Acceleration of Seed Extension for BWA-MEM DNA Alignment Using GPUs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Jonathan Lévy
born in Montmorency, France

Quantum & Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Acceleration of Seed Extension for BWA-MEM DNA Alignment Using GPUs

Abstract

NA aligning is a compute-intensive and time-consuming task required for all further DNA processing. It consists in finding for each DNA string from a sample its location in a reference genome. Usual techniques for short reads (hundreds of bases) involve seed-extension, where a small matching seed is found with quick search through FM-index and then extended on both ends with a custom Smith-Waterman algorithm, giving optimal solution. However this seed-extension takes a tremendous amount of time. This is why we present in this thesis a solution to offload extension on a GPU to be done in a parallel fashion. This is possible thanks to the fact that the DNA reads do not present any kind of relation between each other. We used the Burrows-Wheeler Aligner - Maximal Exact Match (BWA-MEM), a reference program in the field, to which we integrated a GPU-accelerated library for extension, GASAL2. However, BWA-MEM has a left-right dependency on extension starting scores, with the left alignment starting with the seed score, then the right part starting with the previously calculated left score. We designed a solution by starting both extensions with the seed score, we called this method the "seed-only" paradigm. On our test machine featuring 12 hyperthreaded cores and an NVIDIA Tesla K40c, when running with 12 threads, we could observe a raw kernel speed-up of 4.8× ; but if we allow non-blocking calls to let the CPU run the seeding tasks while the GPU computes the extension, we can reach a 16× effective speed-up for the extension. This extension part takes around 27% of the total time but our acceleration introduces a small overhead due to memory preparations and copying, which makes the whole application 1.28× faster, getting close to the theoretical maximum of 1.37×. Additionally, the paradigm shift we operated creates minimal differences in the final main scores on good quality alignments, with a 1.82% difference for our 5.2 million sequences data set. This makes our acceleration with GASAL2 an solid and efficient solution for a single machine.

Dedicated to my family and friends

Contents

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Ir. Zaid Al-Ars for his valuable insights all along the road of this thesis. I would also like to thank Nauman Ahmed, the creator of GASAL2 and GASE on which this work is based, who has been the scientific advisor who always answered my questions and helped me clear the path to make this journey full of learning experiences.

Jonathan Lévy
Delft, The Netherlands
August 16, 2019

1 Introduction

The work in this thesis is about using the latest computational techniques to speed up a bioinformatics application. To understand the context in which this work takes place, an introduction to some biological concepts is presented in this chapter.

1.1 Context

Desoxyribonucleic Acid, commonly known as DNA [1] is the general medium of information common to all known living creatures. It defines the traits that all individuals from a species share, and thus, is the most important source of biological information for the understanding of life in general. This information is replicated when a cell goes under mitosis. It is the phase when a cell replicates its DNA and splits up to create two new cells. This ensures that the whole organism has a coherent DNA across all its cells.

DNA takes the form of two long string of proteins connected to each other to form a helical structure. These proteins are called "nucleotides" or "bases", as they are the base of the DNA code. There are four of them: adenine [A], thymine [T], cytosine [C] and guanine [G], often referred to by their first letter for ease of use. Adenine and thymine are linked together, and cytosine and guanine are also paired together, forming a double-helix, as shown in Figure 1.1. This double-helix string is compacted in chromosomes, with an X-shape. Most animals are diploids, meaning they carry each chromosome twice, one from the mother and one from the father. Each species has its own number of chromosomes: for example, humans have 46 (23 pairs of chromosomes [2]), cats have 38 (19 pairs [3]), and dogs have 78 (39 pairs [4]). The full DNA information of an individual is the genome of this individual.

Inside a species, DNA can present small differences from one individual to another, in regions that are usually defining a variable trait (for example, hair colour). Portions of DNA defining a characteristic are called *genes*, and there can be multiple viable variants of a gene. These variants are called *allels*. Sexual reproduction is a key step to mix DNA from two individuals and create diversity among a species. These variations can be introduced at sexual reproduction. If the mutation gives the bearer an advantage in life over the non-bearers, it has a high probability to get passed to the bearer's descendants, fostering its propagation in the population (for example, having coloured petals for flowers makes them attractive for insects, and since coloured flowers have become more pollinated, their heirs will inherit this trait). On the contrary, if the mutation becomes a drawback, it is unlikely to be passed to the descendants.

Some DNA mutations are proven deadly, and the most well-known in this category is cancer [6]. A cell whose DNA has been damaged or altered can start multiplying in an uncontrollable fashion, creating a pack of cells growing ever bigger, becoming malign and threatening the normal behaviour of an organ. This process, known as carcinoma,

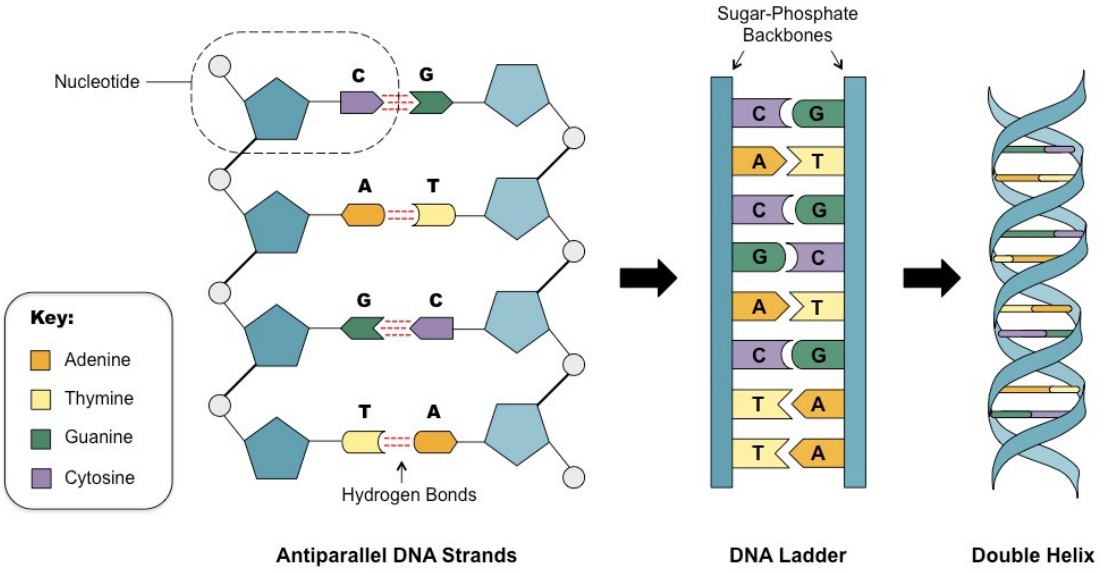


Figure 1.1: DNA double-helix with nucleotides (from [5])

can be detected by multiple ways, some more empirical than others (for example, an easy setup with X-rays can allow a doctor to detect a small abnormal spot, which could be a sign of carcinoma). One way to detect this is by getting a sample of DNA from a patient, and comparing it with a known reference. *Reference genomes* are build from a pool of donors: it is meant to be representative of a species' genes. In the case of the human genome sequences in 2009 "hg19", it has been put together from thirteen American individuals from Buffalo, NY [7]. To achieve this comparison, one needs to find which area of the reference genome matches with the samples, or said differently, one needs to align the sample DNA with the reference and find how similar these two are.

1.2 DNA alignment, a time-consuming process

Mapping is the process of finding the location in the reference genome where a DNA read from a sample probably belongs to. Since 99% of the genome is the same in a species, finding the location in the reference genome is similar to identify the position of a read in the sample DNA. Moreover, the mapping must also provide the information about how well the read is aligned to the reference genome giving the exact location and nature of each kind of mutation (insertion, deletion or modification of one or multiple bases). Mapping DNA reads is a complex task due to large sequencing data and reference genome size. Different types of mutations also increase the complexity of the mapping. In the case of the human, we can write the whole sequence of nucleotides with A, T, C, and G letters, just like a long string of characters. If each character is encoded on a byte, the whole text would be around 3.4GB. Even though it seems small regarding today's standards of data storage, looking for a particular area that would match a string is far

from trivial. As such, a sophisticated way to look into the genome and find an alignment is needed.

Today, DNA mapping represents the first genomics analysis step of many DNA analysis approaches in practice. The complicated process of DNA mapping is rather time consuming, both due to the high complexity of the analysis involved as well as the amount of data that needs to be processed. In many cases, mapping can take between 30% and 50% of the total DNA analysis time. Figure 1.2 shows that the time taken by mapping which is about a third of the total pipeline time. Moreover, the time taken for this part is counted in thousands of CPU-core hours for this example data set. Therefore DNA mapping is a computational challenge and various techniques to achieve it in a timely manner. The problem at stake is then to find an effective way to compute DNA alignment as fast as possible.

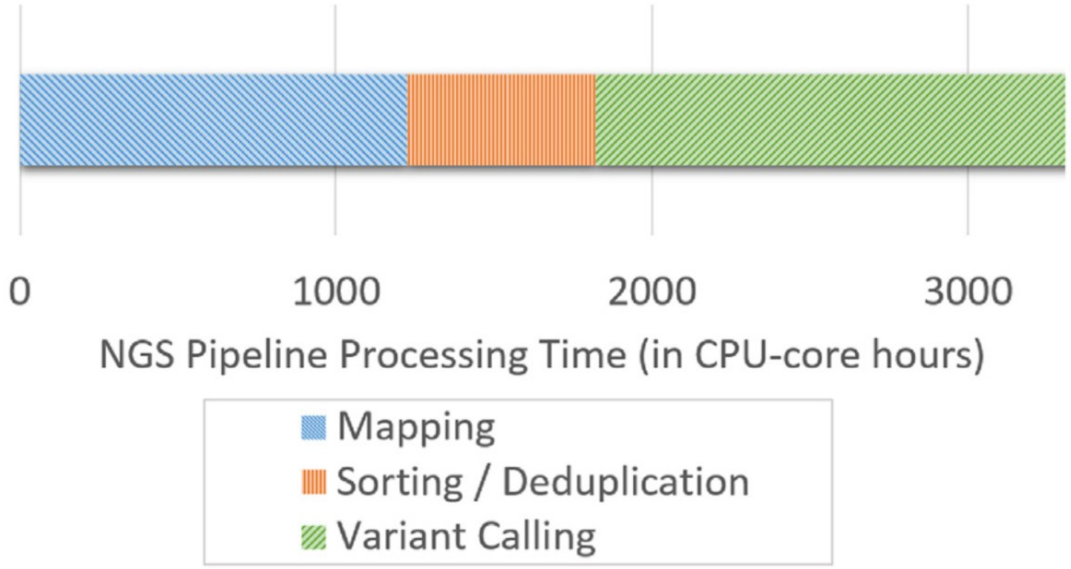


Figure 1.2: DNA pipeline process time share for a typical 30× coverage cancer DNA data set. The data set consists of three tumor samples and one normal tissue sample (time given in CPU-core hours). (from [8])

1.3 The alignment problem

Mapping DNA has become possible with the democratisation of *sequencing machines* or *sequencers*. The first sequencing technique was invented by Sanger in 1977 [9] and the first sequencing machine was released in 1986, making it a rather recent invention. These machines take a DNA sample and find the sequences of A, T, C and G bases from it. The latest ones are called *third generation sequencers*, and are remarkable for their long sequences reads of more than 5000 bases and their ability to compensate for their low raw

read precision with prediction algorithms [10]. Different technologies exist depending on the machine manufacturer. For example, one technique involves using a nanopore as an electrical sensor with a hole through which the DNA string passes, much like a sensor reading a cassette tape. The ionic composition of each base creates a different current change in the sensor, making it possible to deduce which nucleotide it is [11].

In this thesis, we focus on Next Generation Sequencing (NGS) with Illumina sequencing machines. Their technology is called "Sequencing by Synthesis" [12]. All steps are summarised in Figure 1.3. First, DNA is broken into small pieces of 200 to 600 bases pairs called fragments during the sample preparation phase. At the end of each fragment, special molecules called adapters are added to give each fragment an index, and a binding site for the rest of the process. Each fragment is attached by one of its ends to a substrate, the flow cell. It is replicated by bending over, attaching its other end to the substrate, and letting a polymerase create the complement of each base (pairing A with T, T with A, C with G and G with C), then washing away the reverse-complement strand newly created. This is the cluster generation phase. By bending over, creating the complement, then washing it away, the direct strand is replicated millions of times. The direct strand is then sequenced by sending particular DNA nucleotides that have a photosensitive end. After being attached, they are excited by a laser, and the attached base emits a wavelength characteristic of the base (A, T, C or G). The colour of the emitted light is captured by a camera to deduce the base attached. The reverse strand is then created by complementing the direct strand. With another cluster generation phase, millions of copies of the reverse strand are created, and sequenced using the same photosensitive bases. Illumina machines output both the forward and reverse reads, which are called pair-end reads.

This process generates millions of pair-end reads, representing all the fragments. These fragments have to be mapped to a reference genome to deduce information about the sample they come from.

Read sequences and the reference genome are composed of nucleotides or bases, which are reported from their first letter: A, T, C and G. First, we need to introduce the fact that the alignment should provide some kind of metric, to quantify how good the alignment is. Basically, this metric should be a score that increases when there is very few to no differences between the two string of characters, and decreases if they differ, so when letters are not matching. In the case of DNA, mutations often take the form of adding bases, deleting bases, or changing some bases, as seen in Figure 1.4, so the meaning of the score value should reflect these mutations.

Another challenge is to process a huge number of sequences as fast as possible. In fact, DNA sequencers produce strings of a given length proper to each machine. While older sequencers produce short strings of 80 or 150 bases, more recent ones can output several thousand of bases per string, and millions of strings are produced. These strings are not related to each other, hence, there is no obstacle in processing multiple of them at the same time. This mode is called *inter-sequence parallelisation*. It is also possible to parallelise the alignment calculation for each sequence, but this mode, named *intra-sequence parallelisation*, requires extra care about synchronisation.

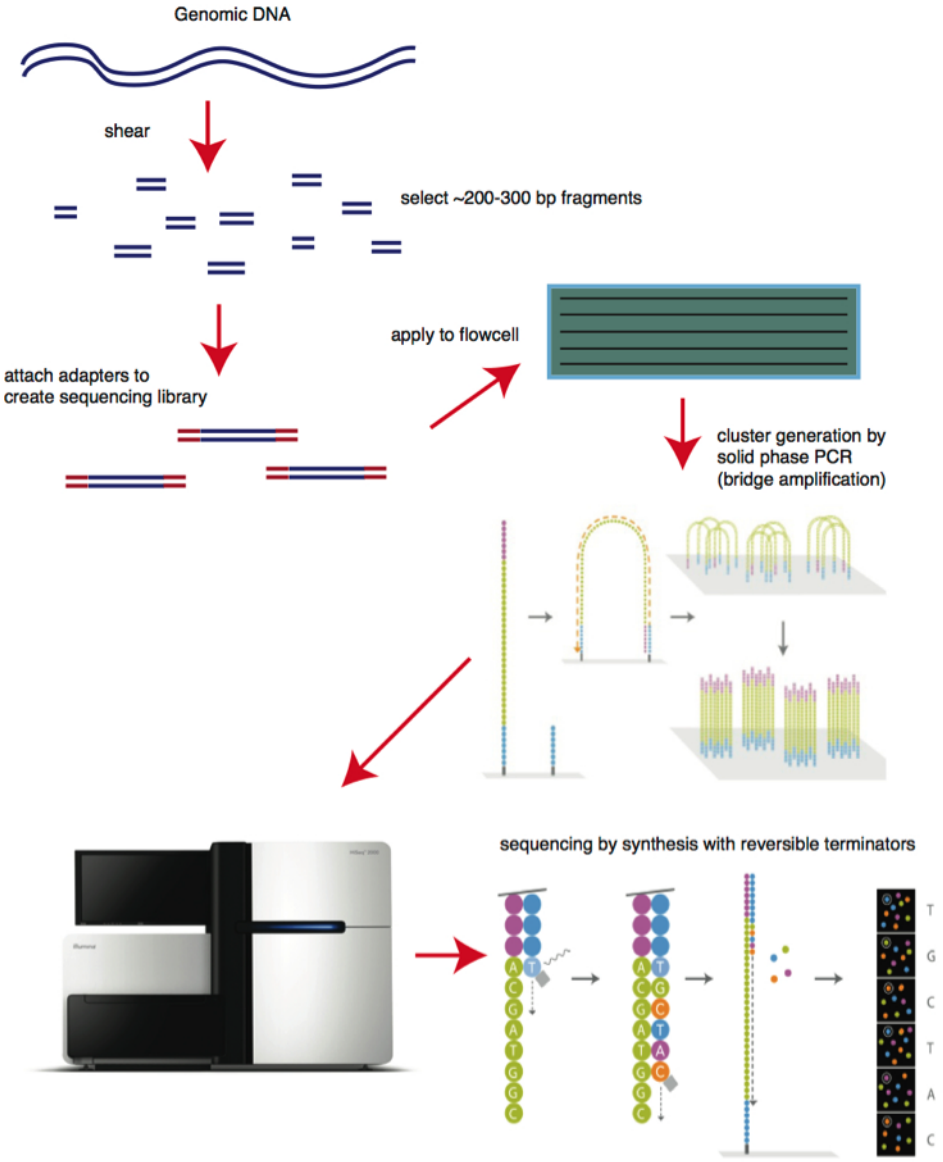


Figure 1.3: Sequencing by Synthesis process for Illumina sequencers. (from [13]

1.4 Research questions

In the sections above, we detailed the alignment problem and how GPUs can help in solving it in a timely manner. The heart of the problem addressed in this thesis is how to accomplish this task. We can formulate this in the following research questions.

- How can we accelerate DNA alignment in an already existing program?
- How much speed-up can we get from GPU acceleration?
- How close can the results with a different computing method be to the original

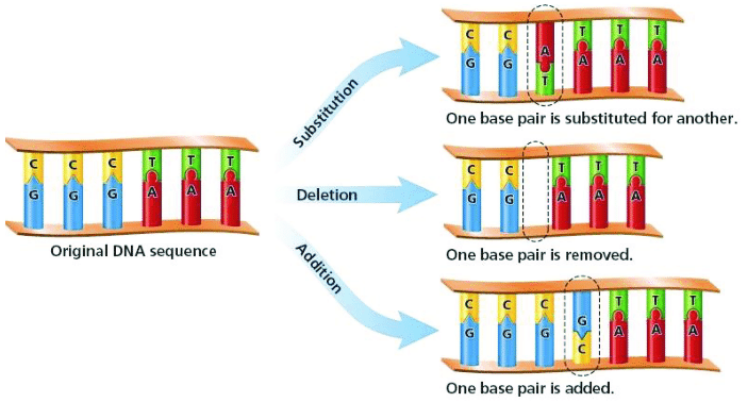


Figure 1.4: Examples of mutation (from [14])

software?

- How to ensure that the GPU resources are well used while leaving more space if needed for future evolution?

1.5 Thesis overview

This thesis is organised as follows. In Chapter 2, an overview of the background to understand DNA alignment is given. We also provide explanations about GPU computing and its usage in this application. This aims to show the reason why we chose this approach to solve the problem of DNA alignment.

Chapter 3 contains details about the accelerator discussed in this thesis. While it originally supported few features, more functionality and flexibility are added and its integration to an existing aligning software is presented.

The implementation of proposed improvements is shown in Chapter 4. In particular, we provide a detailed view of how we adapted the original software to include our accelerator. We also review the modifications we brought to our solution to integrate it successfully.

Chapter 5 presents the experimental setup used to test the implemented accelerator and also discusses the measurement results showing its performance and accuracy.

Finally we will conclude in Chapter 6 with the final thoughts and possible future works.

2 Background

2.1 Seed-and-extend mappers

DNA mapping comes at a time where sequencing machines become more and more available and affordable. The principle behind mapping is to find a match between *reads* coming out of a sequencing machine in a given *reference* genome **misc:mapping**.

To achieve this in a timely manner, most software solutions rely on the *seed-and-extend* technique. In the first step, the *seeding* part consists in finding a substring of a given length that matches exactly in the read and in the reference. Depending on the parameter used in various software products, a maximum of zero, one, or several mismatches can be allowed. For this step, it is necessary to look into the whole genome for a match, and an index of the reference genome is used. The indexing algorithm which is commonly being used in today's DNA mappers is based on the Burrows-Wheeler transform [15]. During this step, multiple seeds can be found in various parts of the reference genome, and depending on how they overlap, they can be grouped in chains.

2.2 The DNA mapper BWA-MEM

BWA-MEM is a popular DNA read mapper used in many real-life applications. Its name stands for *Burrows-Wheeler Aligner - Maximal Exact Match*. In this section, we will provide details of the BWA-MEM algorithms.

2.2.1 Considerations about mapping and alignment

First, we will start with some definitions. *Mapping* is the process of finding correspondence between a data set of DNA fragments called "reads" and a reference genome. To do so, for each read, we first find small areas where the read exactly matches somewhere in the reference genome. This is the *seeding* phase. Then, for all seeds, a *seed-extension* is performed.

After seeding, overlapping seeds or seeds that are close enough in the reference are grouped into *chains*. This allows to fetch a single area in the reference to perform the mapping with the read. The area fetched is larger than the read, since there may be deletions in the read.

When we have a read and a segment of the reference with a corresponding seed, we perform the *seed-extension*, which consists in two alignments on both left and right sides of the seed. If the seed is located at the beginning or the end of the read, only one seed-extension is performed (there is nothing to extend if the seed is on the border). In the extension process, we call *query* the string coming from the read, and *target* the one

from the reference. When both sides of the seed must be aligned, we have then a left query to align with a left target, and a right query to align with the right target.

BWA-MEM operates in three major steps [16]: seeding, chaining, and seed-extension. We will detail these three phases.

2.2.2 Seeding

Seeding consists in finding small areas in the read that exactly match in the reference. This search is done with an FM-index.

The FM-index is a compressed representation for the reference genome. It is widely used due to its lightweight memory footprint, which is sub-linear with respect to the size of the data. Searching for a pattern in the compressed text is also sub-linear in time, which makes it ideal for seeding. When a seed is located, a chunk is taken from the genome around the seed to perform the alignment with the query sequence. This chunk should be larger than the query with which it shares a seed, since there could be gaps in the alignment. More details are available online [17], but the following work does not rely on how FM-index works.

When finding seeds, multiple areas can overlap in the read and the reference, or some seeds can be contained in others. To produce the most useful results, BWA-MEM produces seeds that have two characteristics:

- they cannot be extended any further,
- each seed is not contained in any other seed of the read.

These seeds are called Super-maximal Exact Matches, or *SMEM*s[18].

2.2.3 Chaining

During seeding, multiple seeds can be found in various parts of the reference genome, and depending on how they overlap, they can be grouped in chains. Colinear seeds, or seeds very close to each other are grouped. Very small seeds, or seeds largely contained in longer seeds are filtered out. This heuristics reduces the number of seeds to extend by only keeping the most valuable ones. Seeding and chaining are shown in Figure 2.1. In this example, the dark-blue seed has been found as substring of the red string in the query. But since it is long enough to be considered and close enough to the already existing light-blue seed, these two seeds are grouped in a chain, as shown at the bottom of the Figure. The light blue seed is extended and its extension reaches the dark-blue seed. This is taken into account, and the dark-blue seed is not extended since it is contained in the alignment of the light-blue seed.

When starting extension, seeds are ordered by decreasing score of their chain. Seeds within longest chains are aligned first, since they have a high probability of reaching other seeds of their chains when aligned.

2.2.4 Seed-extension

After seeding, we have to continue to align on both sides of the seed.

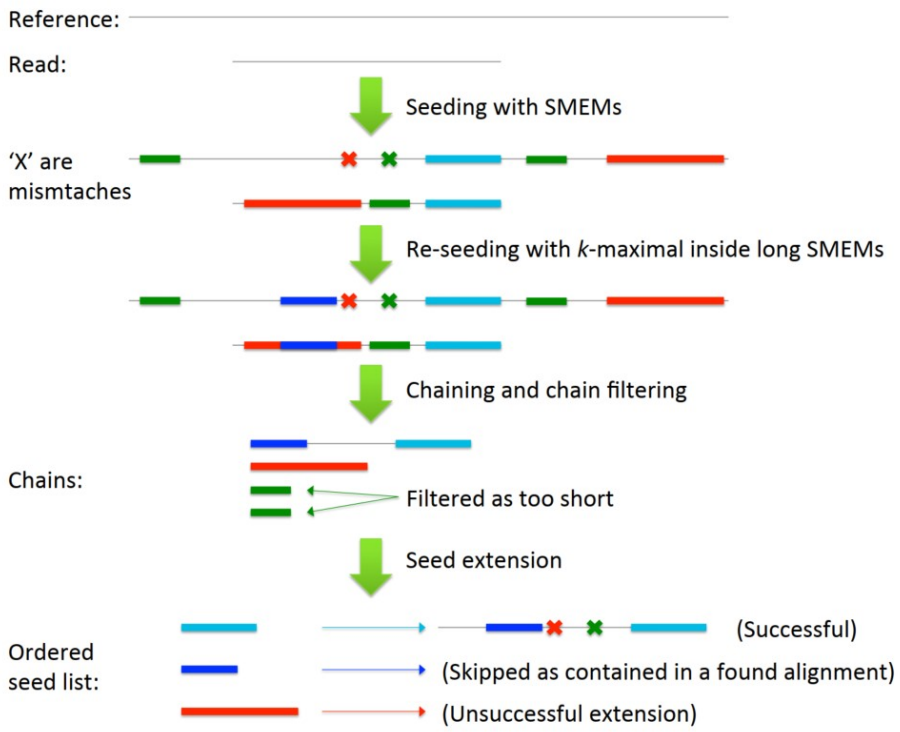


Figure 2.1: Seeding process along with chaining. (from [19])

We define a match score corresponding to the score when two bases are equal. In the case of mismatches, they can appear in the form of insertion or deletion of one or multiple bases, in either the read or the reference. We then define an insertion mismatch score and a deletion mismatch score. The matching score has a positive value, and the mismatch scores are negative, and often both insertion and deletion mismatch scores are set at the same value. For the query sequences, it may happen that a base cannot be correctly retrieved by the sequencing machine, and if this happens, the unknown base is denoted as "N" and is always counted as a mismatch. Furthermore, in real-life mutations, it is much more likely to have a small number of long gaps rather than multiple short gaps. To reflect this, the model "affine gap penalty" for DNA mutation is used. We define a big mismatch score for opening a gap, and a smaller mismatch score for extending the gap. This way, during the alignment, we apply a bigger penalty when trying to open a gap rather than when a mismatch only causes a gap to get wider.

For extension, dynamic programming algorithms are used. They are compute-intensive (and compute-bound), but they provide an optimal solution to the alignment problem. We could actually use them to compute the whole alignment but it is much more efficient to run smaller alignments as left and right extensions of the seed. They rely on computing a matrix of size $N \times M$, N and M being the length of the query and target sequences respectively. As shown in Figure 2.2, we place the target sequence as a row, and the query as a column. Each cell is filled with a score depending on the north

cell, the west cell, and the north-west cell, and the current bases to align. For the pair of bases, it corresponds to row and the column. We assume that we count matches with a score $\alpha > 0$, mismatches with a score $\beta < 0$, and a score for insertion and deletion $\gamma < 0$. For affine gap penalty, we need to differentiate when the alignment opens a gap and when it expands a gap. To this end, we define $g < 0$ as score applied to open a gap, and $h < 0$ as score applied when expanding an existing gap. We have $|g| > |h|$ as opening a gap should be more costly than widening it. Equation 2.2 defines the score to apply when opening or widening a gap in the query, and Equation 2.3 shows the same for target. We define the relation for the score $S[i, j]$ on cell i, j related to bases a_i and b_j with affine gap penalty in Equation 2.1.

In the example in Figure 2.2, we have $\alpha = 2$ and $\beta = \gamma = -1$. More detailed information can be found in [20].

$$S[i,j] = max \begin{cases} S[i-1,j-1] + \alpha & \text{if} \quad a_i = b_j \\ S[i-1,j-1] + \beta & \text{if} \quad a_i \neq b_j \\ G_A[i,j] \\ G_B[i,j] \end{cases} \tag{2.1}$$

$$G_A[i,j] = max \begin{cases} S[i-1,j] + g + h \\ G_A[i-1,j] + h \end{cases} \tag{2.2}$$

$$G_B[i,j] = max \begin{cases} S[i,j-1] + g + h \\ G_B[i,j-1] + h \end{cases} \tag{2.3}$$

S		a	g	g	c	t	g	a
	0	-1	-2	-3	-4	-5	-6	-7
a	-1	2	1	0	-1	-2	-3	-4
g	-2	1	4	3	2	1	0	-1
c	-3	0	3	3	5	4	3	2
t	-4	-1	2	2	4	7	6	5
t	-5	-2	1	1	3	6	6	5
g	-6	-3	0	3	2	5	8	7

(a) Global alignment

S		g	g	g	c	t	g	g	c	g	a
	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0	0	0	2
g	0	2	2	2	1	0	2	2	1	2	1
g	0	2	4	4	3	2	2	4	3	3	2
c	0	1	3	3	6	5	4	3	6	5	4
g	0	2	3	5	5	5	7	6	5	8	7
g	0	2	4	4	4	4	7	6	5	7	7

(b) Local alignment

Figure 2.2: Dynamic programming alignments (from [20])

Among dynamic programming (DP) algorithms, we can cite main techniques:

- Needleman-Wunsch [21], or *global* alignment: it tries to match the whole sequences. The results contains both sequences completely, as Figure 2.2a shows, with the alignment path traversing the matrix. Mismatches on ends cause a penalty, as shows the negative scores on the edges;
- Smith-Waterman [22], or *local* alignment: it aligns two sequences but only looks for the maximal score, which reports the best alignment there is. On Figure 2.2b, one can see we select the best score, the score is initialised at 0 and stays positive;

- A mix of the two, *semi-global* [23] alignment, that performs a global alignment but allows to skip both ends of the query sequence. In practical, semi-global is often used instead of global;

- BLAST [24]-like extension: it performs two alignments on both ends of the chain instead of aligning the sequences entirely. If first makes a local alignment on the left side, starting with a non-zero score, but taking the chain score instead. Then it makes a local alignment on the right side, starting as initial score the previous computed score (taking the chain and left alignment). This splits the alignment problem in two, which makes it much faster. This is the technique used in seed-extension.

The pseudocode for local alignment is presented in Algorithm 1. It computes the whole dynamic programming matrix S with the equation 2.1 showed above. There are small difference between global, local and semi-global alignments in the initialisation step, the computing formula, and how the final score is obtained from the matrix. For the local alignment, which is the one we will use in the rest of the thesis, initialisation is done with a score of zero along both north and west edges of the matrix. This means that no penalty is made on the ends of both sequences. Second, in the computing formula showed in equation 2.1, the score can actually go negative if a lot of mismatches occur. We do not want this behaviour with local alignment, meaning that there is no penalty if an area is not aligned (the alignment can occur later on in the sequence). In Algorithm 1, this is shown by taking the maximum of the computed score with 0. Finally, in local alignment, we want the best possible score without constraints on the end of the alignment, so we simply take the maximum value in the dynamic programming matrix as the end position of the alignment.

Another optimisation called *z-dropoff* is interrupting the calculation of a row when the score drops sharply. This is meant to stop computing as soon as the sequences are visibly not aligned anymore. As shown in Figure 2.3, the goal is to avoid finding a matching area after a non-matching one, since it does not bear any biological meaning and the next aligned region could simply be another seed.

Figure 2.3: Z-dropoff use case: when a badly matching region is found between two matching alignments. (from 2.1)

Finally, it can be noticed that the north-east and south-west corners of the dynamic programming matrix are often useless to compute, since reaching these cells would mean that the alignment is containing a huge gap (and hence has a mediocre score). One can

Algorithm 1 Dynamic programming matrix computation algorithm

procedure COMPUTE THE DYNAMIC PROGRAMMING MATRIX, LOCAL ALIGNMENT(*query_string*, *target_string*, *query_length*, *target_length*) ▷
  Initialise score
  for i from -1 to *query_length* **do**
    $S_{i,-1} \leftarrow 0$
  end for
  for j from -1 to *target_length* **do**
    $S_{-1,j} \leftarrow 0$
  end for
  compute S matrix
  for (i from 0 to $query\_length - 1$ **do**
    for j from 0 to $target\_length - 1$ **do**
      Read base $query\_base_i$ in *query_string*
      Read base $target\_base_j$ in *target_string*
      $$S_{i,j} \leftarrow max \begin{cases} S_{i-1,j-1} + \alpha & \text{if} \quad query\_base_i = target\_base_j \\ S_{i-1,j-1} + \beta & \text{if} \quad query\_base_i \neq target\_base_j \\ G_A[i,j] \\ G_B[i,j] \\ 0 \end{cases}$$
    end for
  end for
  Find i_{max} and j_{max} for which $S_{i_{max},j_{max}} = max(S_{i,j})$
  $score \leftarrow S_{i_{max},j_{max}}$
  $end\_position\_query \leftarrow i_{max}$
  $end\_position\_target \leftarrow j_{max}$
end procedure

simply avoid to compute them, resulting in only calculating a band around the diagonal, hence calling this technique *banded* dynamic programming.

2.2.5 BWA-MEM output

BWA-MEM outputs the alignments in the Sequence Alignment Mapping [25] (SAM) format. It is a text-based format with fields separated by tabulations. The first eleven fields are mandatory and Table 2.1 describes them. In particular, the 5^{th} field "Mapping quality" describes how good the obtained mapping is. In downstream analysis, low-quality mapping are often filtered out using the value of this field. The threshold usually taken for filtering is 20: mapping with a quality strictly lower than 20 are removed.

BWA-MEM outputs in the 6^{th} field the *CIGAR string*, which is a compact way to write the alignment, and describes the way the read maps with the reference. Each letter bears a meaning concerning the alignment. They are presented in Table 2.2. Each number before a letter means how many times the next letter is applied. For example, a CIGAR string of `100M3D12I` means that the first 100 bases of the read are matching,

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

Table 2.1: The mandatory fields for SAM format. (from [25])

then the 3 next bases are deletions (gaps in the target), then the next 12 bases are insertions (gap in the query). In the table, "consumes query" and "consumes target" is a way to define if, when reading the CIGAR string, one should go forward in the target or query DNA string.

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

Table 2.2: The CIGAR string components. (from [25])

2.3 DNA read mapping on GPU

DNA read mappers have to process a huge number of DNA reads as fast as possible. The first way to parallelise the mapping is to use multiple CPU threads. BWA-MEM supports multhreading mode where each CPU thread maps one DNA read. But the number of threads on a CPU is often very limited, as a single CPU has at best a few dozen of cores. Other types of hardware can present more parallelism.

To accelerate the computation, several approaches were tried. For example, Ahmed *et al.* used a dedicated FPGA (*Field Programmable Gate Array*) to accelerate the research with the FM-index and the extension [26]. Another approach from Mashtaq *et al.* consisted in streaming DNA sequences alignment tasks to a cluster with Apache Spark [27]. But these solutions require extra hardware and can be difficult to set up. We would like to propose a solution using common hardware found on computers, in particular, Graphics Processing Units, or GPUs.

DNA computation can take advantage of the inner structure of a GPU to compute the alignment faster. The use of GPUs have been investigated for DNA acceleration, first for global alignment [28], then in the NVBIO CUDA library [29], and later for semi-global and local alignment with on-GPU data compression [30]. GPUs have been proven very efficient for these tasks. First, we will quickly review the current solutions at hand. Then a presentation of the GPU architecture and programming paradigm will show how these devices work and how they are adapted for this workload.

2.3.1 Parallel computing for DNA alignment

In the seed-extension stage, DNA read mapper align a pair of sequences (pairwise alignment) using a sequence alignment algorithm. The computation for alignment between two DNA strings is intensive, yet simple. The main problem comes from the input data size. Each DNA read has many seed hits and each seed have to be extended. Hence, a large number of seed-extensions has to be performed. Still, it is important to note that each pairwise alignment is independent from the others. A simple and effective way to accelerate the computation is to parallelise it. On a computer, one can use multiple threads on a CPU to achieve that parallelisation and compute several alignments at the same time. However, the number of threads available on a CPU is limited to a dozen, or a few dozen at best; but Graphics Computing Units (GPUs) can have hundreds to thousands of cores at disposal. This makes GPUs a hardware of choice for acceleration using parallel computing. Running multiple alignments in parallel is called *inter-sequence parallelisation*.

It is possible to go even further with *intra-sequence parallelisation*, that is, to calculate the dynamic programming matrix with multiple threads. This implies having synchronisation phases: since the cells values depend on the previously calculated ones, it implies processing with a logic order, and requires a high level of optimisation for efficient speed-up [31]. Even though this have many advantages in speed but also in power consumption [32], it involves complex algorithm crafting to get effective benefits, and we will not explore this topic later on.

Currently a lot of solutions exists [33] so we will only name some of the existing software programs for DNA alignment. These tools are most often present as libraries, to be able to integrate them in bigger and more easy-to-use software.

- *SeqAn* [34] is a C++ toolbox for DNA alignment. It features a lot of various tools, and implements its own C++ library for alignment. It runs on multiple CPU threads, and can rely on SIMD instructions for inter- and intra-sequence parallelisation. Both AVX2 S as well AVX512 SIMD instruction sets are supported.

- *Basic Local Alignment Seach Tool* (*BLAST*) [24] searches for pairwise alignment with the seed-extension method,

- *Bowtie2* [35] is a fast and lightweight DNA aligner that runs even on low-end machines. It allows for up to two mismatches in the seeding phase and in general it reaches maximum efficiency by trading exactness for reasonable heuristics.

- *NVBIO* [29] is a GPU-accelerated library written in CUDA, the dedicated language for general-purpose GPU computing on NVIDIA GPUs. It has a lot of features including banded dynamic programming, fast FM-index construction, and efficient data handling between CPU and GPU.
- *GASAL2* [36] is also a CUDA-written library for DNA alignment, developed with speed in mind. Despite its relatively small number of features, it runs all compute-intensive parts on GPU for maximum speed-up.

2.3.2 General-purpose GPU computing basics

Historically, GPUs have been developed to render graphics, and because of this purpose, have a highly parallel architecture to cope with independent parts of the picture. GPUs can now be used for generic computing, as a separate accelerator towards which the CPU can offload parts of the computation. As such, they have large number of cores, orders of magnitude higher than CPUs (several thousands versus a few dozen). Yet, programming on GPU highly depends on the hardware at hand. There is an open programming framework called *OpenCL* [37] and it has the advantage of being cross-platform for all GPUs and royalty-free; yet it is quite complex to write and run. Hence a lot of people are rather using CUDA [38], the proprietary framework to program NVIDIA GPUs. For the rest of the thesis, we will only focus on NVIDIA GPU architecture and CUDA.

We introduce the terms *host* to designate the CPU-side of the machine, constituted of the CPU and its RAM; and *device* the GPU-side with its dedicated onboard memory called *global memory* or *VRAM*, short for Video RAM. Figure 2.4 shows a summarised view of the CPU-GPU tandem. First, the CPU "host" transfers data from the RAM to the device VRAM. Then the CPU launches parallel functions called *kernels* on the GPU. This kernel is launched on a *grid*, divided into *blocks*, each of them having a certain number of *threads*. The grid and block sizes are specified as launch parameters, and they are adapted to divide all the data to process into a suited number of threads. Inside a block, threads can access a shared memory. Scopes from different memories are specified in Figure 2.5. Each thread executes an instance of the kernel, has thread ID and has its own private memory for local variables. Threads in a block access a shared memory, allowing to synchronise them and share data. Blocks are arranged in grids, and grids share data with a memory space allocated for the application context in global memory.

When the computation is done on the GPU, output data is available in VRAM. The CPU can then fetch the results from VRAM to load them in RAM, so that they can be further processed, stored or displayed.

In our case, we use NVIDIA GPUs that present their own particularities in the architecture [40]. NVIDIA GPUs organise computing resources in Streaming Multiprocessors (SM). We will describe the features from the Kepler architecture since it is the hardware we used for this thesis. For example, the Kepler architecture features 15 SMs which share an L2 cache. Each SM has an instruction cache, L1 cache used as shared memory, texture memories, and a set of computing resources. The main computing core are *CUDA cores* or Stream Processors (SP) for single-precision calculation. But there are also double-precision units, special functions units, and load/store units to manage registry operations.

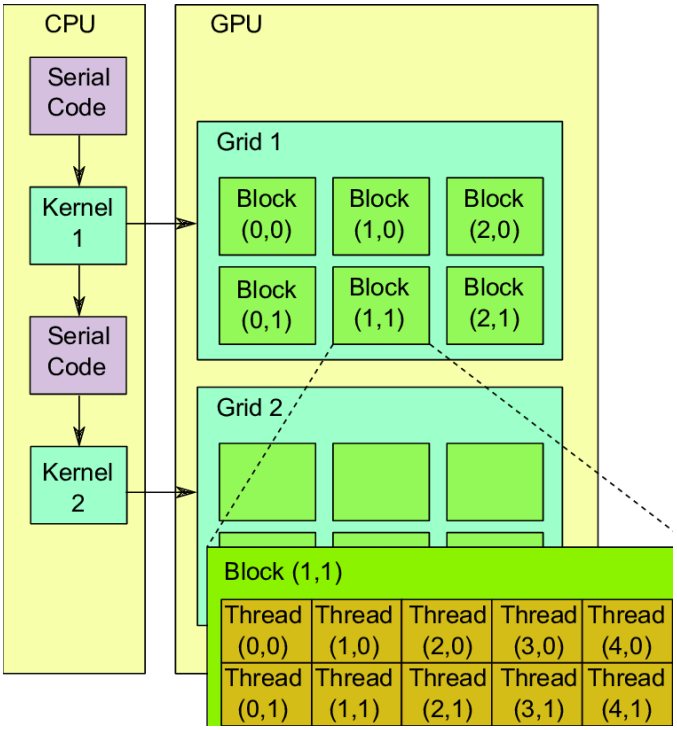


Figure 2.4: GPU architecture schematics (from [39])

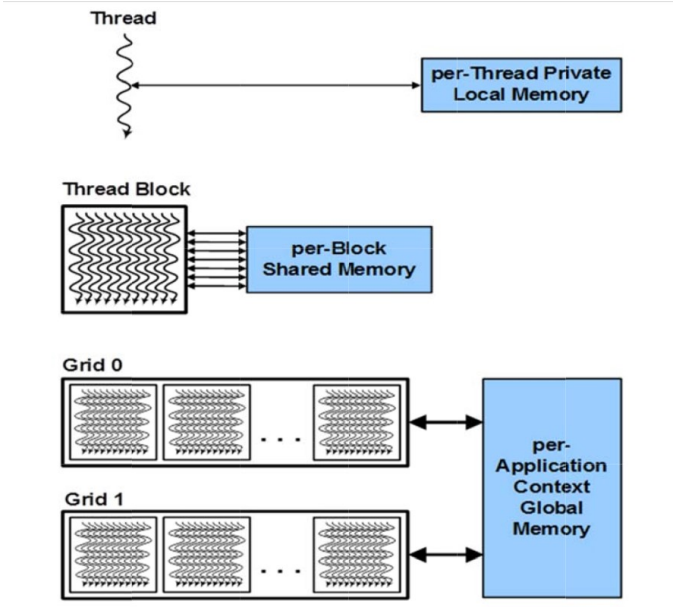


Figure 2.5: GPU memories scope (from [40])

This presents two main benefits. First, some problems that can be highly parallelisable will take advantage of the huge number of GPU cores. Second, being able to offload

the calculation to a separate device means that both the CPU and the GPU are busy at the same time: this enables hidden time computation, that is, the CPU can continue to execute another part of the code while the kernel is running on the GPU runs its part, and the former can retrieve the results only when the latter is done.

Hidden time capability is achieved thanks to non-blocking launches and streams [41]. Streams are execution environments for GPUs. For several years now, it is possible to launch multiple threads of the same function on the GPU, the *kernel*, without needing to wait for all kernel instances to finish. The GPU kernel launch returns immediately, and the host can perform other tasks while the kernel is running on the GPU, and the host simply checks if the stream has finished. For example, a program declares two streams, the host can launch the kernel on Stream #1, then fill up the data for Stream #2 while the first one computes. When Stream #2 is launched, by that time, Stream #1 has finished, so results can be retrieved and Stream #1 can be filled again with the next data to compute while Stream #2 computes; and so on. The principle is explained here with two streams, but one can use as many streams as needed to utilise the GPU resources as much as possible.

It is crucial to monitor some metrics to ensure the GPU is used to its fullest while complying with its hardware limitations. We can mention:

- occupancy (in %): it represents how much of the computing resources are used, so the fraction of SP in use when a kernel is running.
- VRAM use (in MB): it is the onboard memory and cannot be exceeded.
- data transfer time: since it is a separate device, it is important to check if data transfers are taking more or less time than the computation part.

2.3.3 Motivation

We have seen that DNA mapping is a time-consuming task and that various techniques have been tried to accelerate it. In particular, parallelisation is a way to reach high throughput by using adequate hardware. However, the approaches tried present a huge drawback for DNA scientists. When they rely on a DNA mapper like BWA-MEM, they expect to trust the output of the tool they work with. Hence, an accelerated BWA-MEM must provide matching results with the original BWA-MEM.

Many of the approaches used in previous works use pruning to accelerate the calculation of the matrix [42]. As shown in Figure 2.6, pruning is a strategy to avoid computing useless cells in a banded fashion. However it does not preserve results which is unacceptable for end-users. This is why we aim to provide an accelerated implementation of BWA-MEM with trusted output that could actually comply with end-users expectations.

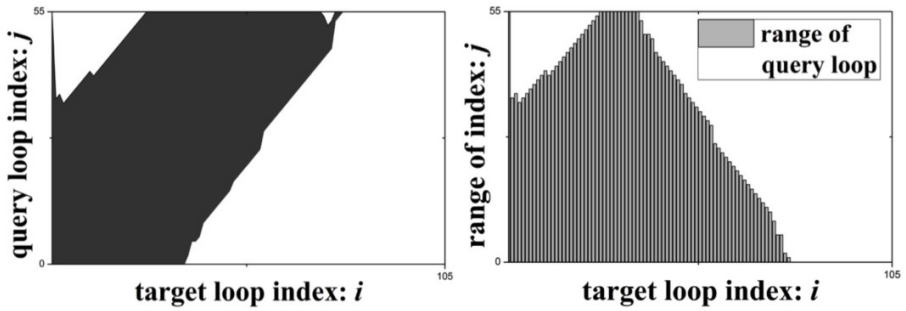


Figure 2.6: An example of BWA-MEM Smith-Waterman task. Left: cells actually computed in the matrix with pruning. Right: amount of elements calculated for each loop index. (from [42])

3 Accelerator design

The following chapter is dedicated to acceleration of the seed-extension, or simply called *extension* part and its integration in BWA-MEM. To do this, we developed a CUDA kernel for extension in the GASAL2 library. Then, we integrated the kernel in BWA-MEM to accelerate it. We will first show that extension acceleration can bring a substantial benefit to BWA-MEM performance. Then we will present the current state of GASAL2, the CUDA library for DNA sequence alignment, with its possibilities and shortcomings. We will formulate the specifications for this integration and how to verify them.

3.1 Shortcomings of BWA

We will first introduce how the time taken by some crucial parts in BWA-MEM makes it a good candidate for acceleration. We will then take a look at GASAL2 architecture to see how it can be integrated in BWA-MEM to accelerate the extension phase of the mapper.

3.1.1 BWA-MEM computational parts

BWA-MEM is a full-featured DNA read mapper based on seed-and-extend. In Chapter 2 we described the processing steps of the mapper. It performs a lot of steps to align a batch of query sequences against a reference genome. The seed-extension part that can be accelerated with GASAL2 represents a considerable fraction of the total time.

We profiled the mapper by mapping paired-end reads to the the human reference genome "hg19" [2]. The read data set consists of two files containing pair-end reads with 5.2 million sequences each, all of them being 150 bases long. The read data set is SRR949537 [43] downloaded from Sequence Read Archive. We used 12 threads on our test machine, which will be presented later in Chapter 5. We reported the time needed for the execution of each thread, then took the mean of all of them. The time share is visible in Figure 3.1. We can see that the extension takes approximately 27% of the total time. Depending on the data set, this can go from 25% to 33%. This makes the extension an interesting part to accelerate, since it takes an reasonable fraction of the total compute time. The measurements also show that a theoretical speed-up of $1/(1-0.27) = 1.37\times$ can be reached by accelerating seed-extension.

3.1.2 Architecture of GASAL2

The GASAL2 library is a C++/CUDA library for DNA sequence alignment computation. In the beginning of the project, GASAL2 was able to perform global, local and the semi-global alignment. It features data compression on GPU: the goal is to quickly

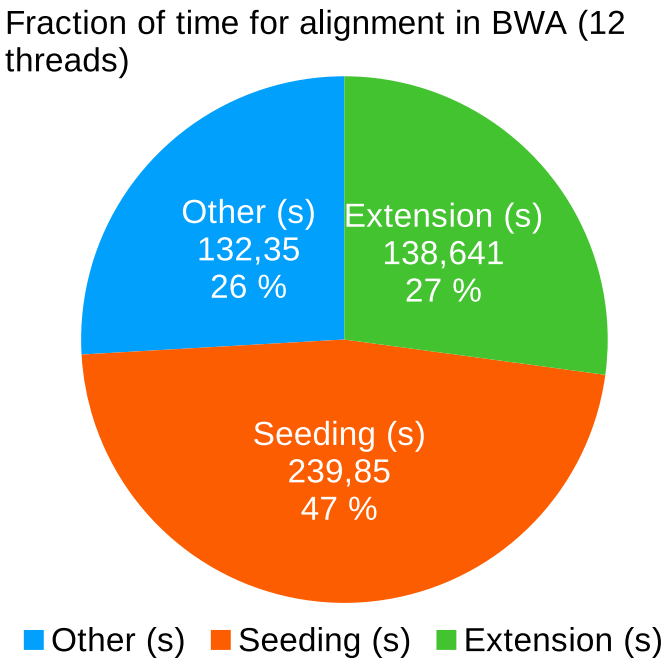


Figure 3.1: Fraction of time for seeding, extension in an example for BWA.

compress DNA strings directly in VRAM to fetch them faster when calculating the dynamic programming matrix.

Here is the list of features of GASAL2 before this thesis:

- Local, global, semi-global sequence alignment run on GPU,
- Operates on compressed data: there are 5 bases to describe, A, T, C, G and the unknown base N, hence needing a minimum of 3 bits to encore the base, but each based is stored on 4 bits to facilitate storing (more on this topic below)
- Data packing compresses the data directly on GPU: despite having to transfer uncompressed data from host to device, the speed-up provided by parallel processing when packing makes up for the longer transfer time,
- Capable to reverse and, or complement any sequence on the batch independently on the GPU.

GASAL2 works with batches of sequences to align. Here, we will present the original data structure of GASAL2 . It revolves around a single data structure called `gasal_gpu_storage_t` with the following fields, commented, on Listing 1. All fields from line 2 to 13 have their counterpart on the GPU, named the same way without the `host_` prefix in their name. Since the host and the device have distinct memories, data should be handed from the host side to the device before launching computation, then the results should be retrieved from the device when it is done. These

similar fields have been omitted for clarity, and only reminded with the comment `//[copies of the aforementioned field on the GPU]`.

This structure is initialised for a given size and a number of sequences. Host fields are filled up with the sequences data and their metadata (length, offsets). Trying to fill more than the capacity of the fields results in a forced exit.

GASAL2 went under an important structure rework at the beginning of this thesis to drastically improve its maintainability. In the beginning, it was entirely written in plain C and had a simple architecture consisting in three files:

- `gasal.h`: contains all function prototypes and structure definition
- `gasal.cu`: contains all functions code and the CUDA calls
- `gasal_kernels.h`: contains all CUDA kernels, duplicated

Although this may seem as an easy way to split the code, it grew up to the point where it became hardly manageable. The rewrite included separating the different parts of the code in distinct files, adding interfaces for structure filling, and a class to handle all parameters in an centralized way.

In addition, a shift to C++ has been operated to take advantage of C++ templates: they allow to produce different versions of a given function at compilation time. This will me more discussed on Chapter 4.

3.2 GASAL2 and the extension kernel

More details will be given in the workflow of GASAL2 and its distinctive features. Finally, we will outline a list of characteristics we need for GASAL2 to be integrated in BWA-MEM.

3.2.1 Typical workflow of GASAL2

We will now describe the workflow usually followed when using GASAL2. Figure 3.2 shows the processing flow in a graph. This flow can be instantiated multiple times from each CPU thread.

The first step is to fill the parameters used for alignment: match and mismatch scores, maximum number of sequences and maximum sizes. Then the GPU and host storages are initialised with this given size. Data sizes are specified in advance, so the initialisation takes an important amount of VRAM.

Then, the first available stream is selected. It is filled up with the query and target sequences and their metadata (lengths, offsets, operations for reverse/complementing if needed). Then it launches the non-blocking call for the GPU memory copies and kernel launches. These particular calls are represented in the yellow block. Note that no direct array links functions from outside the block to the inside. This is meant to show that the CPU, once making the non-blocking calls, directly jumps to the next blue box in the graph, and performs the next tasks.

```
typedef struct {
        uint8_t *host_unpacked_query_batch; // (string) the query sequences, butted
        ↪ together
        uint8_t *host_unpacked_target_batch; // (string) the target sequences, butted
        ↪ together
        uint32_t *host_query_batch_offsets; // array with the offsets to tell the
        ↪ sequences apart
        uint32_t *host_target_batch_offsets;
        uint32_t *host_query_batch_lens; // array with the length of each sequence
        uint32_t *host_target_batch_lens;

        int32_t *host_aln_score; // array with the alignment scores for all the
        ↪ sequences
        int32_t *host_query_batch_end; // array with the end position of the alignment
        ↪ on the query sequence
        int32_t *host_target_batch_end; // array with the end position of the
        ↪ alignment on the target sequence
        int32_t *host_query_batch_start; // array with the start position of the
        ↪ alignment on the query sequence
        int32_t *host_target_batch_start; // array with the start position of the
        ↪ alignment on the target sequence

        //[copies of the aforementioned field on the GPU]

        // Information about the batch
        uint32_t gpu_max_query_batch_bytes; // size of the buffer in the GPU (in
        ↪ bytes) for the query sequences
        uint32_t gpu_max_target_batch_bytes; // size of the buffer in the GPU (in
        ↪ bytes) for the target sequences
        uint32_t host_max_query_batch_bytes; // size of the buffer in the host memory
        ↪ (in bytes) for the query sequences
        uint32_t host_max_target_batch_bytes; // size of the buffer in the host memory
        ↪ (in bytes) for the target sequences
        uint32_t gpu_max_n_alns; // maximum number of sequences for the buffers on the
        ↪ GPU
        uint32_t host_max_n_alns; // maximum number of sequences for the buffers on
        ↪ the host
        cudaStream_t str;
        int is_free; // flag to know if the computation is being run (0) or if it is
        ↪ finished (1)

} gasal_gpu_storage_t;
```

Listing 1: the gasal_gpu_storage_t data structure.

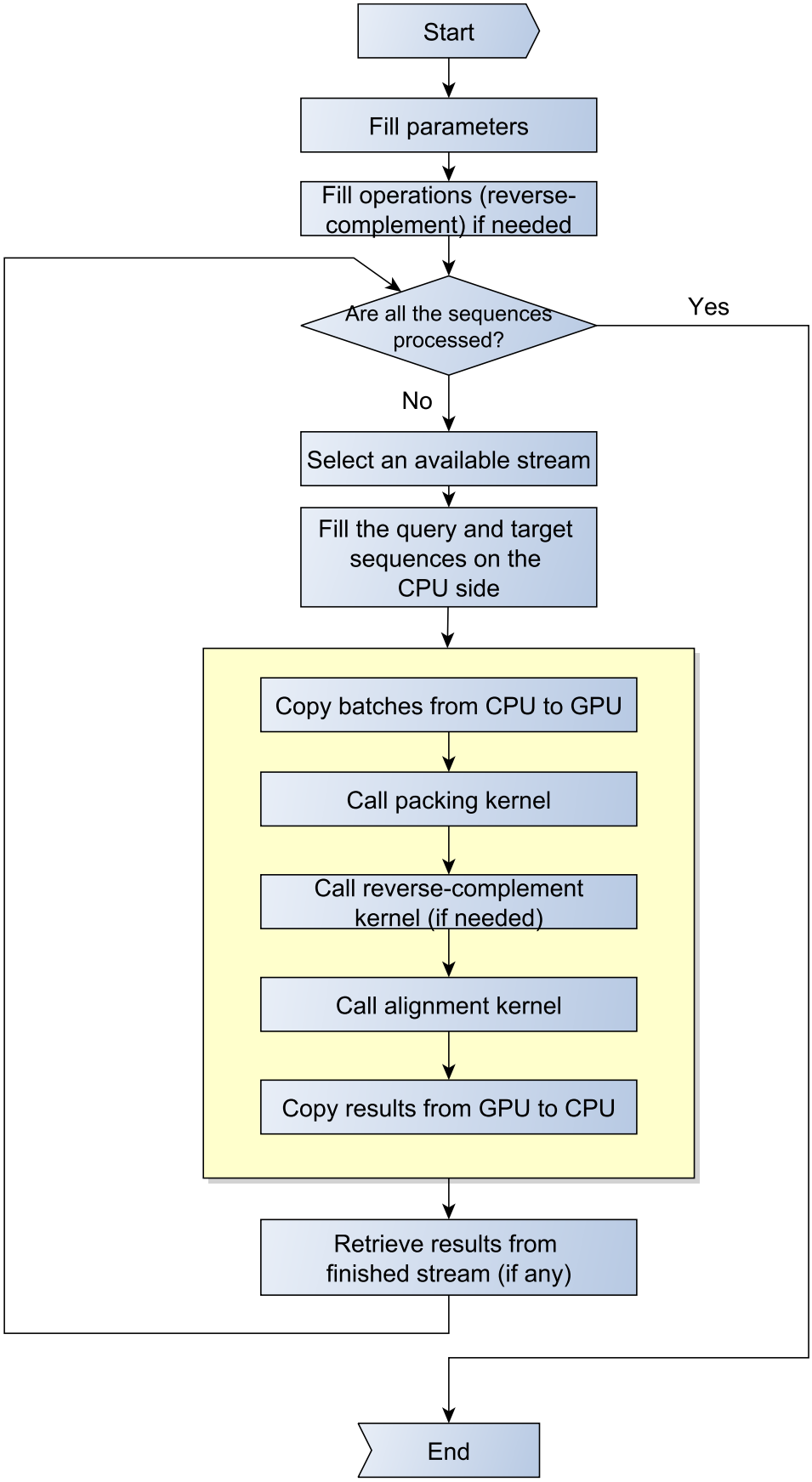


Figure 3.2: GASAL2 typical workflow

Finally, if a stream has finished, its results are retrieved. The streams allows for full CPU-GPU overlap execution, giving significant speed-up, as we will see later on.

An important feature of GASAL2 is that it packs and compresses the sequences on GPU to accelerate its handling in the alignment kernel. There are 5 different bases possible: A, T, C, G and the unknown base N. This means that a minimum of 3 bits is needed to encore 5 different values. The bases are then stored in 32-bits words. The goal is to allow the alignment kernel to fetch a pack of bases from the VRAM, put them in the cache, and then perform operations on the cached bases. This cache memory on a GPU is extremely small (for example, each SM can give its threads an access to 48KB of L1 cache for the GPU in our test machine [40]), but very fast and close to the computing resources. However, using 3 bits would mean needing a lot of bitwise operations to uncompress the data, since it would not align easily with 32-bits words. This is why the decision has been made to use 4 bits to encode the base value, hence, 8 bases are stored in each 32-bit word. This way, the kernel fetches two 32 bits words (one for query, one for target) and can easily uncompress the 8 bases stored. The dynamic programming matrix is then computed in tiles of 8×8 cells.

3.2.2 Characteristics of GASAL2 launches

The kernel launch has adaptable parameters to tune it for a given data set, number of CPU threads and number of GPU threads, depending on the hardware at hand. GASAL2 in its current state, has up to three different types of kernels:

- the packing kernel
- the reverse/complement kernel, if needed
- the alignment kernel (be it local, global, or else)

The packing and the reverse/complement kernels are very fast due to having very few computation, and take almost no time to complete compared to the alignment kernel. With various data sets, it has been measured that the packing kernel takes at most 0.6% of the kernels run time, and the reverse/complement kernel takes at most 4% of the kernels run time when all sequences must be both reversed and complemented. Data transfer time take at most 4% of the total time. Note that these times are given here as a rough estimate to show the importance of some tasks compared to others, and that precise measurements are presented on Chapter 5. As expected, the large majority of the time is taken by the alignment kernel.

The alignment kernel has a variable GPU occupation depending on the data processed. On our test machine, with a single CPU thread, the SM utilisation oscillates between 3% and 10%. Even though it seems a very low occupation, it is not a major problem since the kernels are compute bound and higher occupancy does not necessarily increase the performance.

3.2.3 Extension kernel behaviour

The aim of this project is to integrate GASAL2 in BWA-MEM to perform the extension part on GPU. To do so, we need to reproduce the behaviour of the extension function

of BWA-MEM to replicate it in GASAL2.

The original extension function has a BLAST-like seed-extension behaviour. It starts from a pair of target and query sequences and knowing the position of the beginning of the seed, its length, and its score, it proceeds as follows:

- it isolates the left part of both sequences, and reverse them.
- it computes a local alignment with the beginning score equal to the seed score. The end position of this alignment is equal to the beginning of the alignment of the two sequences (the alignment progresses from right to left, since the left parts have been reversed)
- then it isolates the right part of both sequences,
- it computes a local alignment with the start score equal to the left score previously computed (which includes the seed score).

In our case, this approach is unpractical because we would like to compute both sides in parallel to maximise GPU usage. In this fashion, we do not have other choices than making both extensions (left and right) start with a score equal to the seed score, which could make a small difference in the final result. We expected this error between the regular BLAST-like computation and our slightly different approach to be small enough to consider this approach valid. We measured it after implementation, with results shown in Chapter 5 and we reached the conclusion that the difference was effectively very small and that calculating both sides at the same time only using the seed score is an acceptable way to solve this problem. We call that paradigm the ”seed-only” method.

Another problem comes from the rigidity of GASAL2 memory scheme. Originally, GASAL2 was not able to scale in memory and adapt if some sequences were longer than expected, resulting in a crash of the program. This is particularly problematic when integrating it in BWA-MEM, because the number of seeds is unknown in advance, so each batch in GASAL2 need to have a variable size; plus, left and right parts of the target and query sequences have an unpredictable length. An extensible data structure should be implemented to tackle this issue with minimum overhead.

3.3 Proposed CUDA kernel

Due to the complexity of seed-extension acceleration on GPU described in the previous section, we would like to design a library interface with a kernel that complies with the following technical specifications:

- The library should be able to start with any amount of memory and extend it whenever needed. Moreover, the memory allocations should remain as scarce as possible.
- The kernel should reproduce the behaviour of BLAST-like kernels by allowing local DNA alignment with a non-zero starting score.

- The interface on BWA-MEM side with the library should have similar behaviour as the original extension function, but using the seed-only approach explained in Section 3.2.3, and the difference for the final results should be minimal.
- The end product should be readily available as an open-source project, respectful of the original licenses, and with a complete traceback of code production.

With the above requirements, now we have to state the goals and metrics used to verify that the end product meets the specifications:

- The integration of GASAL2 in BWA-MEM should effectively speed up the alignment process. We will compare the kernel execution times before and after acceleration with and without hidden time execution. We will also compare total execution times to verify the global speed-up.
- Using the seed-only paradigm should not bring any difference to the vast majority of the alignments. Still, there is no guarantee that the results will be exactly the same. Whenever the alignment differs, the difference with the original BLAST-like paradigm should be minimal. Result correctness will be examined. We will quantify the number of differences between the two results output from original BWA-MEM and its accelerated version, for our given data sets.
- The VRAM use should not take more than what it necessary. To verify this, we will take advantage of the feature that should allocate more VRAM when necessary to purposefully initialise the `gpu_storage` structure with a small size and let it grow bigger to just the right amount. This will keep VRAM usage as small as possible.

In the next chapter, we will detail the choices we made to meet these technical specifications.

4 Implementation

In this chapter, we will review the choices we made for the implementation of the features discussed earlier. We will discuss how the functionalities are implemented in GASAL2. Although some code snippets are provided, the detailed implementation is available at [44] and reviewing it can be helpful to understand the coupling between different parts of the software.

4.1 C++ kernel templates

4.1.1 Factorise kernel codes with templates

During the refactoring of the library code, we have to factorise the code from the kernels which have very similar implementation. Many similar kernels are not easy to maintain, as a single change have to be repeated many times in different parts of the code. However, it is not viable to directly make a generic kernel with many `if - then - else` conditions, since it would slow down the execution.

The solution we adopted consists in using C++ templates with light use of meta-programming [45] to generate multiple specialised version of a kernel at compilation time. We can then write branches in a clean fashion that would be resolved by the compiler. However, we cannot make any conditions on integer values from the get-go: since templates are resolved at compilation, the compiler can only resolve symbols on classes and types, and not values.

We then use a small template as shown on Listing 2 to derive our own types from integer values. We first define a template structure `Int2Type` with inside an enumeration with a single value, equal to the one given for its template instantiation. The goal is to instantiate empty structures which are seen distinctly by the compiler. Hence, at compilation time, `Int2Type<0>` is a structure name, `Int2Type<1>` is another structure, and so on.

We then create a template structure `SameType` that can be instantiated in two ways: if two objects of different types are passed (line 5), it bears an enumeration with value 0, and if types are the same (line 9), it bears value 1. Writing `if` blocks using the `SAMETYPE(a, b)` macro will expand it into 0 or 1, allowing the compiler to prune the unused branch automatically. Note that a template `std::is_same<X, Y>` is already available to achieve the same goal in the C++ standard library; but we could not use it in CUDA kernels since this library is not implemented there.

Finally, for all the kernels to be generated, explicit calls must be written in the source code. We decided to group all the kernels in three major kernel templates. Each of them can specialise to various extent:

- global, that does not have any specialisation,

```
template <int Val>
struct Int2Type{
        enum {val_ = Val} dummy;
};
template<typename X, typename Y>
struct SameType {
        enum { result = 0 };
};
template<typename T>
struct SameType<T, T> {
        enum { result = 1 };
};
#define SAMETYPE(a, b) (SameType<a,b>::result)
```

Listing 2: Meta-programming template to derive types from integer values

- local, on which we can select among 4 combinations:
  - start position calculation (on / off),
  - second-best score calculation (on / off);
- semi-global, on which we can select among 64 combinations:
  - if we allow to skip the beginning, the end, both ends or none of the ends of either query or target (making it 16 different cases),
  - start position calculation (on / off),
  - second-best score calculation (on / off).

It would be highly unpractical to list all these variations hard-coded, but we need them written before the compilation phase, where the templates are resolved. Subsequently, we rely on the C++ preprocessor and use a series of macros to develop a `switch - case` including all variations. This allows to write each kernel call exactly once, and condense all calls in a clean way.

4.1.2 Porting newer version of GASAL2 in GASE

Several modifications to BWA-MEM are done to integrate GASAL2 for its extension step. While some of them are already done when starting this thesis, substantial changes are still required to successfully integrate GASAL2 in BWA-MEM. GASAL2 aligns batches of sequences, and with a given batch of k alignments, it instantiates k GPU threads. Each thread runs the alignment kernel on for its own pair of query and target on the GPU. Before launching a kernel, we need to provide to the GPU a batch of data to process in parallel: we have to split the set of alignments in batches. This batching system was already implemented in a custom version of BWA-MEM for measurements called GASE-GASAL2 [46] [47].

Another important change in GASE-GASAL2 is about seed chaining. We reviewed earlier that seeds are processed one after the other, so that if a seed is found in an alignment, it is skipped and not extended. Since GASE-GASAL2 processes the alignments in batches, it cannot later filter seeds based on the actual result of their alignments. The solution implemented to circumvent this issue is to estimate that all alignments for all seeds result in a 85% coverage of the read. With this estimation, it filters seeds that are present in a range around the seed equal to 85% of the length of the read. This feature is necessary for a parallel aligner to be integrated in BWA-MEM: we cannot provide seed filtering based on the previous calculations, so we use an estimation that covers most cases. We keep this feature unchanged for our own implementation.

GASE-GASAL2 only uses standard local alignment kernel of GASAL2 and does not provide same results as in original BWA-MEM. It has only been developed to demonstrate how seed-extension could be accelerated. It uses an old version of GASAL2 which has become incompatible with the changes we made previously.

The first task has been to update GASE-GASAL2 to compile it with the newer version of GASAL2. In particular, the source code had to be ported to C++. Several wrong writing practices had to be fixed, including naming variables "or" (which is a keyword in C++), and defining clear context for function prototypes in the header files. In fact, C++ features name mangling for functions: this adds information at compilation to establish contexts where function names are declared valid. This feature is implemented among others for function overloading. Unless plain C functions are placed in `extern 'C' { ... }` blocks, they cannot be resolved at compilation, inducing a linking error, so some code clean-up has been done on this side.

4.2 Kernel implementation

We will now describe the kernel implementation.

4.2.1 Kernel writing

The kernel in itself is written in CUDA in the GASAL2 library. It should exactly replicate the behaviour of the original function in BWA-MEM, called `ksw_extend`. In the remainder of the thesis we will refer to the original function with its name, `ksw_extend`, and to its GPU implementation as the `ksw_kernel`.

We have seen before that previous kernels in GASAL2 were written as C++ template functions. It has been considered using the existing local alignment kernel template to derive an alternate kernel with the starting scores. This could have reduced the amount of code to maintain. However, there are many differences between the seed-only kernel and the local kernel. In particular, an important number of speed-up techniques are present in `ksw_extend` and the code written to implement them presented many differences with the way the local kernel is written. Therefore, we implement a new kernel that follows the seed-only paradigm.

The kernel is built by copying the original `ksw_extend` and renaming the variables to follow naming convention used in GASAL2 kernels. Initially, to test the implementation, we follow the loop structure used in `ksw_extend` in which the fact that data is compressed

is not exploited. The behaviour is simply to fetch the current base from query and target and calculate the cell of the dynamic programming matrix. Therefore, the pseudocode is the same as presented previously in Algorithm 1 in Subsection 2.2.4.

We introduce the use of compressed data by fetching the 8 bases in the 32-bit words from query and target. This adds two innermost loops in the algorithm to perform the alignment tile by tile, using square tiles of $8 \times 8 = 64$ bases.

Because of memory constraints, the whole matrix is not stored in memory. Since only the north, west, and north-west cells are needed for the computation, only one full row having the length of the query string and a column of 8 cells need to be stored. These are shown in Figure 4.1. All cells are computed in the order given by their number. When a full column is computed, the orange column moves one step on the right, taking the place of the cells numbered 1 to 8; then when cells 9 to 16 are computed, the orange column moves forward to take their place, and so on. This column is the western cells used to compute the score. When all cells in the tile are computed, the 8-cell section of the yellow row moves 8 cells south, taking the place of the cells numbered 8, 16, 24 ... 64. When the cyan tile is computed, the royal blue one is computed, then the navy blue, then the next line.

Finally, we introduce again some computing optimisations that we initially disabled, in particular, z-dropoff. This optimisation allows to skip the calculation of some cells of the matrix provided the score drops below a certain threshold. When this happens, it usually means that the alignment will not improve because the difference between the two sequences grew too big, so it is usually better to stop computing altogether instead of wasting time trying to align two sequences that are very different.

With all of this implemented, the pseudocode for the kernel is shown in Algorithm 2. Some parts detailed in Algorithm 1 are only summarised here for readability.

4.2.2 Score comparison

To verify the correctness of this implementation, a modified version of BWA-MEM is made from the original BWA-MEM [48]. This version is hosted on GitHub [49] and has different branches to switch from one behaviour to another, to compare the results with the GPU-accelerated implementation.

In a branch called `seedonly-time`, we introduce our seed-only method instead of the original BLAST-like behaviour. As the name suggests, the time for each processing step is monitored and reported at the end of the program for each CPU thread.

The results are in the Sequence Alignment/Map (SAM) format, commonly used for DNA mapping. SAM is text-based, extensively documented [25] and widespread among DNA-related programs. It is the default format output for BWA. Using `diff` [50] and other regular UNIX tools, we can compare the number of lines that differ between the reference program and a modified version. Knowing the original number of lines, we can derive a percentage of difference between the files, as shown on Listing 3.

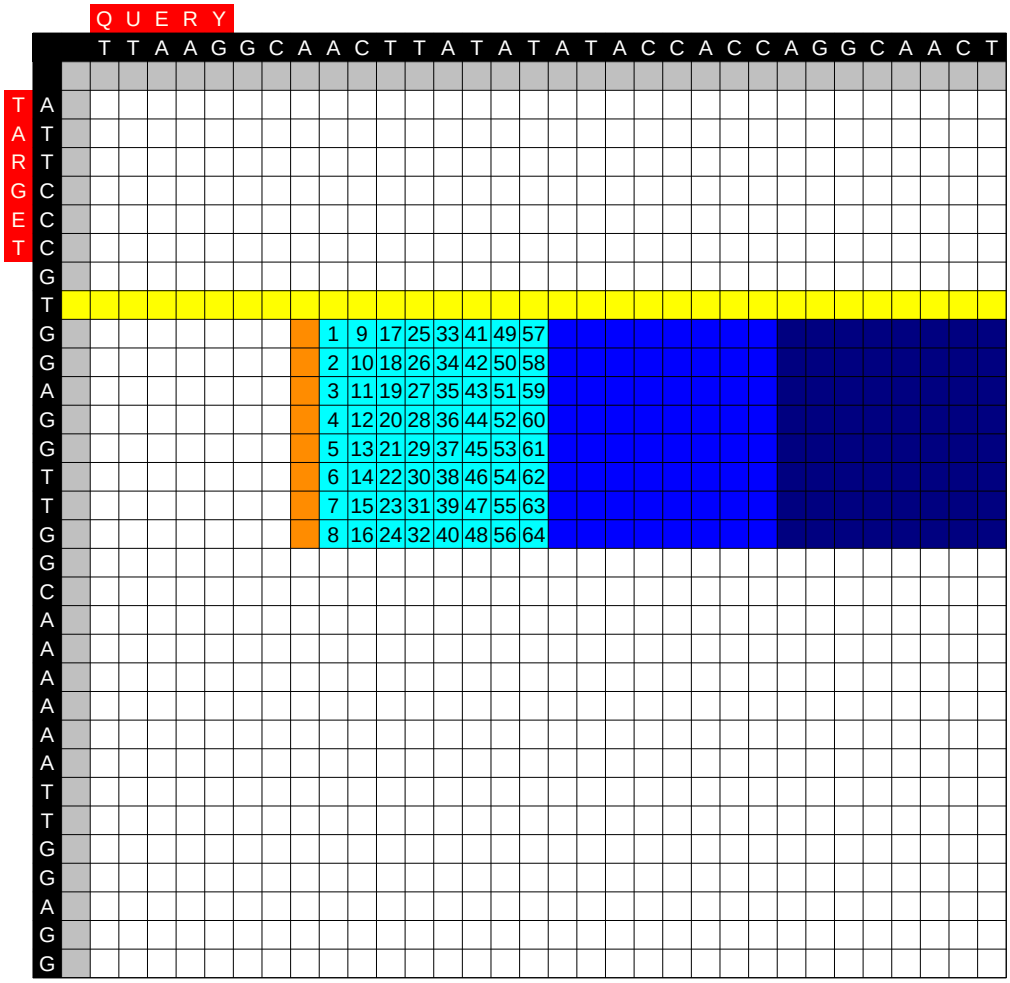


Figure 4.1: Visualisation of the tile-based matrix computation. Only the yellow row and the orange column are stored in memory.

4.3 Library integration

After the C++ adaptation of GASE, the demonstrator for BWA-MEM and GASAL2 integration, we obtained a valid version of BWA-MEM processing sequences in batches, originating from GASE-GASAL2.

BWA-MEM operates with the seed-and-extend paradigm, and GASAL2 can run the extension part. For a single query sequence, a variable number of seeds can be found.

For every seed found in the query, the target sequences is the region in the genome around the seed location. The extension step is then either:

- skipped, meaning that the seed is exactly the size of the sequence,
- or done only on one side, if the seed is located at the beginning or the end of the query sequence,

- or done on both sides, if the seed is in the middle of the sequence.

Most of the time, both sides have to be extended. We create two GPU batches. A simple approach would be to define a batch for the left side, and one for the right side. However, there is an optimised way to split the alignments between two batches.

GPU threads are grouped in *warps*. All threads in a warp run the same instructions. For maximal performance, it is advisable to run as many grouped thread as possible, and avoid differences in execution causing *thread divergence*. To avoid this thread divergence, sequences aligned by the GPU threads in the same warp should have similar lengths. But if one seed is located on the far left and another one on the right, knowing that all queries have the same length, the left parts of the extension will have very different lengths, as shown in Figure 4.2. Moreover, the processing time will be limited by the length of its longest alignment in the GPU batch.

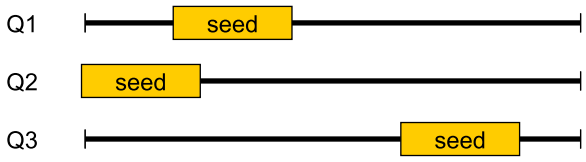


Figure 4.2: Illustration of chains (in yellow) being located on different places

To summarise, we note that :

- all reads have the same length,
- the final score is equal to the sum of the chain score, the left part score, and the right part score,
- to that extent, we don't need to know whose score is the left one and the right one (only the sum matters),
- and we would like to process both sides in parallel.

So instead of using two batches for left and right extension, we use two batches for long and short extension. This puts the long sides together to minimise thread divergence. The difference between the longest and the shortest extension in each batch is now at most half the length of the query sequence. For each chain, we log in a dedicated data structure if it has zero, one or two alignments, and on which part (left or right) the long alignment is. When both the "long" and "short" batches of extension are done, the scores are gathered.

Another problem arises: the seeds may have zero, one or two extensions, the short and long batches can have a different number of extension to make. To show this with an example, we can assume that we have three seeds as in Figure 4.2. For the sake of conciseness, only query sequences have been shown, but assume each of these queries have a corresponding target sequence with the same seed located in it, forming pairs of query-target sequences. Chains from Q1, Q2 and Q3 are found in this order, and

"Long" batch	**"Short" batch**
Pair 1, right part	Pair 1, left part
Pair 2, right part	Pair 3, right part
Pair 3, left part	

Table 4.1: Example of how batches are filled.

the `gpu_storage` structure is filled in this order too. Table 4.1 shows how the batches are filled for this case. Notice that the number of sequences are not the same in two batches, hence two sides of the same alignments are not at the same index on both structures. To circumvent this problem we added information in the data structure of BWA-MEM that contains the alignment scores. After both sides of the seed-extension of a batch are finished, this data structure is used to know how many alignments (0, 1 or 2) for a seed were computed, ensuring correct gathering. This is not an issue in the original software as the seed is extended first to the right and then to the left. But in our implementation we run both the extensions in parallel, with seed score as the starting score of the alignment, to get more parallelism. Hence, if we sum both left and right scores, we count the seed score twice. Therefore, we subtract the seed score from the total of extension scores of left and right side. When only one extension is made (or none), we do not have to do this subtraction.

4.4 Memory management

For GASAL2 to work correctly, all data fields in the `gasal_gpu_storage_t` structure must have their sizes specified at initialisation. This is problematic with BWA-MEM because sizes cannot be inferred beforehand. Even if we process a pack of 4000 reads, each of them have an unpredictable number of seeds. Furthermore, for each seed, the length of the query and target sequences are unknown. We show a schematic view for this problem in Figure 4.3. The GPU accelerated BWA-MEM align the sequences in packs that are selected by the CPU. This pack is then split into muliple batches of sequences, each batch being processed by a CUDA stream. The figure shows two packs of 3 reads each (note the read numbers on the left, Q1 to Q6). For each read, there can be different number of seeds. The pack is distributed evenly between CUDA streams in GPU batches. Reads of the same colour are processed by the same GPU batch. This is difficult to show with a few number of sequences like here (for the first sequence pack, reads Q1 and Q2 are in the same GPU batch and read Q3 is in another GPU batch), but for example, a pack of 4000 sequences is split into 4 GPU batches of 1000 sequences each, each assigned to one CUDA stream. After each stream has been given its GPU batch to process, the seeding phase runs for each read, and multiple seeds are found. For each seed, the left and right parts of the alignment are copied onto the GPU memory and the kernel runs to compute the alignment. Alignments of the same colour are processed in the same GPU batch, so in the same CUDA stream. For example, if we have 2 CUDA streams, all seeds for Q4 are extended by Stream #1, then all seeds from Q5 are processed by Stream #2, then Q6 seeds are processed by Stream #1 if we assume it has finished when Stream #2

was computing. We can see that the total number of alignment is different depending on the number of seeds, and each alignment has its own left and right lengths.

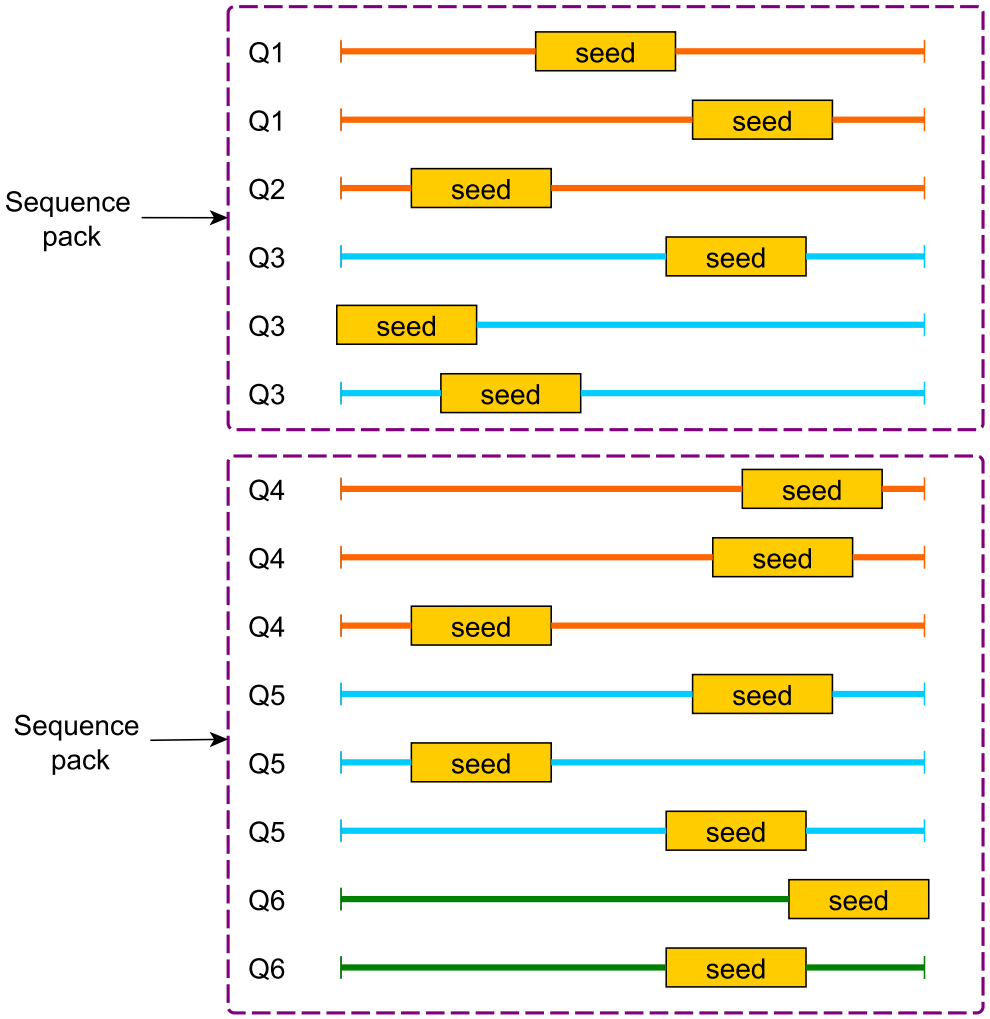


Figure 4.3: Representation of alignment distribution in sequences packs across GPU batches.

We would need some automatic resizing whenever the initial size is insufficient. For a given data set, during testing phase, we can estimate an upper bound. But this is not a good approach since we have to allocate much more memory than actually required.

In this section we will review the solutions we implemented. A simple approach was adopted for all the arrays carrying metadata for the sequences. We chose a more refined approach for the fields bearing the actual sequences to minimise overhead caused by reallocation.

4.4.1 Memory reallocation for arrays fields

A large number of fields in the `gasal_gpu_storage_t` are containing information about the sequences. These fields are arrays of length equal to the number of sequences. As such, they do not store a very large amount of data, and their separation is clear (each slot of the array is holding information about one single sequence). For example, we need to store the length of each query and target, the offset of each each query and target, and so on. These fields are first allocated on host side, then they are copied onto the device, so they are allocated with `CudaHostAlloc`. Since there are a large number of fields and they do not represent a major part of memory allocation, we decided to implement a simple realloc feature for all such fields. This function is displayed on listing 4. Note that on the `gasal_gpu_storage_t` shown on Listing 1, all fields do not have the same type: some are of type `uint32_t`, others are `uint8_t`, and so on. We made a simple use of C++ template to adapt this function to multiple types. It allocates a new memory area, copies the content of the former area, and frees it. It returns a pointer to the newly allocated area in memory filled with the previous data. This re-allocator is called by a larger function resizing all the necessary fields when the number of sequences filled exceeds the number of maximum sequences the data structure can hold.

The user of the library must take care of the sequence count and run the reallocator when needed. Once the fields are reallocated to a bigger memory area, they keep their new size, so they don't need to be re-allocated until the next time the new maximum capacity is exceeded. The library keeps track of size of the array and current number of sequences filled in it.

We chose a different approach for the arrays that hold actual sequences because they are much bigger and reallocation is time expensive.

4.4.2 Extensible data structure for sequences

Using the same reallocation for the sequences could have been possible, but unpractical. In fact, these memory allocations and frees are costly in time. While this is reasonable for smaller arrays having only 4-bytes elements (like `uint32_t`), it takes too much time with the sequences arrays. In the data structure, there are two fields to store the query sequences and the target sequences respectively. Each field is a single array and all sequences are put some following others in this array. If a batch has 1000 reads, there may be 10 000 to even 100 000 seeds for the whole batch, making it up to 200 000 alignments to performed (most seeds needing both left and right extension). Each base being stored in one byte, making them query and target arrays 20 to 30 times larger than the other arrays for sequence lengths, sequence offsets, or other information.

Due to this issue, we designed a structure of linked list to allocate more memory progressively without needing to move around already existing data. This structure shown on Listing 5 starts with a single element and each of them carries some metadata about its content size to ensure correct filling. It stores:

- the sequences, butted together, in the field `*data`,
- how many bytes are already stored, in field `data_size`,

- the total size available in bytes, in the field `page_size`,
- the data `offset`, which is equal to the data size of the previous element of the linked list,
- A flag telling is the structure can store new data or not, `is_locked`, set to 0 or 1,
- and of course, as for all linked list, a pointer to the next element.

The sequences are filled by a dedicated function from GASAL2 that takes care of allocating more memory if needed. This function takes as input the sequence to add, the storage structure, and if it is a query or target sequence to fill, Algorithm 3 shows the pseudocode of the function. We represent the fields belonging to an element with the symbol $\triangleright$, similar to the symbol `->` used in C++ to access the field belonging to a pointer of a structure.

In the beginning, the first non-locked element is selected. Then the algorithm operates in three main steps.

1. First (line 6), it checks if the selected element is the end of the linked list, and if there is not enough space in it. If so, the only solution is to create a new element to append at the end of the linked list.
2. Then (line 11), it checks if the current element is full and if there are more elements down the linked list. This happens when the linked list has created new elements during previous fillings and kernel launches. We already selected the first non-locked element, so if it is full, we simply jumps to the next one, and lock the previous one so that we don't try to fill it anymore.
3. At this point (line 16), we have reached an element which have enough free space for the sequence:
   - if we had an element available from the beginning, we reach this point without going through the previous `if`.
   - if the last non-locked element was full, we locked it and jumped to the next one, so we reached a non-full element,
   - if we were at the end of the list, we created a new element and jumped to it.

   We can now fill it with the sequence and its offset.

In the end, the function returns the current offset, corresponding to the total number of bytes present in the linked list, which is needed to continue the filling for the next sequence.

Globally, we have a linked list of structures filled by full sequences. We do not split the sequences between two elements. Therefore, each element is not filled up to its maximum capacity, leaving an area smaller than a sequence length unused.

When copying the batches onto the device, all elements are processed using `CudaMemCpy`. The size to copy is directly given by the field `data_size`. At this moment, all the elements in the linked list are set as "non-locked". This way, they can

be re-used for future launches. The linked list can only grow in size to minimise the possibility of having to do another allocation later on. This is why we implemented this "locked" flag to get the information if the element has been filled for the current batch, to know where to start filling up.

All in all, this allocation scheme allows to get more memory space without moving around already existing data by re-using all the previously created memory chunks in the linked list, and creating bigger elements. Since we would like to keep the number of allocations to a minimum, we allocate bigger memory chunks each time it is needed. The more allocations are done, the bigger they are: it becomes less and less likely to trigger a new allocation as we grow in size.

Wrap-up

In this chapter, we reviewed all the modifications and new implementation we carried out to successfully integrate GASAL2 in BWA-MEM. We started from a demonstrator (GASE-GASAL2) already processing in pack of reads, and we ported it to C++ to run it with the newer version of GASAL2. We created a CUDA kernel that replicates the behaviour of the original C function in BWA-MEM. We integrated the library with our "seed-only" paradigm to tackle the issue of processing both sides of the extension in parallel. This created its own issues with score synchronisation between left and right sides, and we addressed this problem by keeping track of the number of alignments to guarantee correctness of the score. The memory management is now done with the help of memory reallocation for small arrays and using linked list of elements for the actual sequences arrays to minimise the number of time expensive memory reallocation operations. Finally, we tested this implementation in our hardware with a sample data set, that we will present in the next chapter.

Algorithm 2 ksw_kernel algorithm

```
1: procedure SEED-EXTENSION KERNEL(query_string, target_string, query_length, target_length, initial_score, zdrop)   ▷ TILE_SIDE = 8
2:     Initialise score
3:     Initialise maximum score score ← 0 and maximum positions (max_query, max_target) ← (0, 0)
4:     query_batch_regs ← query_length/TILE_SIDE + 1
5:     target_batch_regs ← target_length/TILE_SIDE + 1
6:     for target_tile_id from 0 to (target_batch_regs − 1) do
7:         gpac ← Fetch a pack of 8 bases from target_string at position target_tile_id (int32)
8:         for target_base_id from 0 to TILE_SIDE − 1 do
9:             i = target_tile_id ∗ TILE_SIDE + target_base_i
10:            target_base_j ← read base in gpac at position target_base_id
11:            Compute first column of the matrix
12:            for query_tile_id from 0 to (query_batch_regs − 1) do
13:                rpac ← Fetch a pack of 8 bases from query_string at position query_tile_id (int32)
14:                for query_base_id from 0 to TILE_SIDE − 1 do
15:                    query_base_j ← read base in rpac at position query_base_id
16:                    j = query_tile_id ∗ TILE_SIDE + query_base_j
17:                    if i < beg then
18:                        continue
19:                    end if
20:                    if i >= end then
21:                        break
22:                    end if
23:                    Compute S[i, j], G_A[i, j] and G_B[i, j]
24:                end for
25:            end for
26:            if j == query_length then
27:                Store global score and global end position
28:            end if
29:            if maximum calculated score > previous maximum then
30:                Store new maximum score and new end position (max_query, max_target)
31:            else if zdrop > 0 then
32:                if the current score dropped more than zdrop since the last maximum score then
33:                    break
34:                end if
35:            end if
36:            Cell-skipping: beg ← first cell of the stored row with non-zero score
37:            Cell-skipping: end ← last cell of the stored row with non-zero score
38:        end for
39:    end for
40:    Find i_max and j_max for which S_{i_max,j_max} = max(S_{i,j})
41:    score ← S_{i_max,j_max}
42:    end_position_query ← i_max
43:    end_position_target ← j_max
44: end procedure
```

```bash
#!/bin/bash
DIFFLINES=$(diff --suppress-common-lines --speed-large-files -y $1 $2 | grep "[|><]" | wc -l)
TOTALLINES1=$(cat $1 | wc -l)
PERCENTDIFF=$(bc -l <<< "scale=2; (100*$DIFFLINES)/$TOTALLINES1")

echo "Different lines between " $1 " and " $2 " = " $DIFFLINES
echo "Total numer of lines in" $1 " = " $TOTALLINES1
echo "Difference = " $PERCENTDIFF "%"
```

Listing 3: Bash script to show percentage of difference between two files

```cpp
// Function for general resizing
template <typename T>
T* cudaHostRealloc(void *source, int new_size, int old_size)
{
        cudaError_t err;
        T* destination = NULL;
        if (new_size < old_size)
        {
                fprintf(stderr, "[GASAL ERROR] cudoHostRealloc: invalid sizes. New size < old size (%d < %d)", new_size, old_size);
                exit(EXIT_FAILURE);
        }
        CHECKCUDAERROR(cudaHostAlloc(&destination, new_size * sizeof(T), cudaHostAllocMapped));
        CHECKCUDAERROR(cudaMemcpy(destination, source, old_size * sizeof(T), cudaMemcpyHostToHost));
        CHECKCUDAERROR(cudaFreeHost(source));
        return destination;
};
```

Listing 4: Reallocation function for CUDA allocated fields.

```
1  struct host_batch{
2          uint8_t *data;
3          uint32_t page_size;
4          uint32_t data_size;
5          uint32_t offset;
6          int is_locked;
7          struct host_batch* next;
8  };
9  typedef struct host_batch host_batch_t;
```

Listing 5: The linked list structure for sequences on host.

Algorithm 3 Behaviour of the sequence filler function

function FILLS ONE SEQUENCE IN AN LINKED LIST ($gasal\_gpu\_storage\_t$, seq, seq_length, seq_offset, $data\_source$)
    Select query or target linked list from $data\_source$
    Select first non-locked linked-list element, e
    Compute padding length $padding\_length$
    $total\_length \leftarrow padding\_length + seq\_length$
    if e is the last element **and** $total\_length \geqslant e \triangleright page\_size - e \triangleright data\_size$ **then**
        Create new element twice as big
        Append it to the last element e
        Select last element (newly added) of the linked list, define it as e
    end if
    if e is not the last element **and** $total\_length \geqslant e \triangleright page\_size - e \triangleright data\_size$ **then**
        $e \triangleright next \triangleright offset \leftarrow e \triangleright offset + e \triangleright data\_size$
        Mark e as locked
        Select next element $e \triangleright next$ of the linked list, define it as e
    end if
    Store the sequence in $e \triangleright data$ add the padding
    $e \triangleright data\_size \leftarrow e \triangleright data\_size + total\_size$
     return $offset$
end function

5 Measurements

We are now ready to test our software for short DNA reads produced by Illummina sequencing machines. BWA-MEM is the industry standard for mapping short DNA reads. We will first review our testing environment and the data set we use for the rest of this thesis. Then we will show the performance improvement we get from the acceleration. In the best cases we can drastically reduce the extension time due to CPU-GPU overlapped execution. We will take a closer look to the results validity and the difference between our solution and the original BWA-MEM. Finally, we will show the resource use with our sample data to give the reader a glimpse of the requirements of our solution.

5.1 Experimental setup

First we review the conditions in which we run our measurements.

5.1.1 Environment definition

The specifications of the machine used for the measurements is shown in Table 5.1. We report hardware references in Table 5.1 and software specifications on Table 5.2.

Hardware	**Reference**
CPU	2 * Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz [51]
RAM	32 GB DDR3
GPU	NVIDIA GK110BGL - Tesla K40c [52]
VRAM	12 GB GDDR5

Table 5.1: Hardware specifications

We apply an optimisation level of -O2 for the GCC compiler since we noted that it gave significantly faster execution than -O0 and -O1. O3 optimisation level makes no difference in speed with respect to -O2. For NVCC (the NVIDIA CUDA compiler) we keep the -O3 optimisation level that was originally used in all GASAL2 papers [30].

The row "NVCC compiler options" has two particular defines passed with -D. `MAX_SEQ_LEN` corresponds the maximum size of the sequence for the extension kernel. As we have seen previously in Figure 4.1, we have to store the score of a whole row (in yellow in the figure). This size is defined at compilation time to avoid dynamic memory allocation inside the kernel, which makes it noticeably faster. Having to define this number at compilation is not very problematic because we know that Illumina machines produce reads with fixed lengths. `N_CODE` defines the value used for the unknown base "N", since

Software	Reference
Operating System	Red Had Enterprise Linux 7
Linux kernel	3.10.0-957.5.1.el7.x86_64
C compiler	g++, GCC version 4.8.5 20150623
CUDA version	10.1.105
C compiler options	`-g -Wall -Wno-unused-function -O2 -msse4.2 -std=c++11 -fpermissive`
NVCC compiler options	`-c -g -O3 -std=c++11 -Xcompiler -Wall,-DMAX_SEQ_LEN=\$(MAX_SEQ\_LEN),-DN_CODE=\$(N_CODE) -Xptxas -Werror -lineinfo --ptxas-options=-v --default-stream per-thread`

Table 5.2: Software specifications

each program can use its own. For example, the test program provided with GASAL2 registers the bases with their ASCII values, so the N code is `0x4E`; but BWA-MEM codes A, C, T, G, and N with integer values from 0 to 4, the "N" code being 4.

All tests are run with 1 and 2 streams, and with 1, 2, 4, 6, 8, 10, and 12 threads. We will observe how overlapped execution provides faster results while using 2 streams. In addition, we run the program with only one stream to force it to wait for its completion before re-filling the data structure containing the query and target sequences to align: this allows to measure the actual kernel time, since the CPU is actively waiting for the kernel to complete.

5.1.2 Data sets

For the tests, we use two paired-end data sets that are mapped to a reference genome. Each data set is constituted of two files. The first file contains the first read of the pair, which is the direct strand of DNA. The second file contains the second read in the pair, made by reading the strand from the other end, hence being the reverse-complement of the first strand.

Data set #1 called "SRR150" [43] is made of 150-base long paired-end DNA reads. Each file has 5.2 million sequences. These are the typical lengths that a state-of-the-art Illumina sequencer can produce. Data set #2, that we called "SRR250" [53] has DNA reads with 250 bases each. This pair of files contains 8.3 million sequences in each file.

The reference we used is the Genome Reference Consortium Human Build 37 (GRCh37), also named "hg19" [2]. It is the genome of a male human being (with X and Y chromosomes) sequenced in 2009. This whole genome takes around 3.2GB in its plain text version. Before running the mapper, an index of reference genome is created with BWA-MEM indexing tool. The index is the Burrows-Wheeler transform of the reference genome and has size of about 4.8 gigabytes. The seed-extension stage of BWA-MEM does not require the index. It only requires the original reference genome FASTA file which is 2-bit packed, and hence around 800 megabytes large. Since the target sequences extracted from the reference genome on the CPU, the file is not copied

to the GPU to accelerate the seed extension.

5.2 Performance measurement

We will measure now the execution time of our solution, that we called **bwa-gasal2**, and compare it to the mainline version of BWA-MEM version 0.7.17 [48], the latest release at the time of writing. For the measurements, bwa-gasal2 processes sequences in batches of 1000, which gives between 10 000 and 100 000 seeds to process in a single GPU stream.

5.2.1 Data set SRR150

For the first data set, full program execution times are given in Figure 5.1a and kernel speed-ups against BWA-MEM in Figure 5.1b. We can notice that the global speed-up fluctuates quite a lot when using different threads, but no trend can be inferred. This may be due to our testing conditions or simply due to the nature of the data set, the results may present uneven behaviour depending on how data is split between threads. As we will see later on, the global speed-up profile with respect to the number of threads has a more consistent trend with the second data set.

We obtain the most important speed-up using a single thread, reaching 1.35× the original program speed. This is promising, but is hardly a good indicator: in fact, these kind of program are usually run with multiple threads, and as we mentioned on Chapter 3, a single thread running the accelerator is far from sufficient to saturate the GPU computing resources.

In the case of 12 threads, the total speed-up almost reaches 1.28×. It is below our theoretical maximum of 1.37×, but it is still an important acceleration. Many factors can participate in making this difference. In particular, memory filling for the batches requires a significant amount of time on the CPU side, in addition to the memory copies from host to device. In this case, we started with an already large memory allocation, to avoid any overhead caused by new allocations.

In all cases, the difference between one and two streams is striking, with the speed-up even leaping from 1.21× to 1.28× in the 12 threads case.

When isolating the kernel times and kernel speed-up, we get the measurements shown respectively in Figure 5.2a and Figure 5.2b. It is important to notice that what is reported as ”kernel time” for the 2-stream variant represents only the time during which the CPU is actively waiting for the GPU results. In other words, we have disabled overlapped execution when using one stream, and enabled with 2 streams. We call this ”visible kernel execution time”.

With overlapped execution, in the 2-stream version, the visible kernel time is an order of magnitude shorter. In this case, the kernel times were hugely different, so we had to display them in log scale. While the speed-up reaches 4.7× in single stream with 12 threads, it jumps to almost 16×, effectively shrinking down the compute time of the extension by the same amount. Single-thread results are even more impressive, with a kernel speed-up reaching 40× with overlapping, and only 6× when actively waiting for the result.

5.2.2 Data set SRR250

We run the same measurements on data set SRR250. Its reads are longer, so they take quite a substantial amount of time to complete. Execution times and overall speed-up are shown in Figure 5.3a and Figure 5.3b, respectively.

For this data set, the extension time is taking approximately 33% of the total time, so we would expect a bigger speed-up, with the theoretical maximum being 1.5×. With 2 streams and single thread, we can effectively reach a speed-up of 1.41×, which gets somewhat close to the theoretical maximum. With 12 threads, the speed-up is 1.29×. This is still a valuable improvement over the original version, and in all cases, overlapped execution gives a substantial boost in performance. When comparing the single stream version to its 2-stream counterpart, we note that using 2 streams definitely helps shrinking the execution time.

We will now examine the kernel times and speed-up, in Figure 5.4a and Figure 5.4b respectively.

Kernel running times for BWA-MEM (running on CPU) are steady. As the number of threads increases, we get closer to a plateau as the execution time hardly gets any lower for all programs. Again, we can remark how overlapped execution helps in computing it faster with two streams. The difference is tremendously high for one thread, with the CPU waiting time for the GPU to complete being 5 times lower with 2 streams (from 1125s with 1 stream to 197s). When using 12 threads, the time the CPU waits for the kernel to finish is 2.5 lower for the 2-stream version (from 250s for 1 stream to 89s for 2 streams). When using multiple threads, the overhead caused by filling a lot of GPU batches and copying them to the device is substantial. It is all the more the case when using twice as many streams, having twice as many structures needing filling and being copied to the device. This could explain why the speed-up boost got from the overlapped execution tends to get smaller with longer reads for 2 streams.

We can also see whether increasing the number of streams could give any performance improvement. Results are shown in Figure 5.5. As we can see, using more than 2 streams does not provide in any improvement.

5.3 Correctness measurement

As explained before, we use the `diff` utility to compare two SAM output files. Having the number of different lines between two files, we can calculate a percentage of difference. To be able to compare the outputs, they must be in the same order: this is why we have to run the program on single thread for this verification. We measure the influence of the seed-only paradigm on the results. To verify that our GPU kernel is correct, we implement a ”seed-only” version of BWA-MEM and verify that the results from this modified version and our GPU-accelerated version are matching. We obtained the exact same outputs for all results that we verified comparing their checksum with the `sha256sum` command.

It is worth noting that, in addition to the best alignment with its score, BWA-MEM outputs alternative alignments which may lead to different, lower quality results. In the case of seed-only paradigm, it regularly happens that the main result is matching,

yet the secondary alignments present very slight differences. This could be because of the seed filtering heuristics applied: since we cannot filter seeds based on the previously computed results, we assume every seed to reach 85% of the maximal possible alignment. But since these secondary scores are rarely useful in real use cases, we also compared the outputs without these results, to know the difference only for the final alignment. Since the output files are several gigabytes big, using regular tools like `sed` are slow and unpractical, processing a single file in tens of minutes. To circumvent this issue, we wrote a small C program (that processes files in seconds) to remove unnecessary results and filter the results depending on the alignment quality. Alignment quality is a number located in the 5th field of each line, and ranging from 0 to 60. We are not interested in having differences if BWA-MEM and bwa-gasal2 are outputing a low quality alignment. The results are presented in Figure 5.6. We ran the comparison between:

- BWA-MEM and bwa-gasal2 complete output, (first bar),
- BWA-MEM and bwa-gasal2 with secondary scores removed (second bar),
- BWA-MEM and bwa-gasal2 with secondary scores removed and all alignments with a quality lower than 50 removed (third bar).

We notice that the seed-only paradigm globally introduces a noticeable number of differences. 11.23% of the lines are not matching. Many of the differences are due to reordering of the secondary results. Still, when we set aside the secondary results, which are mostly neglected in downstream analysis, this percentage drops to 2.48%. Finally, when we remove low quality scores, the number of differences goes down to 1.82%. Again, this small difference can come from seed filtering heuristics, as well as the seed-only paradigm itself. This threshold of 20 for the quality score is the usual value taken for downstream analysis. This is small enough to consider it acceptable.

5.4 Resources use

First we will discuss GPU memory use, then Streaming Multiprocessors (SM) utilisation.

GPU memory use grows strictly linearly with respect to the number of streams and threads. When going from one to two streams, bwa-gasal2 takes twice as much memory. In this section, we will present the case we deemed more representative of real use.

GPU memory (VRAM) use is a delicate topic to assess, since it highly depends on various factors:

- the GPU, since the VRAM available may not be sufficient to saturate its computing resources,
- the CPU, because how many threads we can instantiate directly impacts how much VRAM we will need,
- the RAM, since the sequences also need to be loaded in RAM before being copied to the GPU (although this is rarely a limiting factor, since a given machine usually has far more RAM than its GPU has VRAM),

- and of course, the data set running.

We cannot conduct measurements that are corresponding to any algorithmic truth in this case. Consequently we will present the memory use as a case study corresponding to our current machine and data sets, and we will see if, in our example case, the memory use can be deemed as reasonable. We consider the GPU we used (Tesla K40c) with its 2880 CUDA cores and its 12GB or VRAM, as representative of the accelerators generally used.

For this, we start bwa-gasal2 with a substantially low amount of memory, and let the automatic memory extension grow until the end of the program. The memory use for data set SRR150 and SRR250 is shown in Figure 5.7. We measured memory use for what we consider a regular use case for our machine : 12 threads, with 2 streams.

In these tests, we need around 1570 MB for a data set with sequences of 150 bases, and 2730 MB for sequences 250 bases long. We use less than 1/6th of our total available memory with our worst case scenario. In particular, this allows to use an accelerator with much less VRAM available if needed. Moreover,it leaves more memory to accelerate other parts of the application.

We also measure the SM utilisation in percents with the `nvidia-smi` utility. We log this utilisation during the program execution with a resolution of one measure per second. Results are available for data set SRR150 for 1 and 2 streams in Figure 5.8 and Figure 5.9 respectively.

We can notice several interesting points:

- SM use fluctuates noticeably when the GPU is actively used. While the mean is around 70%, some peaks to 90% are present. Our GPU is not completely used, but we already take the major part of its computing resources.

- Globally, the GPU stays active most of the time, so we are efficiently using its computing power.

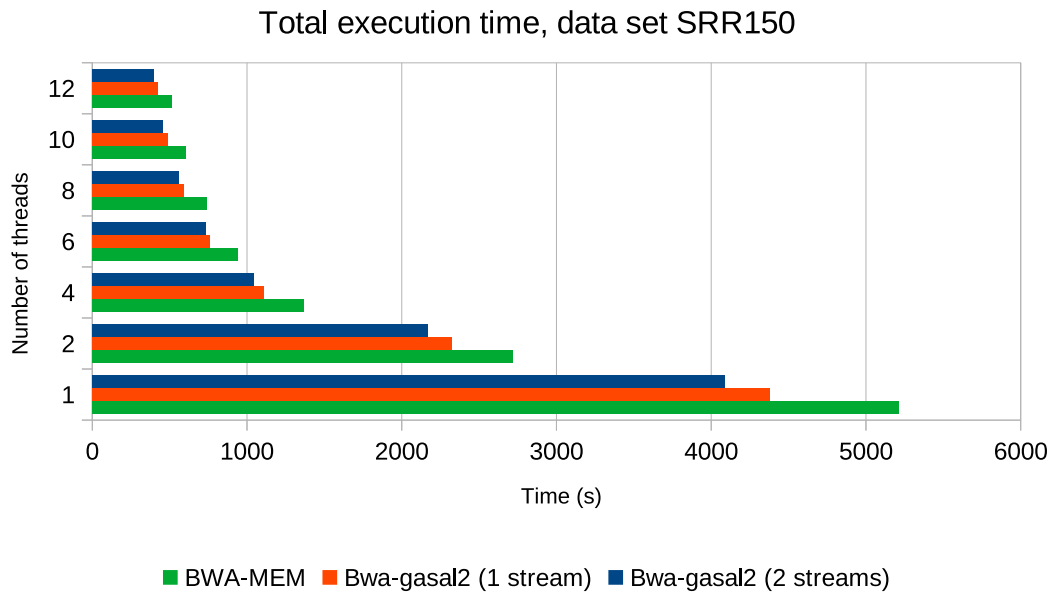
- SM utilisation is a bit lower with a single stream, which is logical: with two streams, the GPU may have to compute both data sets at the same time when they overlap, which leads to a higher SM use. Still, with 2 streams, the SM use is by no means the double of when we use 1 stream: streams are rarely overlapping, meaning that using 2 streams allows to use the resources more efficiently without saturating the GPU.

- During the mapping, the GPU can be idle for short periods of time. Depending on the data it has to process, it can be one second (in the beginning) up to 5 seconds (at the end). These idle periods are moments where the CPU is still running the seeding part. There is room to accelerate another part, for example seeding, on the GPU: this could bring us close to saturation of the computing cores.

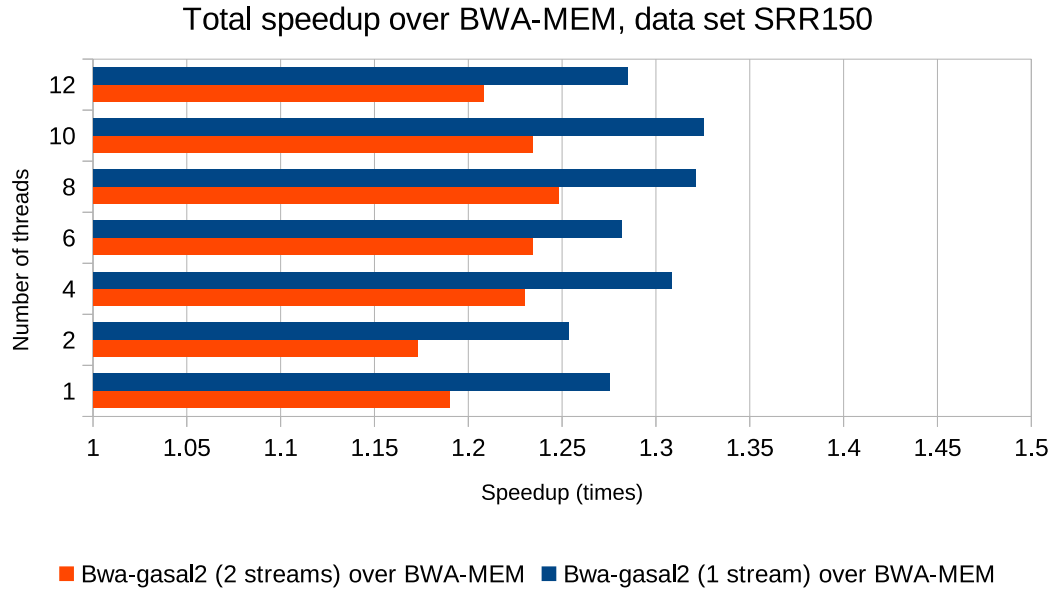
In the case of data set SRR250, results are available for one stream in Figure 5.10 and for two streams in Figure 5.11.

For this data set, the GPU is noticeably more active, as the SM utilisation is most of the time above 80%. With two streams, it even reaches 100% use many times. Still,

there are the same moments where the GPU goes idle for up to several seconds. These moments could still be put to use for another part of the mapping process.

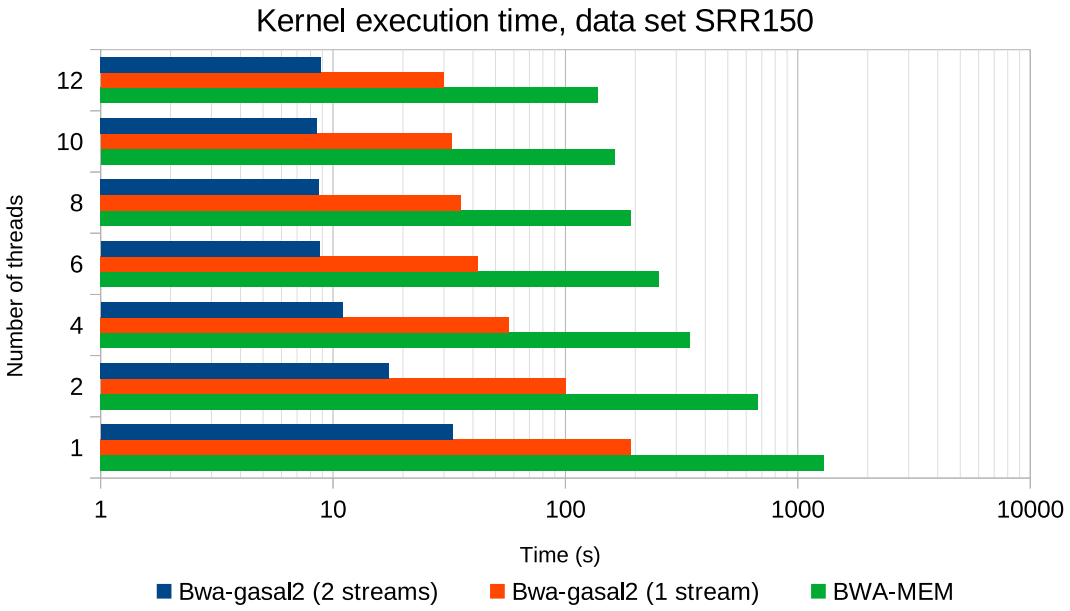


(a) Total execution time for SSR150

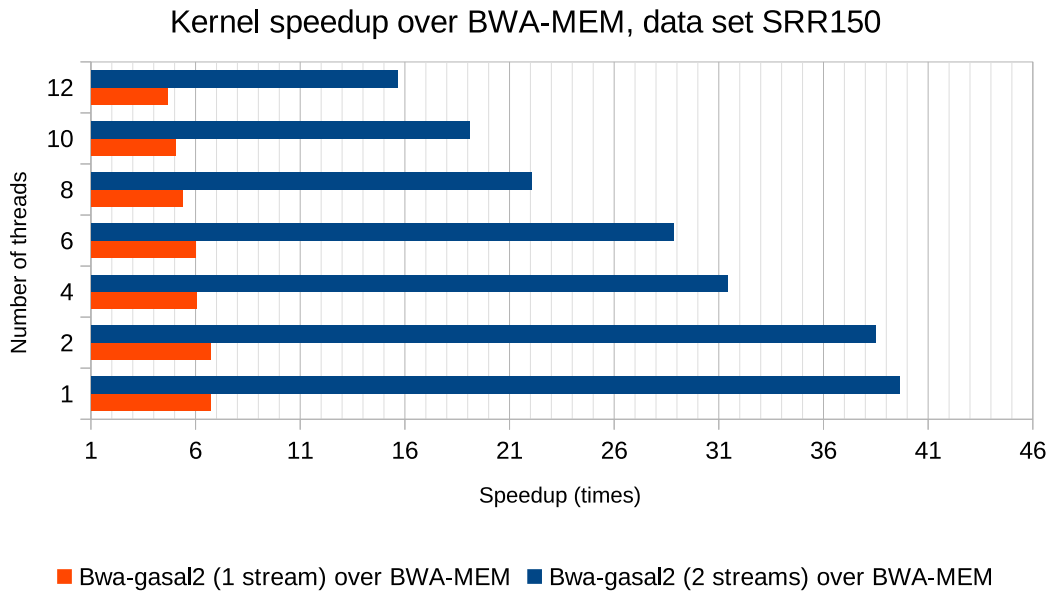


(b) Speed-up for the whole program execution for SRR150

Figure 5.1: Program results for data set SRR150

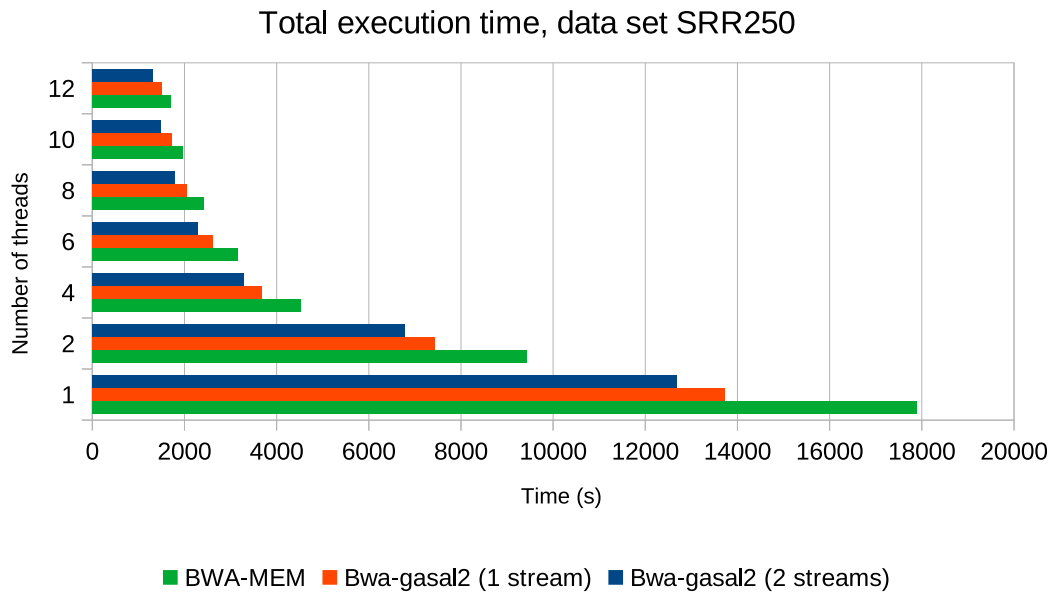


(a) Visible kernel execution time for SRR150

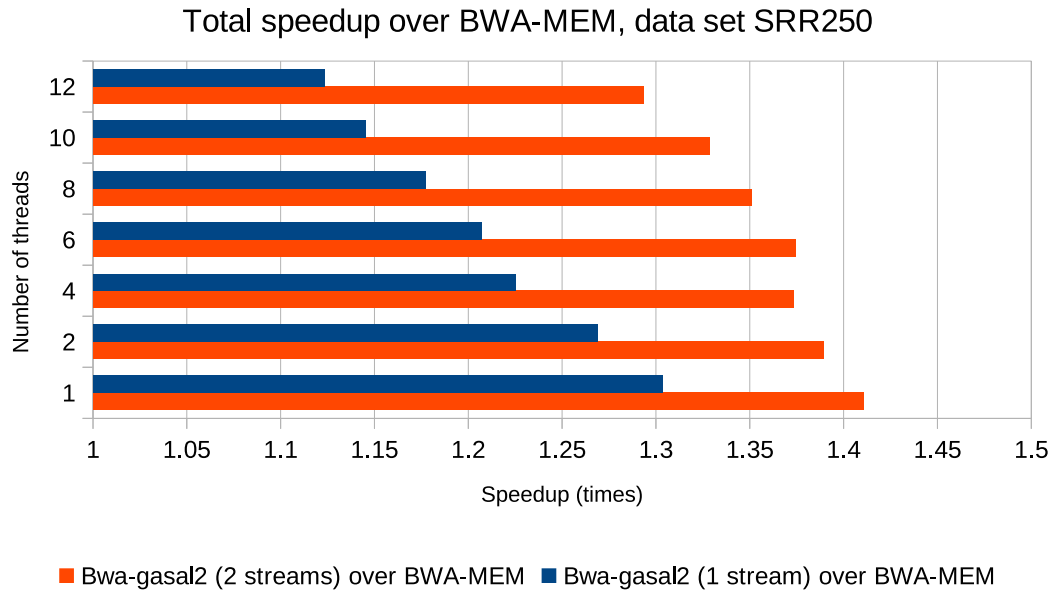


(b) Speed-up for the kernel execution for SRR150

Figure 5.2: Kernel results for data set SRR150

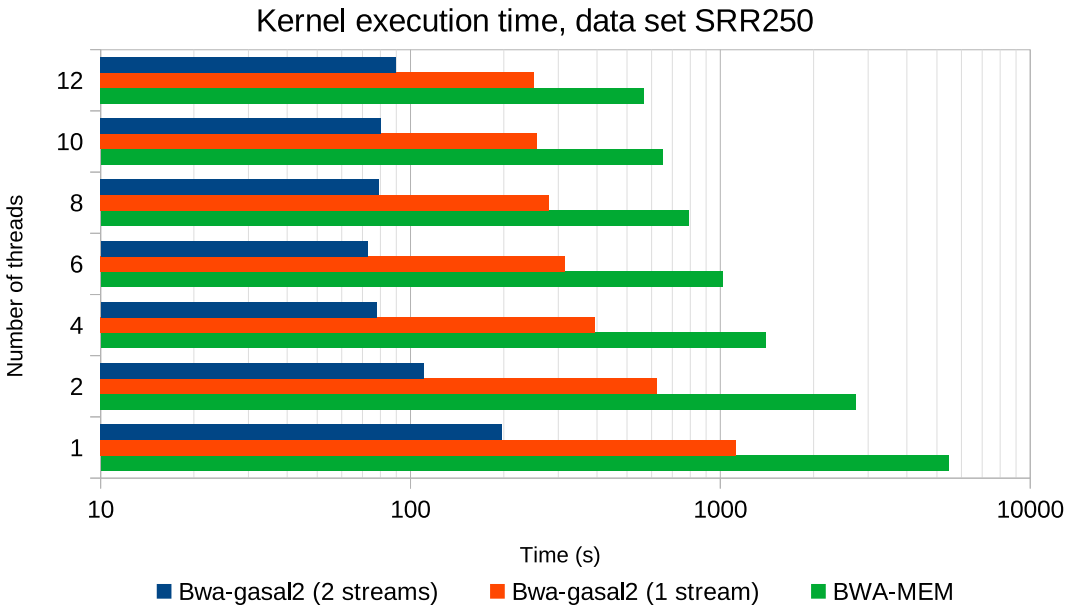


(a) Total execution time for SSR250

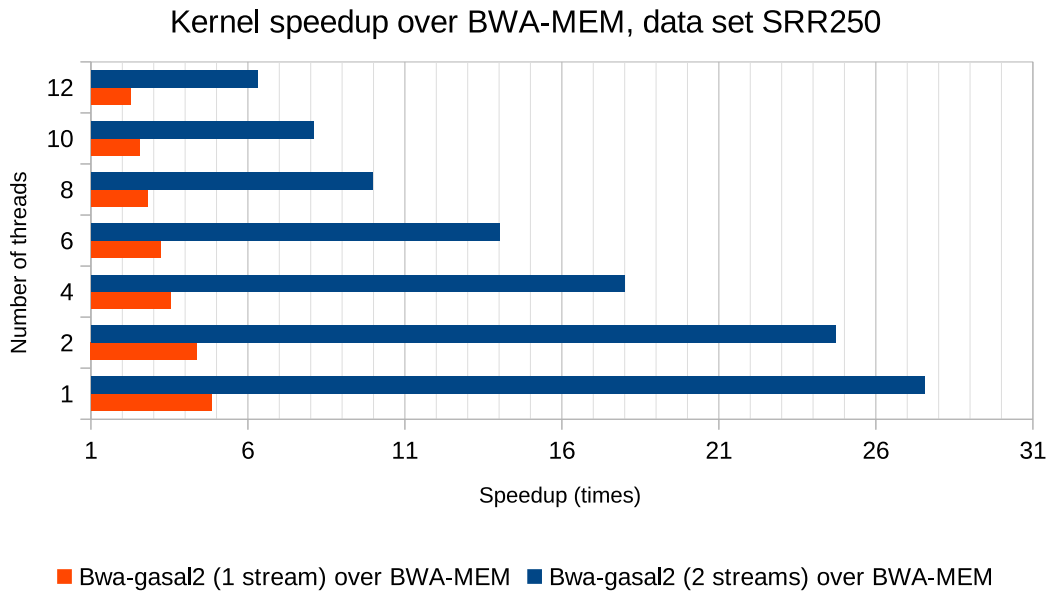


(b) Speed-up for the whole program execution for SRR250

Figure 5.3: Program results for data set SRR250



(a) Visible kernel execution time for SRR250



(b) Speed-up for the kernel execution for SRR250

Figure 5.4: Visible kernel execution for data set SRR250

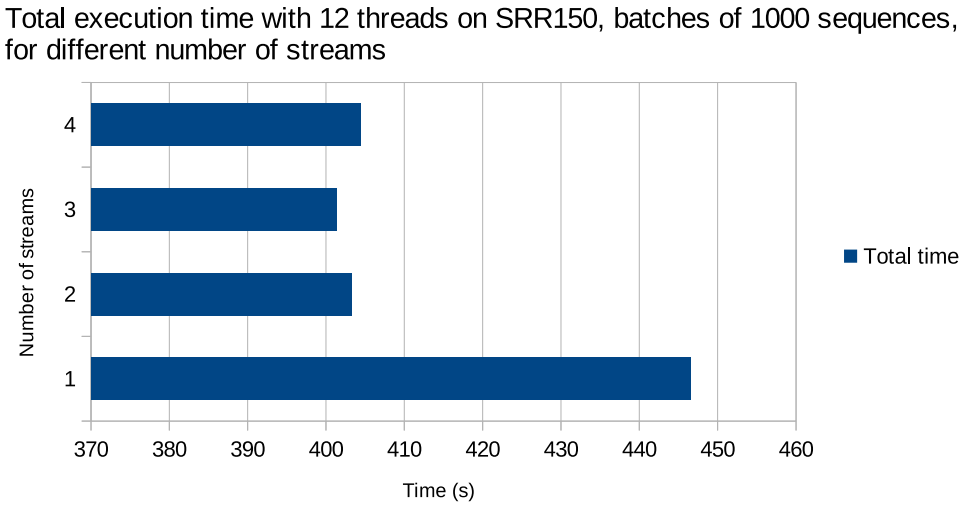


Figure 5.5: Execution time for 12 threads and different number of streams, data set SRR150

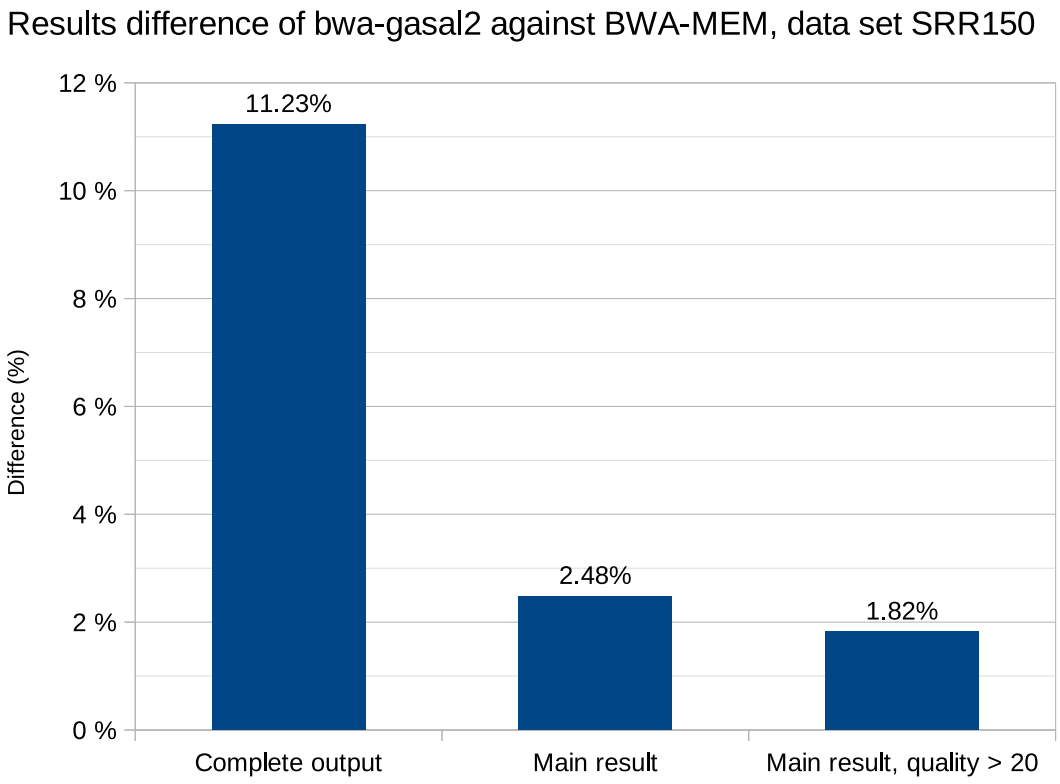


Figure 5.6: Difference in the result with the mainline version of BWA

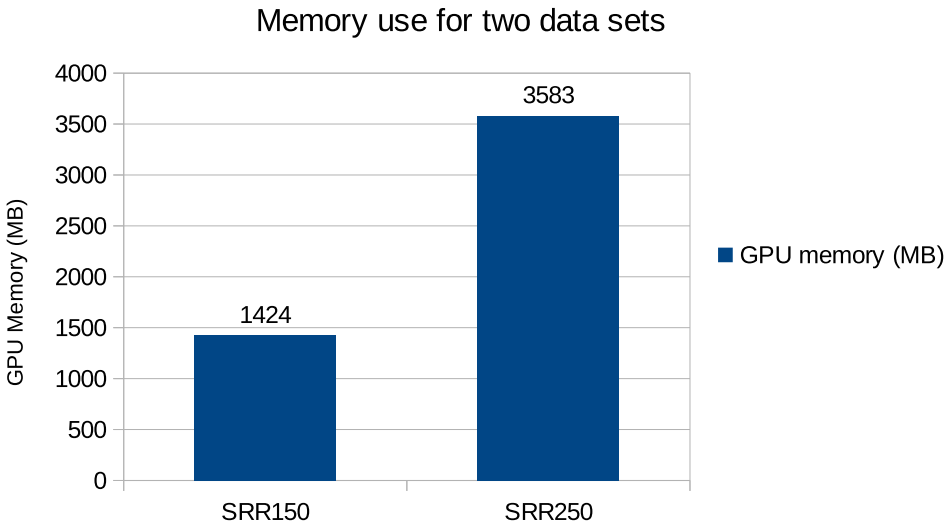


Figure 5.7: Memory use for our two data sets on our testing machine

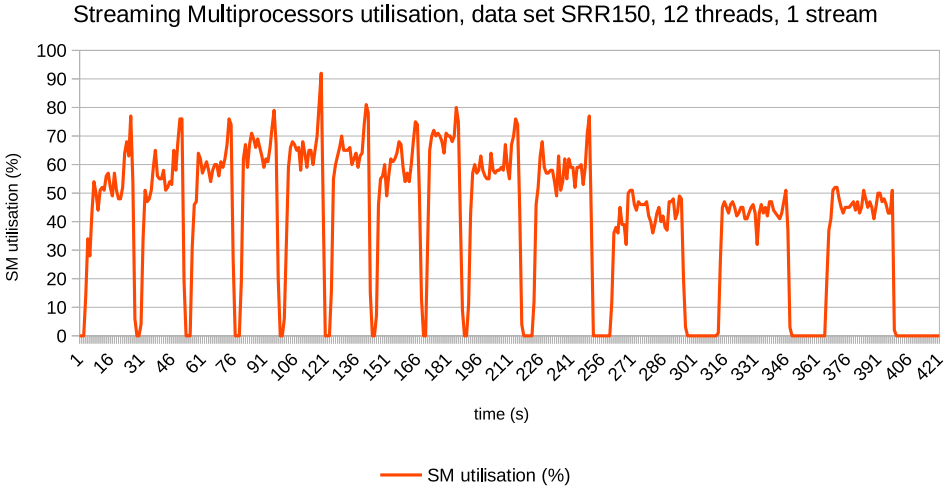


Figure 5.8: SM use for SRR150 with 1 stream on our testing machine

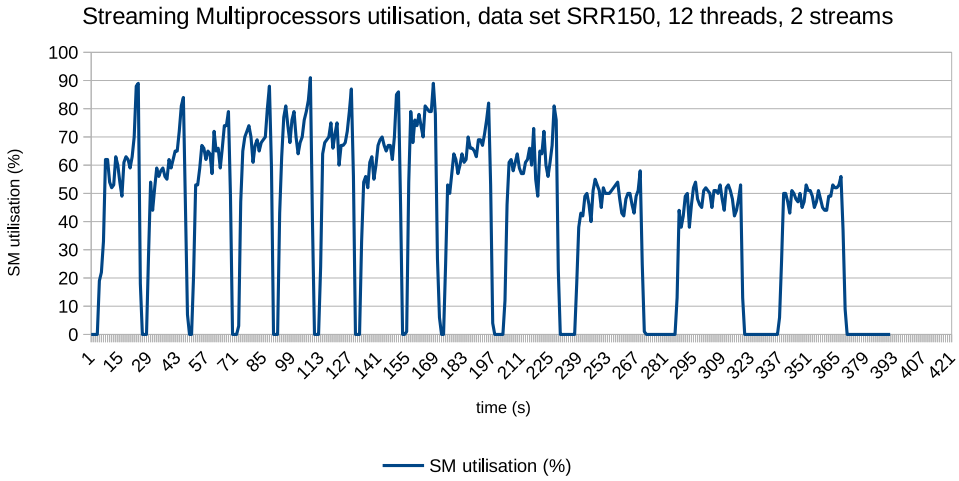


Figure 5.9: SM use for SRR150 with 2 streams on our testing machine

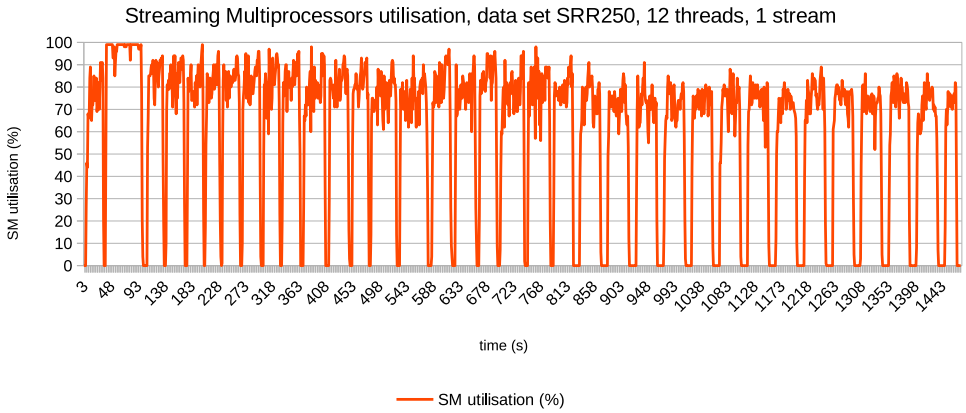


Figure 5.10: SM use for SRR250 with 1 stream on our testing machine

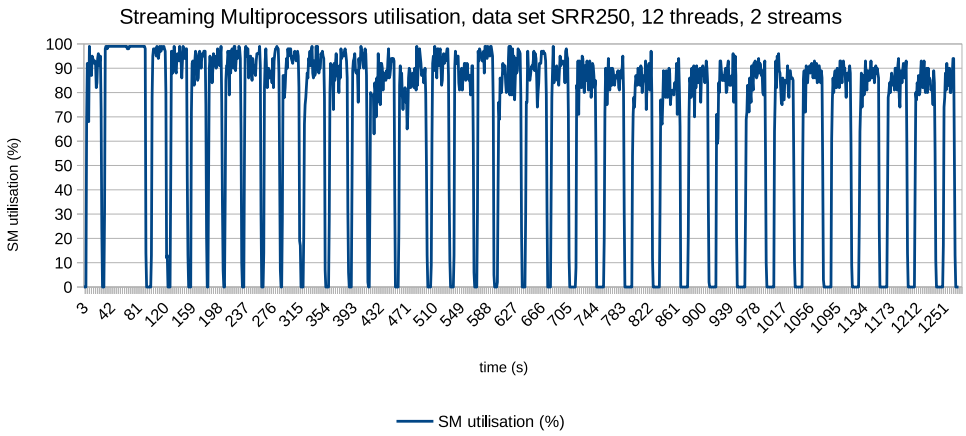


Figure 5.11: SM use for SRR250 with 2 streams on our testing machine

Conclusions and recommendations

6

In this chapter, we present the main conclusions of the thesis and reflect back on the research questions defined in the thesis. We also discuss a number of recommendations for future work.

6.1 Conclusion

This work presents how we addressed the challenge of accelerating time consuming genomics algorithms to make them more accessible for application in the field. The thesis specifically focuses on BWA-MEM as one of the most widely used DNA read mapping programs.

Below are the research questions we defined in the thesis and the answers we can provide.

- How can we accelerate DNA alignment in an already existing program?

We chose an industry standard mapper, BWA-MEM, and we accelerated one of its processing stages. The "seed-extension" phase can be run in an accelerated fashion on a dedicated hardware, the GPU. We integrated the library GASAL2 in BWA-MEM to realise the acceleration. Still, we could not directly compute seed-extension as done in the original BWA-MEM: the left and right parts were computed one after the other, and starting with the previous total score (so, first the seed score is used to compute the left part, then the score from the left alignment is used for the right part). Instead, we compute both parts at the same time only using the seed score as starting score.

To produce a working piece of software, we had to implement multiple improvements in GASAL2. We created a dedicated kernel for extension, copying BWA-MEM behaviour that follows the Smith-Waterman algorithm, but able to start with any given starting score. We also implemented an extensible memory structure for GASAL2 to adapt to an unknown number of alignment with unpredictable lengths for each of them. Finally, we instrumented both the new version bwa-gasal2 and the original BWA-MEM to be able to log their execution time and compare them.

We started from the demonstrator GASE-GASAL2 (based on BWA-MEM) that we ported to support C++ compilation before integrating the new GASAL2 version. It features templates for generic functions and a parameters class. We implemented all the aforementioned features to produce the current version of the software, **bwa-gasal2**.

- How much speed-up can we get from GPU acceleration?

We ran our solution on two data sets, with reads of 150 and 250 bases, named SRR150 and SRR250 respectively. Depending on the data, in BWA-MEM, extension

takes between 27% and 33% of the total time. This means that, depending on the situation, we can reach a theoretical 1.37× and 1.50× speed-up.

We saw that on our test machine, parallel execution alone can provide an already interesting speed-up. On 12 threads in SRR150, we reach 1.21× speed-up. But when enabling overlapped execution with two CUDA streams, we can get a 1.28× speed-up, which gets closer to the theoretical maximum of 1.37×. This is all the more visible with data set SRR250 where the speed-up soars from 1.12× to 1.28×. We obtained a very good acceleration, close to the theoretical maximum.

There are various reasons explaining the difference between achieved and maximum speedup. First, the CPU has to wait to get the final result of a given pack of sequences (it cannot start the next pack right away). Moreover, some overhead is introduced when filling the data structure in GASAL2 on the host side, before copying them on the device.

- How close can the results with a different computing method be to the original software?

Since the output is text-based, we compared them by logging them to text and using the `diff` UNIX utility. For each sequence, the main alignment is reported, along with secondary scores that are not always listed in the same order and can slightly differ because of the "seed-only" paradigm.

When comparing the complete results from the SRR150 data set, we found a 11.23% difference. This is a higher bound that includes non similar secondary score order. This information is usually not important in downstream analysis. This is why we also compared only using the main alignment and also discarding the low quality scores (lower than 20). In this last case, the difference drops to 1.82%, so the two outputs are very similar. We concluded that our solution produces acceptable results.

- How to ensure that the GPU resources are well used, while leaving more space if needed for future evolution?

A GPU is well used when its computing resources have a high utilisation. Although a single kernel execution from GASAL2 has a low utilisation (between 3% to 10%), instantiating multiple CPU threads to launch more kernels at the same time allows us to multiply this utilisation in a direct fashion. This way, we can reach 70% with peaks at 90% of utilisation, which is a good use of resources.

On the contrary, a sane use of GPU VRAM would be not to use more memory than what is necessary. Being frugal in memory can be useful if we want to align longer sequences, as these take a lot of space. Also, it can prevent out-of-memory errors if other parts were accelerated, or simply it could allow the program to run on a moderately-sized accelerator. To this extent, we used extensible memory structures to store the DNA strings, with a linked list structure with elements growing in size. This allows to allocate more memory whenever needed, and since the linked list is re-used, we try to keep the number of memory allocations to the minimum. With this, we could use around 20% of the total memory of our accelerator for data set SRR150, or around 30% for SRR250.

6.2 Future work

Seed-extension kernel can be further optimised. Any optimisation done at this level could have serious repercussions on the total execution time. For example, one could implement data packing on a 3-bit encoding, to save more memory and compute the dynamic programming matrix by tiles of 10×10 bases. It would be interesting to see whether this could bring a noticeable speed-up.

Originally, BWA-MEM is under GNU General Public License v3.0 and GASAL2 is licenced under Apache 2.0 License. Because of the licence choice of BWA-MEM, projects including it must follow the GPL v3.0, providing full source code disclosure. It follows that bwa-gasal2 will get the same licence to respect BWA-MEM original rights; this includes leaving the source open for anyone to see. It means that any volunteer can contribute or fork the project to enlarge it, provided they ship it with the same licence.

Bibliography

[1] W. The Free Encyclopedia. (2019). Desoxyribonucleic acid, [Online]. Available: https://en.wikipedia.org/wiki/DNA.

[2] G. R. Consortium. (2009). Genome reference consortium human build 37 (grch37) "hg19", [Online]. Available: https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.13/.

[3] (2019). Felis catus (domestic cat), [Online]. Available: https://www.ncbi.nlm.nih.gov/genome/78?genome_assembly_id=356698.

[4] (2019). Canis lupus familiaris (dog), [Online]. Available: https://www.ncbi.nlm.nih.gov/genome?term=canis%5C%20lupus%5C%20familiaris.

[5] B. Cornell. (). Dna structure, [Online]. Available: https://ib.bioninja.com.au/standard-level/topic-2-molecular-biology/26-structure-of-dna-and-rna/dna-structure.html.

[6] N. H. G. R. Institute. (2018). Specific genetic disorders, [Online]. Available: https://www.genome.gov/For-Patients-and-Families/Genetic-Disorders.

[7] N. Methods, *E pluribus unum*, 2010. DOI: 10.1038\%2Fnmeth0510-331.

[8] E. J. Houtgast, V.-M. Sima, K. Bertels, and Z. Al-Ars, "Hardware acceleration of bwa-mem genomic short read mapping for longer read lengths," *Computational Biology and Chemistry*, vol. 75, pp. 54–64, 2018, ISSN: 1476-9271. DOI: https://doi.org/10.1016/j.compbiolchem.2018.03.024. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1476927118301555.

[9] F. Sanger, S. Nicklen, and A. R. Coulson, "Dna sequencing with chain-terminating inhibitors," Dec. 1977. DOI: 10.1073/pnas.74.12.5463.

[10] H. Lee, J. Gurtowski, S. Yoo, M. Nattestad, S. Marcus, S. Goodwin, W. Richard McCombie, and M. C. Schatz, "Third-generation sequencing and the future of genomics," *bioRxiv*, 2016. DOI: 10.1101/048603. eprint: https://www.biorxiv.org/content/early/2016/04/13/048603.full.pdf. [Online]. Available: https://www.biorxiv.org/content/early/2016/04/13/048603.

[11] (2019). Nanopores - how it works, [Online]. Available: https://nanoporetech.com/how-it-works.

[12] (2019). Sequencing by synthesis (sbs) technology, [Online]. Available: https://www.illumina.com/science/technology/next-generation-sequencing/sequencing-technology.html.

[13] (2019). Sequencing-by-synthesis: Explaining the illumina sequencing technology, [Online]. Available: https://bitesizebio.com/13546/sequencing-by-synthesis-explaining-the-illumina-sequencing-technology/.

[14] T. Alasadi, *Introductory-chemistry*, Jul. 2016. [Online]. Available: https://www.researchgate.net/publication/305387708_introductory-chemistry.

[15] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," *Digital SRC Research Report 124*, 1994.

[16] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with bwa-mem," *ArXiv*, vol. 1303, Mar. 2013.

[17] W. The Free Encyclopedia. (2019). Fm-index, [Online]. Available: https://en.wikipedia.org/wiki/FM-index.

[18] H. Li, "Exploring single-sample snp and indel calling with whole-genome de novo assembly," *Bioinformatics (Oxford, England)*, vol. 28, no. 14, pp. 1838–1844, Jul. 2012, bts280[PII], ISSN: 1367-4811. DOI: 10.1093/bioinformatics/bts280. [Online]. Available: https://www.ncbi.nlm.nih.gov/pubmed/22569178.

[19] ——, (2013). Aligning sequence reads, clone sequences and assembly contig with bwa-mem.

[20] S. Aluru, *Handbook of Computational Molecular Biology (Chapman & All/Crc Computer and Information Science Series)*. Chapman & Hall/CRC, 2005, ISBN: 1584884061.

[21] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970, ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(70)90057-4. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022283670900574.

[22] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981, ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(81)90087-5. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022283681900875.

[23] D. Durand, *Computational genomics and molecular biology, fall 2015*, 2015.

[24] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990, ISSN: 0022-2836. DOI: https://doi.org/10.1016/S0022-2836(05)80360-2. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022283605803602.

[25] T. S. F. S. W. Group. (2019). Sam format documentation, [Online]. Available: https://samtools.github.io/hts-specs/SAMv1.pdf.

[26] N. Ahmed, V. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars, "Heterogeneous hardware/software acceleration of the bwa-mem dna alignment algorithm," Dec. 2015. DOI: 10.1109/ICCAD.2015.7372576.

[27] H. Mushtaq, N. Ahmed, and Z. Al-Ars, "Streaming distributed dna sequence alignment using apache spark," in *Proc. 17th annual IEEE International Conference on BioInformatics and BioEngineering*, Washington DC, USA, Oct. 2017.

[28] D. Li and M. Becchi, "Abstract: Multiple pairwise sequence alignments with the needleman-wunsch algorithm on gpu," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov. 2012, pp. 1471–1472. DOI: 10.1109/SC.Companion.2012.267.

[29] (2015). Nvbio: A library of reusable components designed by nvidia corporation to accelerate bioinformatics applications using cuda, [Online]. Available: http://nvlabs.github.io/nvbio/.

[30] N. Ahmed, H. Mushtaq, K. Bertels, and Z. Al-Ars, "Gpu accelerated api for alignment of genomics sequencing data," *IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2017)*, 2017.

[31] E. J. Houtgast, V. Sima, K. Bertels, and Z. AlArs, "An efficient gpu-accelerated implementation of genomic short read mapping with bwamem," *SIGARCH Comput. Archit. News*, vol. 44, no. 4, pp. 38–43, Jan. 2017, ISSN: 0163-5964. DOI: 10.1145/3039902.3039910. [Online]. Available: http://doi.acm.org/10.1145/3039902.3039910.

[32] E. J. Houtgast, V. Sima, G. Marchiori, K. Bertels, and Z. Al-Ars, "Power-efficiency analysis of accelerated bwa-mem implementations on heterogeneous computing platforms," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Nov. 2016, pp. 1–8. DOI: 10.1109/ReConFig.2016.7857181.

[33] W. The Free Encyclopedia. (2019). List of sequence alignment software, [Online]. Available: https://en.wikipedia.org/wiki/List_of_sequence_alignment_software.

[34] A. D"oring, D. Weese, T. Rausch, and K. Reinert, "SeqAn an efficient, generic C++ library for sequence analysis.," English, *BMC Bioinformatics*, vol. 9, no. 1, p. 11, 2008. DOI: 10.1186/1471-2105-9-11. [Online]. Available: http://www.biomedcentral.com/1471-2105/9/11.

[35] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome Biology*, vol. 10, no. 3, R25, 2009, ISSN: 1474-760X. DOI: 10.1186/gb-2009-10-3-r25. [Online]. Available: https://doi.org/10.1186/gb-2009-10-3-r25.

[36] N. Ahmed and J. Lévy. (2019). Gasal2: Extension of gasal allowing full cpu-gpu overlap, along with multiple improvements, [Online]. Available: https://github.com/nahmedraja/GASAL2.

[37] T. K. Group. (2019). Opencl - the open standard for parallel programming of heterogeneous systems, [Online]. Available: https://www.khronos.org/opencl/.

[38] NVIDIA. (2019). Cuda zone, [Online]. Available: https://developer.nvidia.com/cuda-zone.

[39] A. Bartezzaghi, M. Cremonesi, N. Parolini, and U. Perego, "An explicit dynamics gpu structural solver for thin shell finite elements," *Computers & Structures*, vol. 154, Jul. 2015. DOI: 10.1016/j.compstruc.2015.03.005.

[40] NVIDIA. (2012). Nvidia kepler architecture whitepaper, [Online]. Available: http://gec.di.uminho.pt/miei/cpd1718/AA/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[41] N. Steve Rennich. (2015). Cuda streams and concurrency, [Online]. Available: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.

[42] Y. Chen, J. Cong, J. Lei, and P. Wei, "A novel high-throughput acceleration engine for read alignment," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp. 199–202. DOI: 10.1109/FCCM.2015.27.

[43] (2013). Giab: Illumina srp011558 miseq srr473302-srr473304(srr949537), [Online]. Available: https://trace.ncbi.nlm.nih.gov/Traces/sra/?run=SRR949537.

[44] J. Lévy. (2019). Modified burrows-wheeler aligner using gasal2 for accelerated extension, [Online]. Available: https://github.com/j-levy/bwa-gasal2.

[45] Panic2k4 and T. Wikibook Community. (2015). C++ programming - template meta-programming, [Online]. Available: https://en.wikibooks.org/wiki/C%2B%2B_Programming/Templates/Template_Meta-Programming.

[46] N. Ahmed. (2018). Gase-gasal2, bwa-based aligner with demonstration of gasal2 acceleration, [Online]. Available: https://github.com/nahmedraja/gase-gasal2.

[47] N. Ahmed, K. Bertels, and Z. Al-Ars, "A comparison of seed-and-extend techniques in modern dna read alignment algorithms," Dec. 2016, pp. 1421–1428. DOI: 10.1109/BIBM.2016.7822731.

[48] H. Li. (2017). Burrows-wheeler aligner, [Online]. Available: https://github.com/lh3/bwa.

[49] J. Lévy. (2019). Modified burrows-wheeler aligner for seed-only alignment and timed execution, [Online]. Available: https://github.com/j-levy/bwa.

[50] F. S. Foundation. (2016). Gnu diff, [Online]. Available: https://www.gnu.org/software/diffutils/.

[51] Intel. (2012). Intel® xeon® processor e5-2620, [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/64594/intel-xeon-processor-e5-2620-15m-cache-2-00-ghz-7-20-gt-s-intel-qpi.html.

[52] NVIDIA. (2012). Nvidia tesla k40 specifications, [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/TeslaK80-datasheet.pdf.

[53] (2019). Illumina whole genome shotgun sequencing of genomic dna paired-end library 'solexa-127365' containing sample 'gm12878', [Online]. Available: https://trace.ncbi.nlm.nih.gov/Traces/sra/?run=SRR835433.