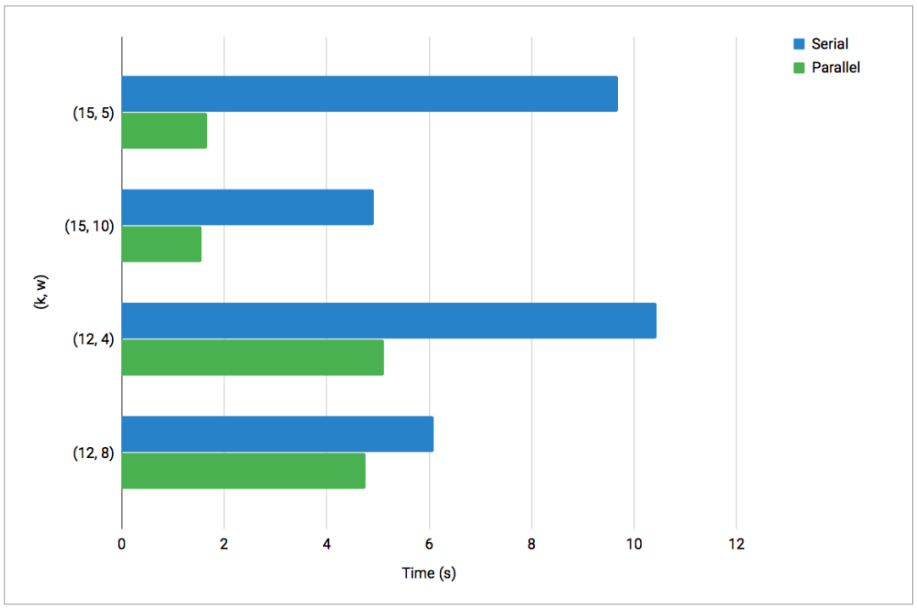
Admittedly, no extensive calculations were done to choose the (k, w)-configurations, and in most literature most authors report using the default parameters [23, 6]. Regardless, I decided that it would be interesting to analyse how these parameters affected the results.

Furthermore, to be able to fully understand the effect of the parameters, we must seperate the indexing and overlap stage. It is interesting to see how indexing and overlap perform individually, because it is likely that the two steps behave differently for seperate reasons.

In Table 5.3 I have compiled the indexing times, speedup, as well as the number of minimizers stored in the minimizer table for both reference sets and for each implementation. Also, a full overview of overlap results are available in Table 5.4, where we can review the results of overlapping each test data set to their corresponding reference genome. The table contains the overlap times for both implementations; speedup; the number of overlaps; and the ratio between the number of overlaps and the number of reads in the data set, ie. average number of overlaps produced per read.

Including the number of output results is important because it can give us a good indication of how much work is done for that run.

(a)



(b)

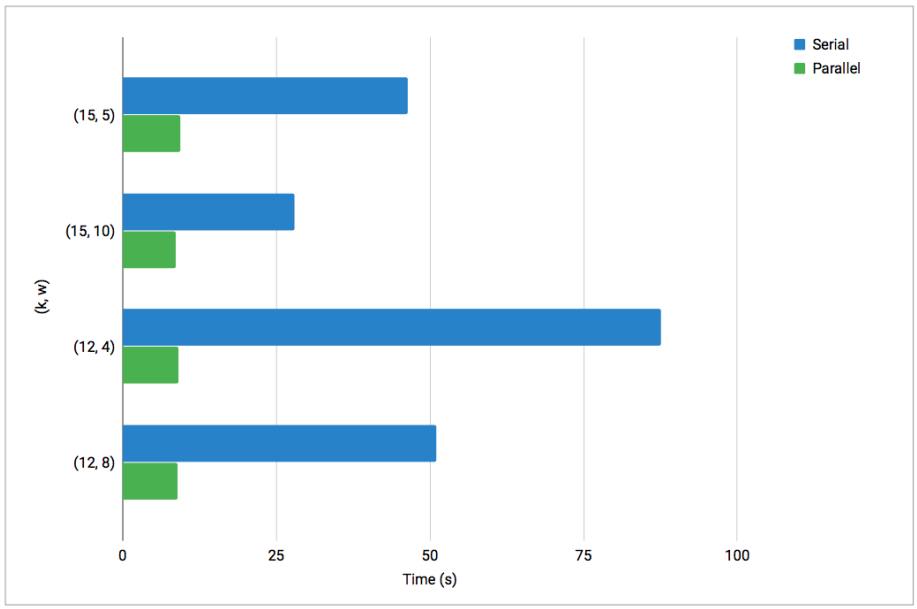


Figure 5.1: **Running Minimap implementations on short and long reference genomes**. Total run time of serial and parallel implentation with varying (k, w)-configuration. (a) 2D E.coli reads (1A). (b) 1D H.sapiens reads (1B)

Table 5.3: Indexing Results

	E.coli				H.sapiens, Chr. 1			
	Time (s)				Time (s)			
(k, w)	Serial	Parallel	Speedup	Minimizers	Serial	Parallel	Speedup	Minimizers
(15, 5)	0,544	0,045	12,090	1580432	19,963	1,665	11,987	78929072
(15, 10)	0,265	0,039	6,743	866844	13,068	1,491	8,763	43435341
(12, 4)	0,372	0,039	9,433	1874410	14,975	1,559	9,603	93707293
(12, 8)	0,240	0,040	5,960	1044445	10,724	1,489	7,204	52349027

The reason that I have separated the tables in this way is to avoid cluttering one large table with duplicate entries caused by the indexing results, as these depend solely on the reference set.

Starting with indexing, we can see that the parallel implementation of this stage performs relatively well for all configurations, showing the best speedup with the default parameters on both reference sets. Overall, we can see that the parallel results differ very little from each other, whereas the serial results tend to be much more dependent on the configurations and number of minimizers. We can see that the number of minimizers is affected by the configurations, and configurations with smaller windows, such as $(k,\ w) = (15,\ 5)$ and $(12,\ 4)$ tend to give the best speedup on both small and large reference data sets, with 12,1x and 12x speedup respectively. These configurations also produce the most minimizers, which is to be expected, as smaller window sizes will reduce the size more than large ones, as described in Section 2.2.7. We can confirm that there is a negative correlation between the window size and the number of resulting minimizers with Figure 5.2.

From this we can deduce that the indexing stage of the parallel minimap implementation performs better on configurations that will produce many minimizers, ie. minimap is able to index many minimizers at approximately the same speed as with fewer ones. This supports the idea of throughput oriented architectures as discussed in Section 2.4.1.

Moving on to the overlap results, we will stick to data sets 1A and 1B for the time being. These are the two first entries in Table 5.4. We see that the portion of time spent on this stage is generally greater than the time spent on the indexing stage. Also, once again it seems that the serial implementation is much more affected by the configurations than the parallel.

We can expect w to have the same relative effect on the number of minimizers per read, as it has on the number of minimizers in the minimizer table. However, in this case the results seem to be more dependant on k. This is because k determines the sensitivity of matching overlaps, as discussed in 2.2.7. And so by decreasing k, we can expect to see more overlaps. Taking for example the human chromosome 1, this set is approximately 250Mbp long, which gives a $\frac{L}{b^k}$ ratio (Section 2.2.7) of exactly 14,84, meaning that the program is not specific enough under this configuration to correctly classify false positives. And as the sensitivity increases, we also get more true positives, all accumulating to more overlaps.

Table 5.4: Overlapping Results of All Read Data Sets

E.coli 2D Reads (1A)					
	Time (s)				
(k, w)	Serial	Parallel	Speedup	Overlaps	Average Overlaps/Read
(15, 5)	9,138	1,627	4,436	2196633	176,863
(15, 10)	4,66	1,525	2,445	1113230	89,632
(12, 4)	10,059	5,074	1,871	25559835	2057,958
(12, 8)	5,855	4,709	1,180	13302319	1071,040
H.sapiens 1D Reads (1B)					
	Time (s)				
(k, w)	Serial	Parallel	Speedup	Overlaps	Average Overlaps/Read
(15, 5)	26,358	7,743	3,404	58188244	3175,348
(15, 10)	14,881	7,137	2,085	30543738	1666,780
(12, 4)	72,535	7,756	9,352	359006068	19591,054
(12, 8)	40,368	7,419	5,441	196092312	10700,808
E.coli 1D Reads (2A)					
	Time (s)				
(k, w)	Serial	Parallel	Speedup	Overlaps	Average Overlaps/Read
(15, 5)	7,878	1,535	5,132	478567	38,532
(15, 10)	4,429	1,509	2,935	242398	19,517
(12, 4)	9,183	5,052	1,818	19486382	1568,952
(12, 8)	5,283	4,333	1,219	10141530	816,548
H.sapiens Ultra-long Reads (3A)					
	Time (s)				
(k, w)	Serial	Parallel	Speedup	Overlaps	Average Overlaps/Read
(15, 5)	19,657	2,039	9,641	40144524	15410,566
(15, 10)	10,267	1,467	6,999	21009425	8065,038
(12, 4)	58,159	1,697	34,271	262541421	100783,655
(12, 8)	31,878	1,564	20,382	143357757	55031,769
E.coli 1D longer average Reads (3B)					
	Time (s)				
(k, w)	Serial	Parallel	Speedup	Overlaps	Average Overlaps/Read
(15, 5)	7,608	0,946	8,042	390075	73,350
(15, 10)	4,427	0,884	5,008	198002	37,232
(12, 4)	10,005	2,465	4,059	19304633	3630,055
(12, 8)	5,300	2,373	2,234	10054171	1890,593

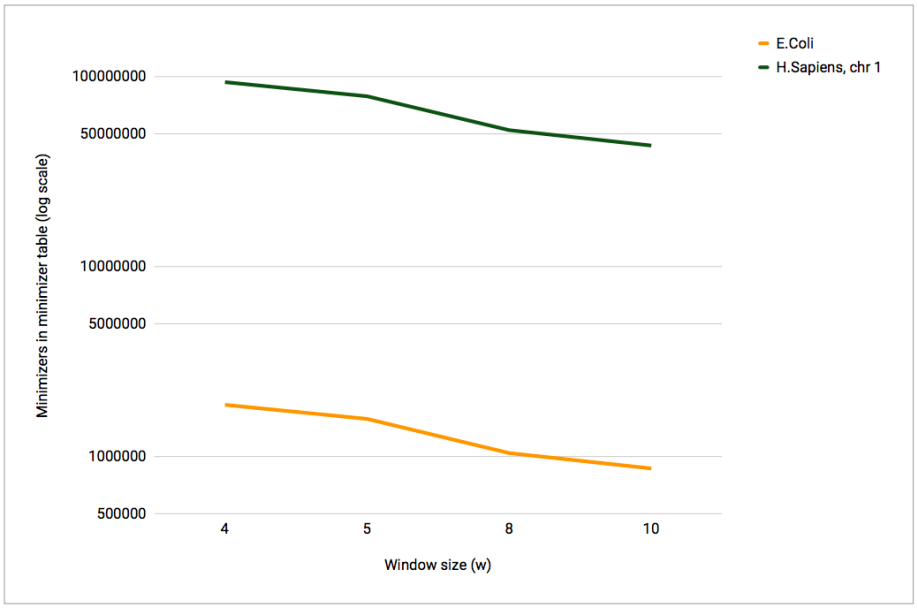


Figure 5.2: **Relation between window size and number of minimizers**. By reducing the window size we can see that the number of minimizers increases.

We can also observe that speedup tends to be better for results with more total overlaps on the H.sapiens reads (1B), but this seems to be the opposite case for E.coli (1A) reads. This is quite interesting, because intuitively, one would think that the parallel implementation would perform better for cases where there are many overlaps to process.

To investigate this further, I divided the number of overlaps by the number of reads in the data set. This can be seen in the sixth column in the table (5.4). This gives us a good indication of how much work each kernel must do. By doing so we can see that there indeed is a relation between a higher $\frac{overlap}{read}$ ratio and a better speedup. However for 1A, the ratio is 177 and 90 overlaps per read for (15,5) and (15, 10), but the speedup is greater than for (12, 4) and (12, 8). I believe this is an anomaly, and my hypothesis was originally that when the ratio becomes significantly small, there will be an abundance of reads where there are no overlaps, thus removing the overhead of dispatching CUBs radix sort function on small sets of data. However, doing some small tests, this did not turn out to be true, although there were cases where one read would only produce as little as 4 overlaps.

So my assessment is that it wasn't the $k = 15$ configurations that were speeding up abnormally, rather that the $k = 12$ configurations were slowing down.

This behaviour continues to be a mystery, but some further speculation as to why this is the case is discussed in section 5.4.2, where I analyse the kernels of the implementation in more depth.

If we observe the 1B experiment singularly, it is easier to see that there is a

positive correlation between between achieved speedup and the average overlap to read ratio. We can see the relationship between the two for 1B in Figure 5.3. From this, I hypothesised that increasing the overlap to read ratio even further, by using longer reads, I might gain an even better speedup.

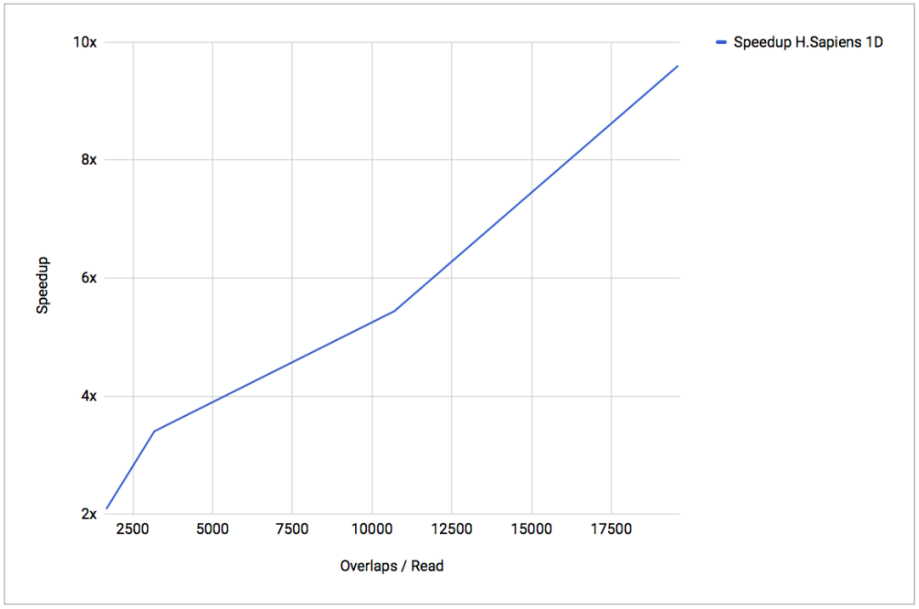


Figure 5.3: **Relation between speedup and overlaps per read**. As the number of overlaps per read increases, the speedup increases too.

5.3.3 Longer Average Read Lengths

To see whether my hypothesis that longer average read lengths improves the parallel overlap performance holds, I tested on the two data sets 3A and 3B. 3A is a subset of the H.sapiens ultra-long reads data set. 3B is a subset of the 1D E.coli reads with a larger interval surrounded around a higher mean read length. The interval used for 3B is quite wide (13000 - 50000). This was to be able to generate enough reads with a higher mean, yet still having the same number of bases in total. This also means that although the mean read length is larger, the set will likely still contain more shorter reads, as the mean is not as concentrated around a higher average, as with the ultra long reads.

The results of this experiment are also presented in table 5.4 (3A, 3B).

For 3B, we see similar results as 2A for runs with very low overlap to read length ratio. However for $(k,\ w) = (12,\ 4)$ and $(12,\ 8)$, we now see that the relative speedup is much better. Even though the number of bytes to process is the same, the parallel implementation performs better when as much of this data is contained in larger reads, as seen by the larger overlap to read length ratio. In Figure 5.4 we observe how changing the average read lengths affect the performance of the algorithms. Also, the serial results remain more similar.

This is because the serial implementation will run through each byte the same way regardless of whether these are separated among shorter or long reads.

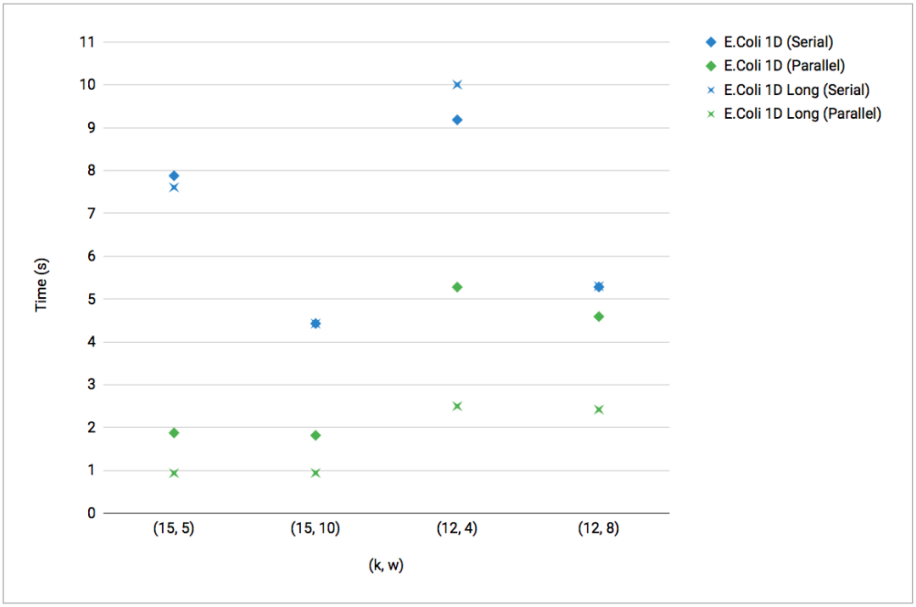


Figure 5.4: **Results from comparing normal and longer 1D reads**. Relative speedup is improved compared to shorter average reads. Serial implementation remains more or less unchanged.

Using 3A, the aim was to see if this effect was greater for ultra-long reads. This experiment provides an even stronger evidence to prove my hypothesis, with ratios ranging up to circa 100784 overlaps per read, resulting in a 34,3x speedup for $(k,\ w) = (12,\ 4)$. Figure 5.5 illustrates these differences clearly. The results show that overall run time of the parallel implementation now has a speedup 3,52x times for ultra long reads when comparing it to its shorter read results, although it should be noted that the number of bases in 3A is 26,2% less than in 1B.

Because the reads are sampled from the entire human genome, and the reference only consists of one chromosome. The reads are only being mapped to 7,7% of their possible targets. It it therefore likely that by indexing more chromosomes, the number of overlaps would increase even further.

1D vs. 2D Reads

It was also interesting to investigate whether 1D (data set 1A) and 2D reads (data set 2A) yield different results. In Figure 5.6 we can see the results of this experiment. The results tend to be more or less the same, with both implementations running slightly faster on the 1D reads compared to the 2D reads. This is because the 2D set contains more accurate reads than the 1D, thus resulting in more overlaps for the program to process. We can confirm this by having a look at the number of overlaps for entries 1A and 2A in Table 5.4.

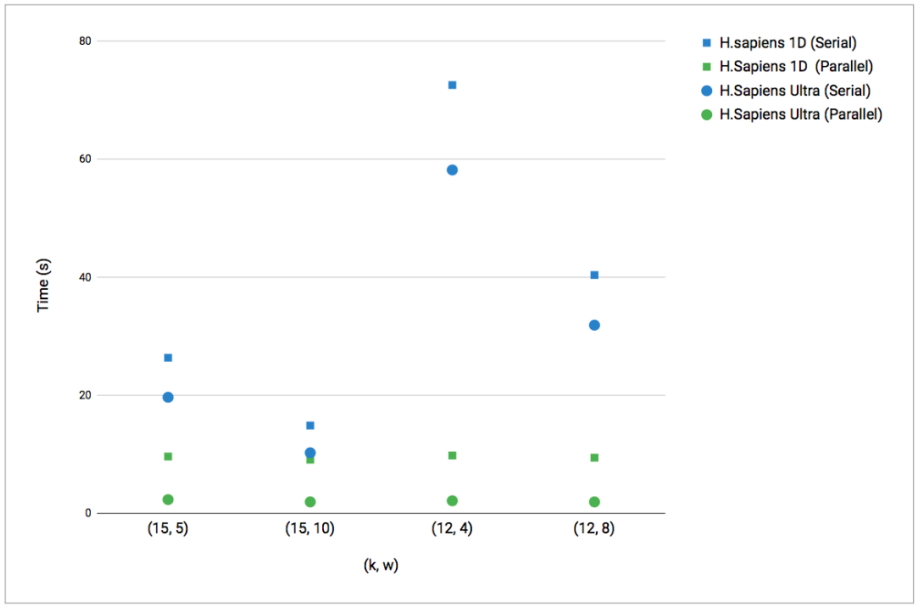


Figure 5.5: **Results from comparing normal and "ultra-long" reads**. Parallel overlap performs better for configurations producing longer reads.

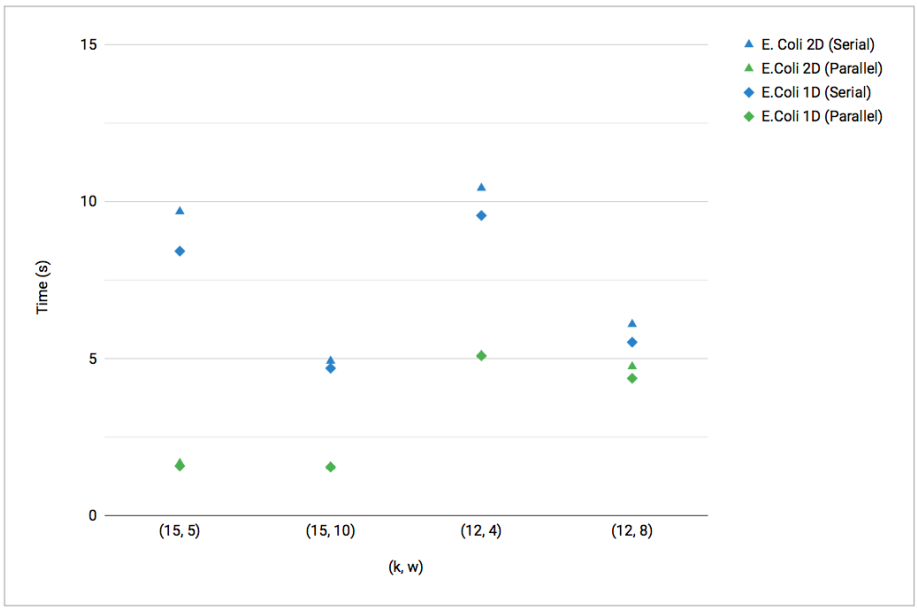


Figure 5.6: **Comparison of 1D and 2D E.coli reads**. Total run time of serial and parallel implementation on 1D and 2D ONT read data sets for all configurations.

5.4 Profiling Results

The aim of this section is to present a quick overview of profiling results on a subset of the experiments using default parameters.

5.4.1 Serial Minimap

Results on profiling the serial implementation suggests that it on average spends most time on lookup, sorting and sketching. The most significant results are presented in Table 5.5 Additionally we can also see that the time spent on mapping overlaps together makes up a very little portion of the running time. Which is the reason why this part was not prioritised for the parallel implementation in the first place.

Table 5.5: Summary of profiling results for serial implementation on data set 1B.

Procedure	% Time	Average Latency
Lookup Hash Table	22.4	0,26μs
Sort Overlap Array	18.57	5,07ms
Sort Minimizer Array	16.83	8.41s
Sketching	14.57	0,40ms
Overlapping	10.27	0,28ms
Insert to Hash Table	7.97	92μs
Map Overlaps	1.10	30μs
Other	8,29	-

5.4.2 Parallel Minimap

Analysing the timeline of the GPU implementation showed that most of the latency is caused by sorting and index lookup, which are also some of the dominant procedures in the serial implementation. Also, for larger data sets (3A), *find_overlaps* was shown to make up a larger portion of the time. A summary of the most interesting profiling results are given in Table 5.6. "Other" represents miscellaneous activity such as device to host copies and subroutines used by Thrust and CUB, such as casting from regular device pointers to Thrust device pointers and so on.

I learned from the profiling that the sorting routines, such as those accomplished using the Thrust and CUB libraries may not be well suited for the relatively small arrays which they sometimes need to process. My understanding is that this causes the overhead of dispatching these functions to sometimes make up a significant portion of the sorting operation.

Further, one can observe that the portion of time spent on my own kernels are relatively small compared to Thrust and CUB calls. The only exception being *find_overlaps*. In general, profiling also suggests that some improvement could be made to the total utilisation of the GPU if we reduce the number of short transactions required before the CUB calls, as this is causing some unwanted latency, due to poor bandwidth utilisation.

CUB calls require variables to be passed by value. Specifically the number of inputs to process in each CUB call, ie. number of elements to sort etc. This means that we cannot pass a device pointer to the function, but instead we

must copy the word to host to be able to use it. This happens two times per read, when filtering out unique minimizers, and when sorting the overlap sets. A consequence of this is that the streams cannot continue to new kernels until the transfer is over, as they are dependant on these values. Meaning that there are some very small sections of time when the GPU may not be doing much work.

If we recall the "mysterious" issue from Section 5.3.2, where abnormal results were produced for $k = 12$ configurations on all E.coli data sets (1A, 2A, 3B). If we compare the results on different configurations in Table 5.6, we see that (12, 4) elicits some rather strange behaviour in the sense that the sort operations are making up a total of 83,94% of the time. This is almost entirely dominated by the sorting of the overlap results, ie. the CUB sort. Exactly what is causing this is yet to be understood. I did do some tests to see whether there were any "bugs" causing this, but all the results are correct. All I could gather was that CUB was invoking a much larger number of subroutines on these experiments compared to others when called on E.coli with these configurations.

Table 5.6: Summary of profiling results for parallel implementation

	Time %			
	1A (15, 5)	1A (12, 4)	3A (15, 5)	3A (12, 4)
Sorting (Thrust + CUB)	50	83,94	40,21	39,19
Lookup (binary search)	26,72	7,69	32	16,16
encode_string	4,18	1,14	2,58	2,61
reduce_minimizers	2,54	0,69	1,30	1,30
find_overlaps	5,86	2,68	14,74	17,2
compact_index	0,01	0,001	0,17	0,22
Filter (Thrust + CUB)	7,83	1,85	3,69	3,37
Other	2,86	2,011	5,31	19,95

Because of some issues with gathering profiling data on ABEL, only some of the performance metrics were obtainable. As a consequence, entries such as computational throughput and global memory bandwidth had to be calculated by hand. This is quite an involved calculation, so values are based on the average over all seperate kernel calls. These values may also differ slightly from those produced by a working profiler, but they should serve as an adequate estimation.

I calculated throughput in GIOP/s (Giga Integer Operations Per Second) using this calculation:

$$IPC * Warp\_Size * SM * Clock\_Rate * 10^{-9}$$

As a reminder, the Kepler architecture supports 32 threads per warp, 14 streaming multiprocessors, and the GPU clock rate is 732 Mhz. By quickly analysing the assembly code, did I learn that multiply-add operations occur very rarely per kernel, so I expect this to have a very little effect on this estimation, so I have not included these in my calculations. Using Nvidia's CUDA C programming guide, they define one instruction to correspond to 32 operations [66]. So

to achieve GInstr/s (Giga Instructions Per Second), it is my understanding that we can divide the GIOP/s by 32, ie. the warp size.

Global memory bandwidth was calculated by counting the number of global loads and stores for each kernel, in bytes. This number was multiplied by the number of working threads in the kernel, and then divided by the latency of the kernel [63].

Table 5.7 gives a summary over the average performance statistics for the non-library kernels used in the parallel implementation. CUB and Thrust have not been included as I was most interested in kernels I could improve myself, and because these calls tend to clutter the profiling log too much as the functions are often split into multiple sub-procedures making it hard to do any concrete analysis on each function.

Generally, we can see that the kernels tend to perform better for indexing than overlapping, with *encode_string* for indexing, achieving the best throughput, having approximately 1148 GIOP/s (Giga Integer Operations Per Second). This is likely because the GPU is finding it easier to fill the SMs with enough work to utilise the throughput better with large data sets, than with smaller, even though I actively tried to counter this when designing my implementation.

To accurately estimate the maximum achievable throughput for one kernel, one has to know the exact amount of instructions of each type. For example does 32-bit integer addition have a maximum throughput of 160 operations per cycle (5 IPC), and 32-bit multiplication only has a throughput of 32 operations per cycle (1 IPC), if we recall from Section 2.4.4. We would need to know the exact instruction mix to be able to determine the maximum throughput. This would usually be possible having a fully functioning profiling tool.

An alternative would be to perform an *object dump* of the assembly code, and count each type of instruction per kernel in the machine code, however this would take a considerable amount of time, and it would require me to estimate how many iterations were performed each loop, per kernel and so on.

Instead, using some rough estimations, we can analyse *encode_string* as an example. The makeup of this kernel is mostly logical (OR, XOR etc.) and shifting operations, caused by the hash function and masking operations. Logical operations have a throughput of 5 IPC, and integer shifting only has 1 IPC. This means we can estimate the maximum IPC somewhere around 4 IPC. This means that *encode_string* for indexing is utilising approximately 90% of its theoretical computational throughput.

Analysing the global memory bandwidth, one can see that none of the kernels are utilising the full global bandwidth (249,6 GB/s), this is because the kernels are performing a very small portion of global loads/stores compared to the amount of work it does. Comparing the throughput-to-bandwidth ratio, it suggests that most of the kernels are compute bound [77], meaning that we have a relatively high arithmetic intensity for most kernels.

Occupancy levels are sufficient for most kernels, apart from *find_overlaps*, having room for some improvement. This means that the application is not capable of hiding some of the latency induced by memory accesses and synchronisations. The high amount of global loads required by this kernel, incurrs a high latency

Table 5.7: Performance Measures for parallel implementation

Kernel	Instructions/ Cycle	Achieved Occupancy	SM Efficiency	(Giga) Integer Operations/s	Global Memory Bandwidth	Latency
reduce_to_minimizers (Indexing)	1,93	92,34%	99,83%	633	94 GB/s	8,28 ms
reduce_to_minimizers (Overlapping)	1,25	78,67%	55,25%	409	14,2 GB/s	10 μs
encode_string (Indexing)	3,50	94,74%	99,49	1148	38 GB/s	4,32 ms
encode_string (Overlapping)	2,66	78,10%	65,53	872	34 GB/s	18,13 μs
compact_index	1,67	82,31%	99,69%	548	153 GB/s	9,63 ms
find_overlaps	0,42	48,80%	58,73%	138	16 GB/s	269 μs

penalty, reported as an average of 269μs per invocation. The reason for which being that there was no ideal way of utilising shared memory for this kernel, in addition to having poor coalesced global memory access. It is possible that by altering the block size for this kernel, it may improve the occupancy, thus hiding more of the latency.

Further, a little over half of the SMs capacity is utilised for most of the kernels in the overlapping stage. This means that the kernel is not utilising the maximum amount available multiprocessors (SMs). One reason for low SM efficiency is likely because of the tail effect described in section 2.4.4. This suggests that some kernels may see a higher device utilisation with smaller thread blocks. I suspect the SM efficiency wouldn't be as problematic if the concurrent kernels were able to run independently, but because several kernels are effectively synchronised, as result of the synchronous device-to-host transfer on the default stream, it is likely that there are moments where the remaining thread blocks are running on a relatively empty GPU. Another reason may be caused by load inbalance. This is because each read often has a different length and complexity, causing some blocks or even some kernels to finish before others.

This suggests that there is not enough work to be done by the kernels when used for overlapping. This is likely related to the low device-to-host bandwidth utilisation caused by the small *cudaMemcpy*'s between some of the kernels.

Because of the relatively low throughput of doing integer operations on GPUs, we cannot expect to get an IPC anywhere near what we could expect if the implementation could utilise floating point arithmetic. However, most kernels show acceptable results, and the most critical bottlenecks are the library functions and small memory transactions.

5.5 Further Discussion

One of the most interesting findings of my analysis is that the serial implementation is very sensitive to changes in configurations, whilst the parallel implementation performs steadily regardless of its configurations. This also implies that the parallel implementation is able to perform as well for configurations aiming for higher sensitivity, ie. lower k and w. This could suggest that the parallel implementation is better suited for for overlapping mammalian genomes (such as H. Sapiens) with higher sensitivity configurations, as this has adverse effects on performance on a serial implementation, as mentioned in section 2.2.7 [53], and confirmed in my results.

The lack of difference in running times for a data set suggests that the performance of the GPU implementation is mostly dependant on the number of reads, and their lengths, and not so much the configurations. There are two reasons why the parallel implementation is running faster with longer and fewer reads. The first is that the GPU generally performs better when it has more work provided, ie. many reads. Secondly, if the same amount of data is concentrated into fewer, but longer reads, we can reduce the number of small transactions slowing up the implementation.

We cannot go any further without addressing the "elephant in the room", which is the fact that my parallel GPU implementation has been compared to a serial, one threaded implementation. So the obvious question to ask would be whether the GPU solution would be any better if compared to a parallel CPU version of Minimap. There is reason to think that a parallel CPU implementation of minimap may quite possibly perform better on "shorter" reads, where the GPU solution achieved relatively little speed-up. For ultra long reads however, the GPU solution achieved an average of 17,83x speedup over the serial implementation.

If we consider a parallel CPU solution using a CPU with 16 cores [1]. Even if 100% of the work was to be parallelised (which is extremely unlikely), it would achieve a 16x speedup compared to the serial implementation if we were to use Amdahl's law as an estimation. By the same logic, would using a 32 core processor still require roughly 97,4% of the work to be parallel, to achieve a similar speedup as the GPU implementation for ultra long reads.

Also, considering that ONT reads are gradually becoming longer and longer [39], we can expect to see a greater demand for technology able to overlap long reads quickly and effectively. Meaning that the GPU solutions like the one presented in this thesis may have a better use as this development continues.

Another thing we should take into consideration is that the GPU that was used to test this implementation is not of the newest standard. I believe we would see a significant speedup relatively better than for the CPU implementation, by running the same experiments using a state of the art GPU and CPU. This is mainly because the increase in computational throughput and memory bandwidth of GPUs has been significantly larger than that of CPUs over the past five years since the K20Xm came out [66].

For example, comparing the K20Xm card to a state of the art *GeForce GTX 1080 Ti* GPU marketed at the gaming industry, this card supposedly has a 260% performance increase over the K20Xm [73]. Even though all this performance may not be transferable to this application, I believe we could still expect a significant relative speedup. With even more powerful GPUs being released every year, it is not unlikely that we can expect to see an even greater improvement as this technology continues to evolve.

[1] Ignoring hyper-threading for this example

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The goal for this thesis was to implement a parallel GPU algorithm for aligning long noisy reads, and to assess whether or not there was any significant performance to be gained from doing so.

Programming graphics processing units is a challenging and time-consuming endeavour. Over the course of my work, I have implemented a parallel and a serial version of an existing long read overlapping algorithm called Minimap [6]. The implementation also incorporates a GPU implementation of the minimizer algorithm [52].

Testing and comparisons of the implementations were performed by doing read-to-genome overlapping on a selection of long read data sets gathered from both bacterial and human DNA, using the Oxford Nanopore MinION sequencing device.

The parallel GPU implementation was shown be faster than the serial CPU version for all data sets and configurations. The degree of speedup achieved was found to be positively correlated with the average number of overlaps produced, and consequently the average number of reads. Speedups were shown to range from from 1,3x using shorter E.coli reads, and up to 22.5x speedup using ultra-long H.sapiens reads. The performance of the parallel implementation was also shown to be less affected by configurations aimed at higher sensitivity than the serial implementation.

The main limitations in using GPUs for long reads was found to be indexing many and/or large genomes due to the limited amount of memory resources available on GPUs. For applications aiming at larger genomes, methods must therefore be investigated to overcome the memory limitations of current GPUs. The most challenging aspects of implementing a GPU overlapper was creating a

flexible application capable of fitting both large and small reads into its pipeline without causing unnecessary overhead and load balancing issues.

I first conclude that the speedup gained by overlapping shorter reads may not necessarily merit the amount of work and resources put into implementing a parallel solution.

However, GPUs are well equipped for researchers and other practitioners aiming at overlapping longer or ultra-long reads to reference genomes. It is also well suited for users who want to overlap long reads with sensitive configurations, without the performance penalty that usually comes with it on a CPU implementation.

With the Nvidia Tesla V100 accelerator coming out later this year capable of sporting 900 GB/s of memory bandwidth and 15 TFLOP/s of single precision throughput [95], and regular improvements to sequencing chemistries making it possible to sequence longer and longer reads, there is reason to believe that GPUs will serve as a viable option for solving sequence alignment problems in the future.

6.2 Future Work

Handling the limitations discussed in section 3.3.3 and making this prototype into a full fledged overlapper should be considered future work.

Moreover, I will discuss some potential improvements in the following sections.

6.2.1 General Improvements

In section 3.3.3, I discussed some of the limitations of the GPU implementation caused by the lack available memory. Newer devices can support up to 16GB of device memory [95], and will allow for even larger genomes to be indexed. However, indexing the entire human genome would still require at least 9GB using $w = 5$ and 64-bit minimizers, for the minimizer table alone. That is why methods should be investigated for overcoming this obstacle,in addition to improving memory utilisation and management in general. This also includes efficient methods that make it possible to index more than eight references.

A possible solution to the memory issues could be to only store the index and as much of the minimizer table as possible in device memory, and store rest in pinned host memory. Minimizers could be copied over to device memory in blocks when queried, using some caching scheme. By ordering the queries, we can attempt to achieve some locality in the hits and reduce the number of transfers. This is because the index table is essentially a filtered version of the minimizer table. By doing so, it would sacrifice some performance, but it could be a solution to the memory limitations.

Modifications should be made to enable the use of more than one device. This would probably alleviate some of the memory requirement issues currently im-

posed on the current implementation. So incorporating this should definitively be considered in future improvements.

The mapping stage of the serial algorithm was shown to make out a relatively small portion of run time. However, investigating methods for utilising the GPU for this step may be interesting in cases where data sets are very long. Minimap mapping essentially consists of separating clusters of co-linear hits, based on a sorted array of overlaps. This could possibly be achieved through a few layers of stream compaction, ie. by filtering overlap data in the the order in which they are sorted (reference id, strand) then perform some operations on these subsets, though as I mentioned in Section 3.3.1, will it be hard to balance the amount of work done by each thread due to unevenly sized results.

The GPU implementation presented in this thesis was designed for read-to-genome overlapping. I would expect the program to perform well also for genome-to-genome overlapping, as the results suggested that the program ran faster with fewer and longer reads. The program should be altered so that it is flexible enough to perform both genome-to-genome and read-to-genome overlapping without actively altering the source code. The possibility of read-to-read overlapping has not been explored in thesis, but I believe the core of the algorithm, which is the sketching procedure, could be utilised for this purpose as well, given a new and more appropriate surrounding structure.

I found that there were some minor discrepancies between the number of results produced by the serial and parallel implementation. Diagnosing the exact reason for this should be a priority to be able to utilise the its full precision as well as using this application as a fully fledged read overlapper.

Porting the algorithm to newer device architectures and testing would be helpful to assess the performance and flexibility of the algorithm more thoroughly.

More thorough analysis of the performance of the GPU solution is needed to optimise the implementation further. Some examples of things that should be analysed includes utilisation of shared memory and caches and the degree of branch divergence.

It would be interesting to assess the performance of the algorithm on PB reads, and possibly synthetic ONT and PB reads too.

Some improvements to the implementation in terms of performance are also to be considered future work, and are discussed in the next section.

6.2.2 GPU Performance Improvements

During testing, we learned that the overlapping stage has room for some improvement. This was mostly due to inefficient transfer between the GPU and CPU. One possible remedy for this would be to expand the blocks from a 1-D grid, to a 2-D grid, so that one thread operates on N different reads, instead of one. I expect this will introduce more instruction level parallelism to the kernels, as well as reduce the number of small transfers required between each kernel call.

In hindsight, I think the most promising improvement, that I unfortunately did not think of before later in the development process, would be to use exploit CUDAs *Dynamic Parallelism*. Dynamic Parallelism allows threads to make new work for itself [68]. This means that we could for example delegate a "leader thread" responsible of dispatching the next kernels.

One of the reasons that this was not done in the first place is that my initial design had Thrust functions instead of CUB for the overlapping. Thrust is purely a host side API, so the design had to be centred around calling kernels from the host. CUB however, allows the programmer to write kernels that can call its functions from the device, thus removing much of the need for host-device interaction. Consequently, this would remove most of the overhead involved, as we would not need to transfer between host and device to call the next kernel.

To solve the overhead of doing sorts on possibly small overlap sets, a "quick" solution could be to move all overlap sets into one collective array, and then sort. To be able to do this, one would have to be able to distinguish between overlaps belonging to different reads and one could do this by utilising the four spare bits in each overlap data type by placing the stream number (0-32) in the four least significant bits of the overlap. With four bits, we would have to reduce the stream count from 32 to a maximum of 16. Although using 32 streams tended to have better performance than 16, during testing, this is mostly because by doing so, it reduces the number of short device-to-host transactions, as described in 3.3.1. By implementing the improvements mentioned above, it would likely eliminate the need for this many streams altogether.

It should be noted that by doing so, it will cause all concurrent kernels to compete for the same *atomicAdd*, instead of having one per stream. This may introduce some unforeseen performance penalties, due to the increased degree of collision. Alternatively, I propose implementing a more flexible sorting method, possibly using different sorting algorithms based on the input size. For example, the CUB library also offers "block-wide" radix sort, which apparently is better suited for smaller arrays.

Because it is hard make any general assumptions about the number of reads, or the number of overlaps one read will produce, the GPU implementation must take certain "precautions" to make sure that it can handle both normal and longer reads. I suggest a more robust load balancing scheme be implemented so that reads being processed simultaneously, have a more similar length. A possible solution would be to sort the arrays of start and end position of each read using their lengths as keys, before kernel execution.

A significant part of the overlap execution time is attributed to initialisation of memory and file I/O. Some of this cost could be entirely eliminated by having a seperate CPU thread take care of reading and processing the query data while the GPU is busy indexing. Furthermore, KSEQ, the code used to input sequence files, currently copies each sequence into a pageable memory location. I propose modifying, or possibly writing new code so that reads are appended directly into an area of pinned memory. This would limit the cost of moving data from one memory area to another.

Also, the GPU currently copies all overlaps over to host memory before mapping. This can also be improved by having a CPU thread process finished blocks of

overlaps, as they are copied from the device. This could also reduce the amount of memory allocated for the host overlap set too, as it currently is designed to hold all the finished overlaps.

Although considerable amount of time was put into optimising my code, there is still some work to be done in terms of tuning each kernel to optimal thread block sizes, to better optimise SM efficiency and warp occupancy. And improving the instruction throughput by eliminating unnecessary costly instructions, if possible.

Because the minimizer table and index table remain constant after their creation, it can be worth experimenting with read-only memories to see whether this increases the performance of the kernel.

Bibliography

[1] Y. Goodfellow, I. Bengio and A. Courville. *Deep Learning*. Adaptive Computation and Machine Learning. The MIT Press, 1st edition, 2017.

[2] Sahinidis N. V. Vouzis, P. D. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188, Jan 2011.

[3] Chen Y. Zhang Y. Xu Y. Ye, W. H-BLAST: a fast protein sequence alignment toolkit on heterogeneous computers with GPUs. *Bioinformatics*, 33(8):1130–1138, 2017.

[4] Pacific Biosciences. Smrt science. `http://www.pacb.com/SMRT-science/`, 2015. Accessed: 2017-06-19.

[5] Oxford Nanopore Technologies. `www.nanoporetech.com`, 2016. Accessed: 2016-04-24.

[6] H. Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, page btw152, 2016.

[7] G. Myers. *Efficient Local Alignment Discovery amongst Noisy Long Reads*, pages 52–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[8] K. Berlin, S. Koren, C. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nat Biotech*, 33(6):623–630, Jun 2015. Research.

[9] Sikic M. Wilm A. Fenlon S. N. Chen S. Nagarajan N. Sovic, I. Fast and sensitive mapping of nanopore sequencing reads with GraphMap. *Nat Commun*, 7:11307, 2016.

[10] W Sung. *Algorithms In Bioinformatics: A Practical Introduction*. Chapman Hall/CRC Mathematical and Computational Biology.

[11] Wikipedia. `https://commons.wikimedia.org/wiki/File:DNA_simple2.svg`, 2008. Accessed: 2016-04-20.

[12] F. H. Crick. On protein synthesis. *Symp. Soc. Exp. Biol.*, 12:138–163, 1958.

[13] I. Sovic. *Algorithms For de Novo Genome Assembly From Third Generation Sequencing Data*. PhD thesis, University of Zagreb, Faculty Of Electrical Engineering and Computing, 2016.

[14] F. Sanger, S. Nicklen, and A. R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci. U.S.A.*, 74(12):5463–5467, 1977.

[15] A. M. Maxam and W. Gilbert. A new method for sequencing DNA. *Proc. Natl. Acad. Sci. U.S.A.*, 74(2):560–564, 1977.

[16] C. S. Pareek, R. Smoczynski, and A. Tretyn. Sequencing technologies and genome sequencing. *J. Appl. Genet.*, 52(4):413–435, 2011.

[17] L. Liu, Y. Li, S. Li, N. Hu, Y. He, R. Pong, D. Lin, L. Lu, and M. Law. Comparison of next-generation sequencing systems. *J. Biomed. Biotechnol.*, 2012:251364, 2012.

[18] Wetterstrand K. A. Dna sequencing costs: Data from the nhgri genome sequencing program (gsp). `https://www.genome.gov/sequencingcostsdata/`, 2016. Accessed: 2017-06-11.

[19] Illumina. An introduction to next-generation sequencing technology. `https://www.illumina.com/content/dam/illumina-marketing/documents/products/illumina_sequencing_introduction.pdf`, note = Accessed: 2017-04-12, 2016.

[20] C. Alkan, S. Sajjadian, and E. E. Eichler. Limitations of next-generation genome sequence assembly. *Nat. Methods*, 8(1):61–65, Jan 2011.

[21] Linton L.M. Birren B. Nusbaum C. Zody M.C. Baldwin J. Devon K. Dewar K. Doyle M. FitzHugh W. Lander, E. et al. Initial sequencing and analysis of the human genome. *Nature*, 2001.

[22] M. J. Chaisson and Glenn Tesler. Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory. *BMC Bioinformatics*, 13(1):238, 2012.

[23] J. Chu, H. Mohamadi, R. L. Warren, C. Yang, and I. Birol. Innovations and challenges in detecting long read overlaps: an evaluation of the state-of-the-art. *Bioinformatics*, 33(8):1261, 2017.

[24] Fehr A. Gray J. Luong K. Lyle J. Otto G. Peluso P. Rank D. Baybayan P. Bettman B. Eid, J. et al. Real-time DNA sequencing from single polymerase molecules. *Science*, 323(5910):133–138, Jan 2009.

[25] Pacific Biosciences. Introducing the sequel system: The scalable platform for smrt sequencing. `www.pacb.com/blog/introducing-the-sequel-system-the-scalable-platform-for-smrt-sequencing/`, 2015. Accessed: 2017-06-12.

[26] D. Laehnemann, A. Borkhardt, and A. C. McHardy. Denoising DNA deep sequencing data-high-throughput sequencing errors and their correction. *Brief. Bioinformatics*, 17(1):154–179, Jan 2016.

[27] K. J. Travers, C. S. Chin, D. R. Rank, J. S. Eid, and S. W. Turner. A flexible and efficient template format for circular consensus sequencing and SNP detection. *Nucleic Acids Res.*, 38(15):e159, Aug 2010.

[28] J. A. Frank, Y. Pan, A. Tooming-Klunderud, V. G. Eijsink, A. C. McHardy, A. J. Nederbragt, and P. B. Pope. Improved metagenome assemblies and

taxonomic binning using long-read circular consensus sequence data. *Sci Rep*, 6:25373, May 2016.

[29] NASA Johnson Space Center Gaskill M., International Space Station Program Office. First dna sequencing in space a game changer. `https://www.nasa.gov/mission_pages/station/research/news/dna_sequencing`, 2016. Accessed: 2017-06-19.

[30] Oxford Nanopore Technologies. Scalable, real-time biological analysis technology. `https://nanoporetech.com/products`, 2016. Accessed: 2017-06-19.

[31] M. Jain, H. E. Olsen, B. Paten, and M. Akeson. The Oxford Nanopore MinION: delivery of nanopore sequencing to the genomics community. *Genome Biol.*, 17(1):239, Nov 2016.

[32] Tin M. M. Y. Mikheyev, A. S. A first look at the oxford nanopore minion sequencer. *Molecular Ecology Resources*, 14(6):1097–1102, 2014.

[33] J. Quick, A. R. Quinlan, and N. J. Loman. A reference bacterial genome dataset generated on the MinIONâ„¢ portable single-molecule nanopore sequencer. *Gigascience*, 3:22, 2014.

[34] M. Jain, I. T. Fiddes, K. H. Miga, H. E. Olsen, B. Paten, and M. Akeson. Improved data analysis for the MinION nanopore sequencer. *Nat. Methods*, 12(4):351–356, Apr 2015.

[35] Harrison J. O'Neill P. A. Moore K. Farbos A. Paszkiewicz K. Studholme D. J. Laver, T. Assessing the performance of the Oxford Nanopore Technologies MinION. *Biomol Detect Quantif*, 3:1–8, 2015.

[36] Loose M. Tyson J. R. de Cesare M. Brown B. L. Jain M. Leggett R. M. Eccles D. A. Zalunin V. Urban J. M. Ip, C. L. et al. MinION Analysis and Reference Consortium: Phase 1 data release and analysis. *F1000Res*, 4:1075, 2015.

[37] Quick J. Simpson J. T. Loman, N. J. A complete bacterial genome assembled de novo using only nanopore sequencing data. *Nat. Methods*, 12(8):733–735, 2015.

[38] Koren S. Quick J. Rand A. C. Sasani T. A. Tyson J. R. Beggs A. D. Dilthey A. T. Fiddes I. T. Malla S. Jain, M. et al. Nanopore sequencing and assembly of a human genome with ultra-long reads. *bioRxiv*, 2017.

[39] N. J. Loman. Thar she blows! ultra long read method for nanopore sequencing. `http://lab.loman.net/2017/03/09/ultrareads-for-nanopore/`, 2017.

[40] Au K. F. Rhoads, A. PacBio Sequencing and Its Applications. *Genomics Proteomics Bioinformatics*, 13(5):278–289, Oct 2015.

[41] Giordano F. Ning Z. Lu, H. Oxford Nanopore MinION Sequencing and Genome Assembly. *Genomics Proteomics Bioinformatics*, 14(5):265–279, Oct 2016.

[42] Oxford Nanopore Technologies. Human genome on a minion. https://nanoporetech.com/about-us/news/human-genome-minion, 2016. Accessed: 2017-06-20.

[43] Nederbragt L. developments in NGS. July 2016.

[44] Workman R. E. Fan Y. Eshleman J. R. Timp W. Norris, A. L. Nanopore sequencing detects structural variants in cancer. *Cancer Biol. Ther.*, 17(3):246–253, 2016.

[45] Wikipedia. n-gram. https://en.wikipedia.org/wiki/N-gram.

[46] Pevzner P. A. Tesler G. Compeau, P. E. C. 2011.

[47] Wikipedia. https://en.wikipedia.org/wiki/Suffix_array, 2017. Accessed: 2016-04-20.

[48] H. Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. 2013.

[49] E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.

[50] D. H. Ballard. Readings in computer vision: Issues, problems, principles, and paradigms. chapter Generalizing the Hough Transform to Detect Arbitrary Shapes, pages 714–725. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[51] U.S. National Library of Medicine National Center for Biotechnology Information. Blast: Frequently asked questions. https://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastDocs&DOC_TYPE=FAQ.

[52] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363, 2004.

[53] H. Li. https://github.com/lh3/minimap, 2016. Accessed: 2017-04-24.

[54] Y. Ono, K. Asai, and M. Hamada. PBSIM: PacBio reads simulator–toward accurate genome assembly. *Bioinformatics*, 29(1):119–121, Jan 2013.

[55] Chu J. Warren R. L. Birol I. Yang, C. NanoSim: nanopore sequence read simulator based on statistical characterization. *Gigascience*, Feb 2017.

[56] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972.

[57] Wikipedia. Speedup. https://en.wikipedia.org/wiki/Speedup.

[58] Nvidia Corporation. Cuda c best practices guide. http://www.nvidia.com/docs/IO/116711/sc11-perf-optimization.pdf.

[59] R. E. Bryant and D. R. O'Halloran. *Computer Systems, A Programmers Perspective*. Pearson, 2nd edition, 2014.

[60] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31, 1988.

[61] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics.*, pages 114–117, 1965.

[62] G. E. Moore. Progress in digital integrated electronics. `http://www.eng.auburn.edu/~agrawvd/COURSE/E7770_Spr07/READ/Gordon_Moore_1975_Speech.pdf`, 1975.

[63] M. Harris. How to implement performance metrics in cuda c/c++. `https://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/`.

[64] Ph.D. Dr. M. R. Fernandez. Nodes, sockets, cores and flops, oh, my. `http://en.community.dell.com/techcenter/high-performance-computing/w/wiki/2329`, 2011. Accessed: 2017-05-29.

[65] Håkon Kvale Stensland. Inf5063 - gpu cuda. `http://www.uio.no/studier/emner/matnat/ifi/INF5063/h16/pensumliste/inf5063-nvidia_gpu.pdf`. Accessed: 2017-07-14.

[66] Nvidia Corp. Cuda c programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2017. Accessed: 2017-05-25.

[67] Nvidia Corporation. `https://developer.nvidia.com/cuda-downloads`, 2017. Accessed: 2017-04-23.

[68] NVIDIA Corporation. Whitepaper, nvidia's next generation cuda(tm) compute architecture: Kepler(tm) gk110. `http://www.nvidia.com/content/pdf/kepler/nvidia-kepler-gk110-architecture-whitepaper.pdf`, 2012. Accessed: 2017-05-27.

[69] Microsoft. Description of race conditions and deadlocks. `https://support.microsoft.com/en-us/help/317723/description-of-race-conditions-and-deadlocks`, 2012. Accessed: 2017-05-25.

[70] A. S. Tanenbaum. *Modern Operating Systems*. Pearson International Edition, 3rd edition.

[71] UiO USIT. Gpu/nvidia/cuda. `https://www.uio.no/english/services/it/research/hpc/abel/help/software/gpu-nvidia-cuda.html`, 2013. Accessed: 2017-05-27.

[72] NVIDIA Corporation. Tesla k20x gpu accelerator, board specification. `http://www.nvidia.ru/content/PDF/kepler/Tesla-K20X-BD-06397-001-v07.pdf`, 2013. Accessed: 2017-05-27.

[73] techpowerup.com. Nvidia tesla k20xm. `https://www.techpowerup.com/gpudb/1884/tesla-k20xm`.

[74] Micikevicius P. Performance optimization. `http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf`.

[75] Nvidia Corporation. Tuning cuda applications for kepler. `http://docs.nvidia.com/pdf/Kepler_Tuning_Guide.pdf`, 2017. Accessed: 2017-05-30.

[76] W.W. Hwu and W. Hwu. *GPU Computing Gems*. Number v. 2 in Applications of GPU computing series. Morgan Kaufmann, 2012.

[77] Micikevicius P. Performance optimization. http://www.nvidia.com/docs/IO/116711/sc11-perf-optimization.pdf.

[78] M. Harris. https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/, 2013. Accessed: 2017-03-10.

[79] Luitjens J. Cuda c/c++ streams and concurrency. http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf. Accessed: 2017-05-30.

[80] M. Harris. Is there a maximum number of streams in cuda? https://stackoverflow.com/questions/3565793/is-there-a-maximum-number-of-streams-in-cuda/12326226#12326226, 2012. Accessed: 2017-03-10.

[81] Harris M. How to overlap data transfers in cuda c/c++. https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/, 2012. Accessed: 2017-05-30.

[82] Harris M. How to optimize data transfers in cuda c/c++. https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/, 2012. Accessed: 2017-05-30.

[83] NVIDIA Corporation. Thrust quick start guide. http://docs.nvidia.com/cuda/thrust/index.html#axzz4k5ZFk5WW, 2017. Accessed: 2017-06-15.

[84] Nvidia Corporation. Cub. https://nvlabs.github.io/cub/, 2017. Accessed: 2017-06-15.

[85] M. McCool. Structured patterns. http://www.drdobbs.com/structured-patterns-an-overview/223101515, 2010. Accessed: 2017-06-16.

[86] A. V. Adinet. Cuda pro tip: Optimized filtering with warp-aggregated atomics. https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/, 2014. Accessed: 2017-06-15.

[87] Wikipedia. Embarrassingly parallel. https://en.wikipedia.org/wiki/Embarrassingly_parallel, 2017. Accessed: 2017-04-15.

[88] Algorithmist. Longest increasing subsequence. http://www.algorithmist.com/index.php/Longest_Increasing_Subsequence.cpp, 2014. Accessed: 2017-05-15.

[89] Wikipedia. Human genome. https://en.wikipedia.org/wiki/Human_genome, 2017. Accessed: 2017-06-30.

[90] Nvidia Corporation. cub::deviceselect struct reference. `https://nvlabs.github.io/cub/structcub_1_1_device_select.html`, 2017. Accessed: 2017-05-10.

[91] D. A. F. Alcantara. *Efficient Hash Tables on the GPU*. PhD thesis, University of California, Davis, 2011.

[92] B. Merry. A performance comparison of sort and scan libraries for GPUs. 2016.

[93] UIO USIT. About abel. `http://www.uio.no/english/services/it/research/hpc/abel/more/`, September.

[94] Loman N.J Quick, J. Bacterial whole-genome read data from the oxford nanopore technologies minion™ nanopore sequencer., 2014.

[95] S. Ryan. Nvidia volta unveiled: Gv100 gpu and tesla v100 accelerator announced. `http://www.anandtech.com/show/11367/nvidia-volta-unveiled-gv100-gpu-and-tesla-v100-accelerator-announced`.