Algorithms for the mapping of genome sequences in GPGPU

Author: David Seide(*dasei@upv.es*)
Codirector: José Salavert Torres(*josator@fiv.upv.es*)
Director: Ignacio Blanquer Espert(*iblanque@dsic.upv.es*)

July 2012

Contents

1 Introduction

Since the advent of NGS (New/Next Sequencing Algorithms) at the end of the 2000s decade the speed at which new genomes were sequenced has been increased several orders of magnitudes. In addition, cheaper methods have risen the amount of genomic sequence data exponentially [11]. Unfortunately there is still a drawback. The obtained sequences (reads) produced with these techniques are much shorter than before (usually called short-reads), this fact encourages the need for new processing algorithms to cope with these biological data avalanches. To be more precise, NGS produce reads with an average length of up to 100 nt (nucleotides), whereas conventional methods produce reads with an average length of 1000 nt. However, the number of reads produced in NGS have the order of hundreds of millions, meanwhile conventional methods only reach magnitudes of a few hundred thousand.

The most common objective in this field consists in mapping newly obtained DNA information onto a full reference DNA of an already sequenced genome to find, for example, indication of homologies.

Nowadays, there exist several tools (BLAST, BWA, Bowtie, Soap, etc.) that allow biologists to perform short-read alignments. Apart from these, the research group for grid and high performance computations (GRyCAP) from the institute I3M of the Polytechnic University of Valencia (UPV) has also come up with an approach [18] that supports the exact alignment of short-read nucleotide sequences and takes advantage of the GPGPU parallel architecture.

The objective of this project is to extend this implementation, so that it allows one error between the reference and the mappings.

We present in this work an implementation on GPU of a BWT inexact aligner, that allows 1 error. The program guarantees to find all possible results in GPU and our implementation works about 7 times faster than the same algorithm in CPU.

Section 2 defines the objectives of the development presented in this final year project. In section 3 we overview current solutions in the field of work of sequence alignment. Section 4 gives an insight of the equipment that was employed during this project. Section 5 explains what the BWT is and how the necessary FM-Index is created. Section 6 describes how we can make use of the BWT for sequence alignment and how we implemented it. Section 7 depicts the results of several experiments that show the potential of the GPU implementation. Section 8 contains the conclusions reached with this work.

2 Objectives

This project focuses on using GPGPUs for solving the inexact alignment of short-reads with respect to a reference indexed using the Burrows-Wheeler Transform. To be more specific we dealt with a solution of an alignment that allows up to one error. It follows the work presented in the article [18].

This main objective implies several sub-objectives that have guided the work.

- Get familiar with the CUDA environment.
- Adapt the algorithm allowing one error to be efficient on GPU.
- Find an adequate data structure to store the results for each read and test it on the CPU version before porting it to GPU.
- Measure and compare the gained speed-up from the GPU to the CPU version.

3 State of the art in sequence alignment

The alignment of DNA sequences consists basically in comparing tons of query strings stored in an array W to a reference string X and to process the results in the desired form. In our case the average length of the query strings ($|W| \approx 100$) is a fraction of $|X| \approx 3 \times 10^9$.

To give you an overview about the different result types that can occur, let X be AGGAGC and W accordingly to each line of Table 1. As you can see, we consider four types of results that can occur; exact match, mismatch, insertion and deletion. The already existing program mentioned above allows to align only exact mapping of reads. The target of this project is to create a gpu version of this search algorithm, that considers one of the four types of results during the alignment.

Result types	W	Found String	Description
Match	AGC	AGC	W appears in X
Mismatch	GCA	GGA	W does not appear exactly in X but with one symbol changed for another.
Insertion	AGA	AGGA	W does not appear exactly in X but with one symbol inserted after the first or before the last symbol.
Deletion	AGC	GC	W does not appear exactly in X but with one symbol being deleted.

Table 1: Examples of different result types when X=AGGAGC and W accordingly to each line of the table.

3.1 Format

3.1.1 FASTA

FASTA [13] [17] is a program used for the alignment of DNA sequences and proteins. The name FASTA is derived from FAST-ALL because it allows searches with any alphabet. It is an extension from FAST-P for proteins and FAST-N for nucleotides. The principal advantages of the FASTA package are its variety of possible input types (different alphabets) and the statistical possibilities that it offers by default. This partly all-in-one solution is easy to use even for a biologist with little computer engineering background.

3.2 Long reads

3.2.1 BLAST

BLAST [3] [4] stands for Basic Local Alignment Search Tool and is a program for comparing a query of primary sequence information, such as amino-acid or nucleotide sequences, against a database of queries. It enables scientists to find the most resembling sequences in the database, listed by their grade of similarity to the query. Because *BLAST* is a heuristic adaption of the Smith-Waterman [19] [7] algorithm, it is significantly faster than a non-heuristic

approach, but can never guarantee to find neither all solutions, nor the best one. It is the most used and quoted algorithm in bioinformatics and can be accessed freely through the NCBI (National Center of Biotechnology Information) server.

3.3 Short reads

3.3.1 BWA

The name of this algorithm is an abbreviation of the term Burrows-Wheeler Alignment [10]. and includes a Burrows-Wheeler Transformation within the process. Here it is used to pre-process the reference string and make the real search more simple and less time-consuming afterwards. The BWA allows an unlimited number (defined by parameter) of errors (mismatch, insertion, deletion), though its time comsumption will also increase significantly when incrementing the number of allowed errors. The original publication presents a CPU solution and states that it is especially efficient for the alignment of short-reads against a large reference. It is an interesting approach because while being fast it does not lose its reliability for the fact of not being heuristic by any means in comparison to BLAST.

3.3.2 Bowtie

Bowtie [8] is a rapid short-read aligner for DNA sequences that also makes use of the BWT, it is open-source software and available under [1]. In the publication article the authors state that it has a maximum memory footprint of 1.3 GB during execution time, while aligning more than 25 million reads per CPU hour. This results are named in context of the human genome serving as the reference sequence. The algorithm makes use of a "novel quality-aware backtracking algorithm" that limits the backward steps in an intelligent way to speed up the algorithm, although excluding possible possible results with this technique. The creators of Bowtie furthermore introduced a multithreading CPU version which shows significant speed ups compared to the sequential version.

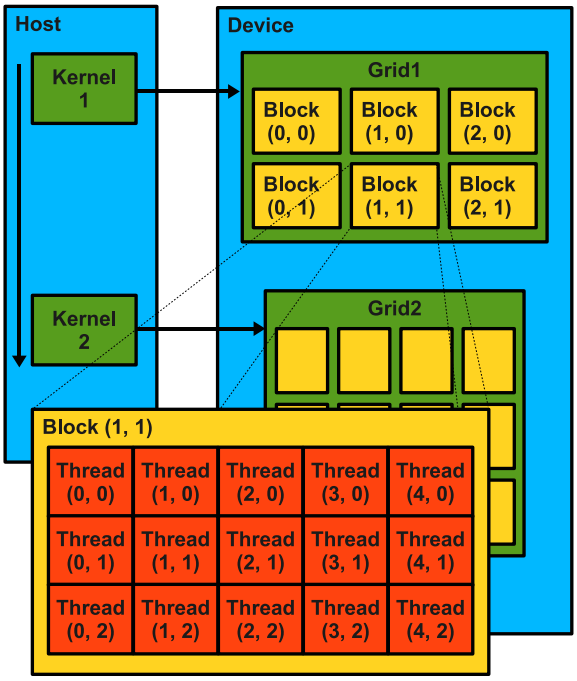
3.3.3 SOAP

SOAP stands for Short Oligonucleotide Alignment Package [11], it is available under [2] as open-source. There are different implementations of the SOAP algorithm. For example SOAP2 [12] is a multithreading CPU implementation that allows "either a certain number of mismatches or one continuous gap for aligning a read onto the reference sequence". SOAP3 [14] however is a GPU-based software that allows up to 4 mismatches and implements a dynamic programming approach.

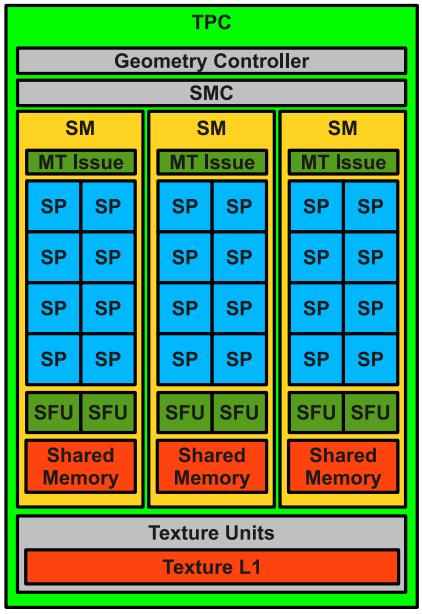
4 The platform of choice - CUDA

In this section a general overview of the NVIDIA CUDA technology [16] is given. CUDA stands for Compute Unified Device Architecture and offers general-purpose computing on general purpose graphics processing units (GPGPU), the said technique is available on NVIDIA graphics cards exclusively. CUDA extends the GPUs field of application from only processing and displaying graphics to more general tasks. To be more precise, such a graphics card can serve for highly intensive computations like compression of audio or video, acceleration of mathematical algorithms or rendering computer graphics.

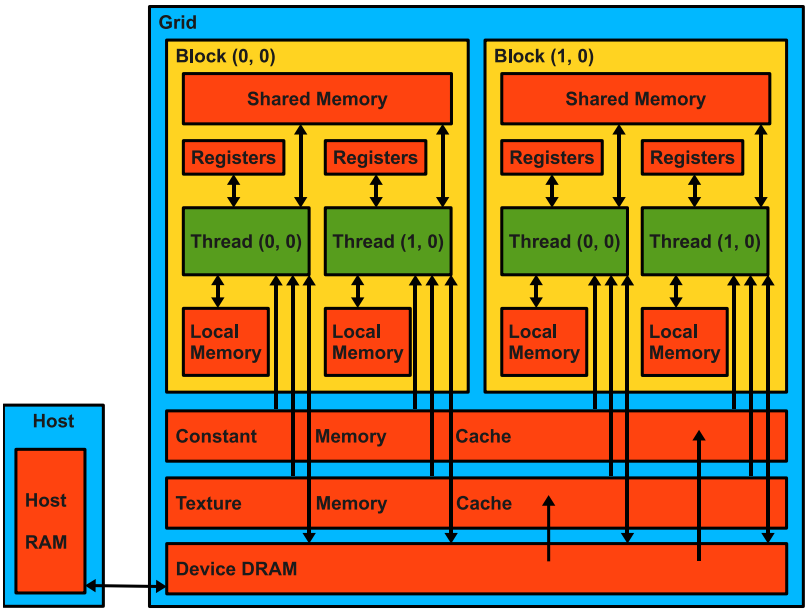
The structure and methodology of CUDA and how programs can be developed with CUDA are explained in this section. The main idea is that the highly computing intensive parts of a program, that can be executed in parallel, are outsourced to the graphics card using a SIMD-like architecture (Single Instruction Multiple Data). Such an outsourced function on the GPU is called a kernel. A kernel will be executed by up to thousands of threads at the same time, using intelligent scheduling algorithms. These threads are extremely lightweight, the context change produces practically no overhead and the graphics card needs hundreds of threads to run at full capacity. The graphics card has its own memory of a few gigabytes and as a whole can be seen as a very potent and intelligent coprocessor.



(a) A kernel is a grid of blocks of threads, that executes the same instructions over possibly different data.



(b) One of the ten Texture/Processor Clusters that form part of a GeForce GTX280.



(c) Correlations of the memory model of a CUDA graphics card.

Figure 1

4.1 Architecture

In this project a CUDA graphics card of the second generation series was employed. Cards of this series still have the ability to output images to a display and are not exclusively designed for high-performance computing like the Tesla series for example.

These graphics cards in general consist of Streaming Multiprocessors (SM) that are spread out on Texture/Processor Clusters (TPC). Figure 1b shows a single TPC of a GeForce GTX280. Each TPC consists of various SMs, that share the same bus to access the global memory and caches (Constant and texture cache). A number of Streaming Processors (SP) comprises each SM. All SPs inside of one SM have access to the same shared memory.

A SP in particular is a single processing core, that has ALUs and FPUs. It does not have any cache and therefore it is only effective when processing the same piece of data intensively. Each SM also includes Special Function Units (SFU). The instructions for the SPs and SFUs within one SM are dispatched by the MT Issue.

One of the reasons, why this architecture can be so powerful, lies in the scheduling algorithm that governs the threads. The thread scheduler intents to execute groups of 32 threads (called warps) with the same instructions over different data. The basic idea is to execute warps, which are prepared to continue immediatley and have no pending read/write operations.

So the order in which warps are executed is unknown while developing the code. It is mandatory to be able to guarantee that for example a memory copy operation has finished, before another warp accesses the same data. This can be achieved with a call to $\_\_syncthreads()$. This command causes the thread scheduler to pause every thread at this line of code, until every thread has reached it. So if placed after a copy operation, the code developer can guarantee, that the memory access in the next line will not be performed until the previous copy operation was finished by all threads.

The above mentioned thread scheduler allows to take profit of the cards full computing power capacity, provided that there are enough threads accessible to the scheduler, so that there is enough chance to have a warp ready to execute in every moment. Various warps are managed (creation, scheduling, execution) by one SM. The belonging threads start with the same instruction code, but can follow different branches during execution due to conditional statements in the code. Given the case of threads within a warp following different branches, they are executed sequential, however different warps can

follow different branches in parallel. Therefore using conditional statements in the kernel code is a possible performance killer, forcing the code to be executed sequentially in a non-optimal case. Following the term SIMD, NVIDIA calls the just described functionality "Single-Instruction, Multiple-Thread" (SIMT).

4.2 Memory layout

A kernel is organized as a grid of blocks of threads (See figure 1a). Table 2 presents a comparison of the different memory types that can be found on a CUDA graphics card. Each thread has exclusive access to its registers and local memory. The hardware registers are extremely fast but small, whereas the bigger local memory is also fast, but limited in size. Threads within a block can cooperate via shared memory, which is fast and therefore the key to fast parallel processing with CUDA. Threads of different blocks can access the global memory which is used to transfer data between CPU and GPU. The global memory classes are the slowest. Both CPU and GPU have direct read and write access to the global memory, whereas the constant cache and the texture cache can only be read by the GPU. These caches are through transmissions to the global memory. To read (GPU side) from the constant cache shows the same latency as from the hardware registers, provided that no page faults occur. The texture cache furthermore is optimized for 2D spatial read access GPU wise.

Memory	**Type**	**Read**	**Write**	**Access time**
Register	Hardware	One thread	One thread	1 Cycle
Local	Hardware	One thread	One thread	1 Cycle
Shared	Hardware	One block	One block	1 Cycle
Global	DRAM	CPU + GPU	CPU + GPU	Slow
Constant	Cache	CPU + GPU	CPU	Can vary (cache)
Texture	Cache	CPU + GPU	CPU	Can vary (cache)

Table 2: Comparison of the different memory models of a GPU

4.3 Programming a CUDA GPU

Programming in CUDA is much simpler compared to the hardware background a graphics card underlies. CUDA is available for different programming languages such as C, C++, JAVA, Python, etc... This project was realized in C. Regardless of what language is used, the CUDA framework adds functionality necessary to manage the GPU, perform memory transfers between CPU and GPU and to create and execute kernels on the device. However programming efficient algorithms is the most challenging part. Benefiting from the memory access locality requires a deep knowledge of the hardware, the CUDA environment and the algorithm itself.

Let us take a look at a small code example written in C. In the first listing, inside the main function we declare an array "a" with 16 values and an integer "b". These variables are passed to the *increment_cpu()* function which is a routine that adds the value of "b" to every value of the array "a". This operation has a O(n) cost, where n is the number of elements in "a".

Listing 1: CPU Increment

```
1  void increment_cpu(int *a, int b, int N)
2  {
3          for(int idx = 0; idx < N; idx++)
4                  a[idx] = a[idx] + b;
5  }
6
7  int main()
8  {
9          int N = 16;
10         int *a = filledIntArray(N);
11         int b = 1;
12
13         increment_cpu(a, b, N);
14 }
```

The second listing shows a CUDA program, lines 1-5 are device code and lines 7-23 are host code. In CUDA, Device and host refer to the place where the code will be executed (The graphics card device or the host computer respectively). CUDA device code stands out from plain C code because of its precedent keyword "__global__". In this example the kernel instructions would be executed by 16 threads simultaneously (4 blocks of 4 threads each).

The following code calculates the global array index for each thread. Grids

Listing 2: GPU Increment

```
1  __global__ increment_gpu(int *a, int b, int N)
2  {
3          int idx = blockIdx.x * blockDim.x + threadIdx.x;
4          a[idx] = a[idx] + b;
5  }
6
7  int main()
8  {
9          int N = 16;
10         int *a = filledIntArray(N);
11         int b = 1;
12
13         float *d_a;
14         cudaMalloc((void**)&d_a, N);
15
16         cudaMemcpy(d_a, h_a, N, cudaMemcpyHostToDevice);
17
18         dim3 dimBlock(4);
19         dim3 dimGrid( ceil( N / (float) 4) );
20         increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
21
22         cudaMemcpy(h_a, d_a, N, cudaMemcpyDeviceToHost);
23 }
```

and blocks can have up to three dimensions whose boundaries (limited by hardware) can be checked in the deviceQuery information of our card. The "cuda.h" header file also offers functions to verify this information during runtime.

The desired dimensions are set by the programmer before launching a kernel and they are included in the arguments for the kernel call.

dimGrid(x,y,z)	Number of blocks in every dimension.
dimBlock(x,y,z)	Number of threads per block in every dimension.

In the example we only set the first member of these variables (x). To check the grid-position of a thread during runtime, we can consult several variables. In our example, the array has only one dimension, so that the members y and z of the following variables are not considered.

blockIdx.x	The ID of the block, the current thread belongs to.
blockDim.x	The number of threads per block.
threadIdx.x	The local thread index.

The calculation in line 3 is a very common pattern and allows the programmer to map from the local index threadIdx.x to the global index. In line 4 each thread modifies its element of the array "a" and returns to main. Every thread does the same (no conditional statements and no diverging branches), so this code would be executed by the 16 threads simultaneously.

In the main function we can see some of the drawbacks of CUDA, such as less code legibility and possible performance trade-offs, when applying algorithms not best-fitting the CUDA model. The first 3 lines are equal to the CPU version, but in line 14 we allocate memory on the device to copy our data to. In lines 18 and 19 we set the grid dimensions. Finally in line 20 the kernel is executed over 4 blocks with 4 threads each. In line 22, after having successfully executed the device code, we copy the results to the host memory. Now we could check if the results are the same as in the CPU version.

This is a very simple example on how to use CUDA code, although it does not speed up the execution time compared to the sequential CPU version. This program has an overhead for

- copying the data to the device,
- executing the kernel and
- copying the data back to the host

In total, the objective is to end up with a program that takes less time than the CPU version. This can be accomplished by significantly extending the amount of calculations in the kernel and limiting the number of copy operations between host and device and vice versa.

5 The Burrows-Wheeler Transformation

BWT stands for Burrows-Wheeler Transform [5] and has had its first commercial success in the pre-processing parts of compression algorithms like bzip2. It was developed by Michael Burrows and David Wheeler in the year 1994. The transformation takes a block of input data and creates a block of output data of the same size. The output represents a permutation of the input. This means that only the order of the symbols but not its distribution will be modified. The new obtained output sequence serves for Run-Length-Encoding (RLE) compression algorithms because the same symbols now tend to occur in succession [15]. The transformed data can be converted back to its genuine version. In bioinformatics one application of the BWT is to create indexes of big reference genomes, accelerating the alignment of short genome sequences [10].

5.1 Necessary data structures

Let $\Sigma = A, C, G, T$ be an alphabet. Let \$ be a symbol not present in Σ that is lexicographically smaller than all symbols in Σ. A string $X = a_0 a_1 \dots a_{n-1}$ always ends with the symbol $\$ = a_{n-1}$, this being the only incidence of \$ in X and has length of $|X| =$ n. Furthermore we define the following expressions:

$X[i] = a_i$	Refer to the i-th symbol of X
$X[i, j] = a_i \dots a_j$	Describe a specified subarray
$X_i = a_i \dots a_{n-1}$	Refer to a suffix from X, starting at the specified position i

Now we create a matrix containing all possible suffixes of the reference string X, by simply rotating its symbols and constructing the matrix M using the string $X =$ "$AGGAGC\$$" results in:

$$M = \begin{pmatrix} A & G & G & A & G & C & \$ \\ G & G & A & G & C & \$ & A \\ G & A & G & C & \$ & A & G \\ A & G & C & \$ & A & G & G \\ G & C & \$ & A & G & G & A \\ C & \$ & A & G & G & A & G \\ \$ & A & G & G & A & G & C \end{pmatrix} \xrightarrow[rows]{sort} M' = \begin{pmatrix} \$ & A & G & G & A & G & \mathbf{C} \\ A & G & C & \$ & A & G & \mathbf{G} \\ A & G & G & A & G & C & \mathbf{\$} \\ C & \$ & A & G & G & A & \mathbf{G} \\ G & A & G & C & \$ & A & \mathbf{G} \\ G & C & \$ & A & G & G & \mathbf{A} \\ G & G & A & G & C & \$ & \mathbf{A} \end{pmatrix}$$

The second step of the BWT is sorting the rows alphabetically obtaining M', now let vector B contain the last column of M'

$$B = \begin{pmatrix} C & G & \$ & G & G & A & A \end{pmatrix}$$

and let vector S contain for every row in M' its original position in M.

$$S = \begin{pmatrix} 6 & 3 & 0 & 5 & 2 & 4 & 1 \end{pmatrix}$$

The BWT is now defined as follows [10]:

$$B[i] = \begin{cases} \$ & \text{if S[i]} = 0 \\ \text{X[S[i]-1]} & \text{if S[i]} \neq 0 \end{cases}$$

Therefore vector S allows us to locate the exact original position in the reference string X. B is the BWT transform of X. Take for example $B[1] = C$, which leads to $S[1] - 1 = 5$, X[5] $\stackrel{!}{=} C$.

Vector B will be stored in matrix O and vector C. This is called FM-Index [6] and will serve us later to run the actual search algorithm.

Each position in vector C ($|C| = |\Sigma|$) stands for one symbol from the alphabet Σ, ordered lexicographically. The actual value in every position states how many symbols in X exist (excluding \$), that are lexicographically smaller than the symbol in that position. Following our example, the described vector results in:

$$\begin{array}{cccccccc} & \text{A} & \text{C} & \text{G} & \text{T} & \\ \text{C} = (& 0 & 2 & 3 & 6 &) \end{array}$$

C shows us, that X holds 3 symbols lexicographically smaller than 'G', which are two 'A' and one 'C'. Generally speaking, $C[a]$ contains the number of symbols present in $X[0 : n-2]$ (excluding \$) that are lexicographically smaller than $a \in A$.

$O[a, i]$ however is the number of occurences of symbol $a \in A$ in $B[0 : i]$. Following our example, matrix O contains:

$$O = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 2 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & \mathbf{2} & 3 & 3 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} A \\ C \\ \mathbf{G} \\ T \end{matrix}$$

As an example, $O['G', 3] = 2$ means that 'G' appears twice in the interval $B[0 : 3] =$ "**CG\$G**GAA".

5.2 Optional velocity improvement

Depending on the length of X, the process of sorting M' and therefore the whole BWT X, can be computationally intensive. As we are sorting a suffix matrix the complexity of sorting the rows of M can be reduced from $O(n^2 log * n)$ to $O(n(log * n)^2)$, as stated in [9]. Also, a ternary quicksort with a random pivot is necessary to avoid inefficient partitioning.

6 Implementation

6.1 Theoretic approach to the search algorithm

Once the FM-Index is calculated, the alignment can be performed. This solution [18] is an adaptation of the algorithm present in [10]. In this section I will explain the theoretical approaches to the exact search algorithm.

If string W appears multiple times in X, the position of these occurrences will also be present in an continuous interval in vector S. This is because all rows of M', that start with W are grouped together (they were sorted beforehand). Based on this observation from [10] we define the following:

$$\lfloor R(W) \rfloor = \min(k : W \text{ is a prefix in } M'(k)) \tag{6.1}$$

$$\lceil R(W) \rceil = \max(k : W \text{ is a prefix in } M'(k)) \tag{6.2}$$

So $\lfloor R(W) \rfloor$ and $\lceil R(W) \rceil$ describe the first and the last occurrence (e.g. lower and upper limit) of W in X. The hereby obtained interval $[\lfloor R(W) \rfloor, \lceil R(W) \rceil]$ is called SA interval and includes all the positions of W in X. Having this information we can use vector S to look up the original positions of W in X.

Being for example $W =$ "AG", we obtain the following:

$$\begin{aligned} \lfloor R(W) \rfloor &= 2 \\ \lceil R(W) \rceil &= 3 \end{aligned}$$

Looking up the SA interval $S[2:3]$ gives us 0 and 3, which are the two starting positions of "AG" in X.

$$S = \{6 \quad \mathbf{0} \quad \mathbf{3} \quad 5 \quad 2 \quad 4 \quad 1\}$$
$$X = \{\mathbf{A} \quad \mathbf{G} \quad G \quad \mathbf{A} \quad \mathbf{G} \quad C\}$$

As we can obtain the original string positions sequence from vector S, the exact string alignment process consists in finding an interval in vector S. It can be

determined in $O(|W|)$ time. The exact matching problem, gives us exactly one S interval, whereas the inexact matching problem can return more than one interval (Described in section 6.5).

We can define $\lfloor R(aW) \rfloor$ and $\lceil R(aW) \rceil$ using the FM-Index structures, vector C and matrix O.

$$\lfloor R(aW) \rfloor = C(a) + O(a, \lfloor R(aW) \rfloor - 1) + 1 \tag{6.3}$$

$$\lceil R(aW) \rceil = C(a) + O(a, \lceil R(aW) \rceil \tag{6.4}$$

Also we define the following properties.

$$\lfloor R(aW) \rfloor \leq \lceil R(aW) \rceil \text{ if } aW \in X \tag{6.5}$$

$$\lfloor R(\varepsilon) \rfloor = 0 \tag{6.6}$$

$$\lfloor R(\varepsilon) \rfloor = |X| - 1 \tag{6.7}$$

These formulas offer the possibility to test whether W is a substring of X and if so, how many occurrences of W exist. This is achieved by calculating the SA interval via (6.3) and (6.4). If the resulting SA interval-length is bigger 0 (6.5), we can conclude that $W \in X$. The underlying process is not obvious and the formulas are recursive, so we will explain them in an example. Matrix O has an additional column which refers to -1 value, because (6.3) and (6.4) can return -1 as a result.

Assumed data structures:

X = "$AGGAGC$"

W = "AGC"

S = (6 3 0 5 2 4 1)

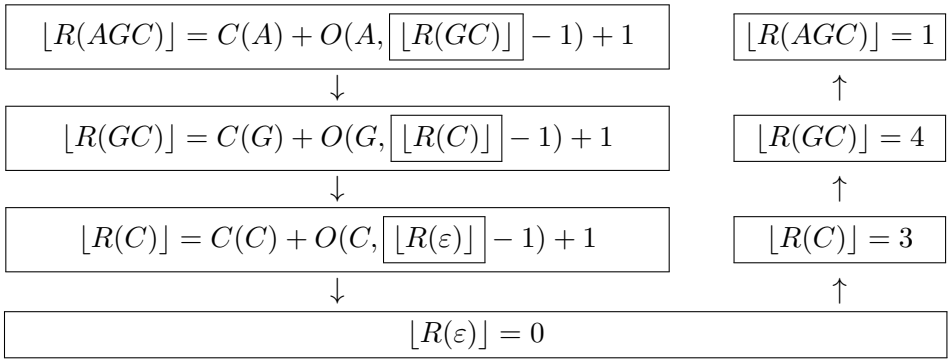
$$C = \begin{array}{cccc} A & C & G & T \\ 0 & 2 & 3 & 6 \end{array}$$

$$O = \begin{array}{c} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 2 & 3 & 3 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \\ -1 \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \end{array} \begin{array}{c} A \\ C \\ G \\ T \end{array}$$

Now we want to determine $\lfloor R(W) \rfloor$ and $\lceil R(W) \rceil$, to verify $W \in X$ and $W = X[3:5]$.

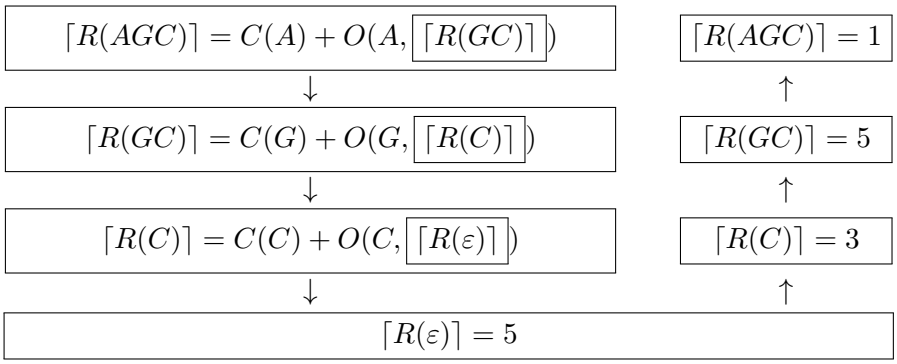
By calling $\lfloor R(W) \rfloor$, we get the following execution flow with $|W|$ levels of recursion, the small arrows between the boxes point towards the direction of the application flow.

Detailed recursion steps for lower limit:



Searching for the upper limit is analogous.

Detailed recursion steps for upper limit:



The obtained SA interval is $S[1:1]$ and $S[1]$ gives 3, consequently $X[3]$ is the starting position of W.

6.2 Recursive exact search algorithm

The pseudo-code presented in this subsection makes use of the theoretical approach introduced beforehand although the formulas vary a little. The C language does not allow negative array indexes like the first column of matrix O in the above example. Because of that, we simply let the indexation of O begin with 0 and delete the decrementing of 1 in (6.3), while incrementing the index by 1 on the very same spot in (6.4). With these changes we achieve the same result, while making the code more corresponding to the C language.

Algorithm 1 Recursive exact search algorithm

1: **function** EXACT_SEARCH(W, C, O, i, k, l)
2: **if** $i < 0$ **then return** $[k, l]$
3: $k \leftarrow C(W[i]) + O(W[i], k) + 1$
4: $l \leftarrow C(W[i]) + O(W[i], l + 1)$
5: **return** exact_search($W, C, O, i - 1, k, l$)
6: **end function**

The parameters of the function described in algorithm 1 are:

Parameter	Description
W	The search string (e.g. read)
C	Vector C
O	Matrix O with the shifted index
k	$k = 0 \rightarrow$ Start value of $\lfloor R \rfloor$
l	$l = \lvert B \rvert - 1 \rightarrow$ Start value of $\lceil R \rceil$

The algorithm returns a unique interval $[k, l]$ which represents the final values of $\lfloor R \rfloor$ and $\lceil R \rceil$ respectively.

6.3 Iterative exact search algorithm

The recursive algorithm, described in section 6.2, bears some problems when executed on a GPGPU. The recursion stack is stored in the global memory of the graphics card, whose access times are very slow. Also the actual size of the recursion stack is unknown until the whole algorithm has finished its execution and therefore constitutes a lack of control. Finally the performance of iterative algorithms is known to be better than its recursive versions.

These reasons make the implementation of recursive algorithms in CUDA ineffective. In order to increase performance, the algorithm was converted into an iterative version (See algorithm 2), leaving the same input parameters and return values as in the recursive version.

Algorithm 2 Iterative exact search algorithm

```
1: function EXACT_ITERATIVE_SEARCH(W, C, O, k, l)
2:     k2 ← k
3:     l2 ← l
4:     for i ← |W| − 1 ... 0 do
5:         k2 ← C(W[i]) + O(W[i], k2) + 1
6:         l2 ← C(W[i]) + O(W[i], l2 + 1)
7:         if k2 > l2 then return ∅
8:     end for
9:     return [k2, l2]
10: end function
```

6.4 Complete exact search algorithm

In this section we will explain how the actual alignment of reads is done sequentially and in parallel. Both approaches will use algorithm 2 to process a single read.

The sequential CPU version works as follows. The main idea is to firstly load BWT data structures of the reference string into the system memory then to repeat a loop containing the following steps:

1. Obtain a read from the queries file.
2. Align the read with the reference, using algorithm 2
3. Write the result to disk

This loop is repeated over all reads one by one, until there are no more reads left.

The parallel GPU version of the program works differently. Instead of only processing one read at a time, the kernel processes blocks of reads simultaneously. To give an idea of the order of magnitude, in this program we perform alignments of over 256000 reads at a time.

First of all, the FM-Index is loaded into the graphics card memory. Secondly this loop is executed:

1. Obtain a block of reads from the queries file and copy it to the GPU memory.
2. Launch the exact search kernel over the reads (Figure 2).
3. Copy the results back to the host computer and write them to disk.

This loop is also repeated until there are no more reads left. Take for example a number of 2000000 reads and a data-block size of 256000, this will result in 8 repetitions of the main loop ($2000000/256000 = 7.8$). The kernel algorithm flow is illustrated in figure 2.

First of all we calculate the global index of all threads to assign one read to each of them. then the first 4 threads of each block (32 threads) copy vector C to the their shared memory. The size of vector C is equal to the alphabet length and can therefore be stored shared memory, matrix O however is too big and resides in global memory. The array W contains all reads and nW the length of each. Subsequently algorithm 2 can be executed in each thread respectively and if present, the result is saved.

The pseudo-code and more details are explained in article [18], section 6.5.

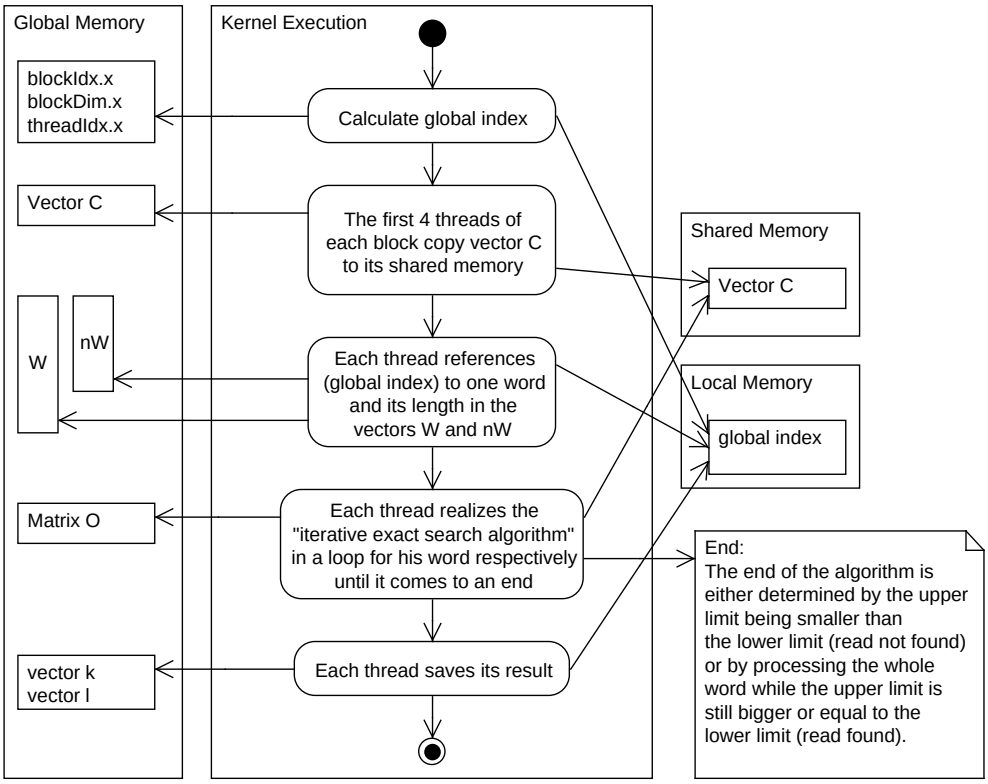


Figure 2: Execution flow of the exact search kernel.

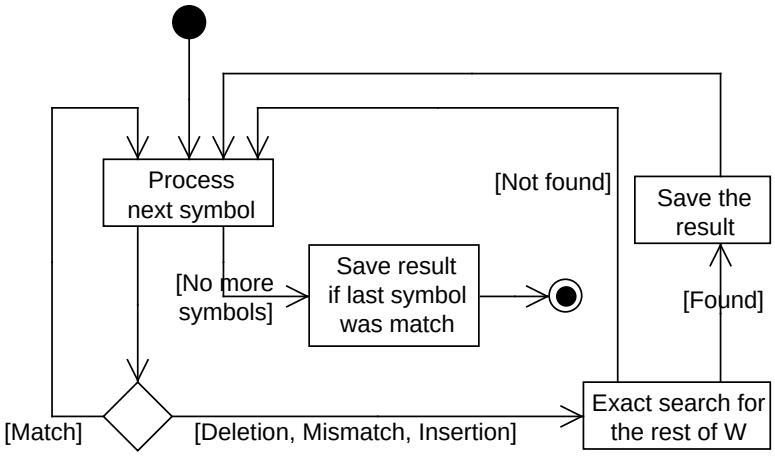


Figure 3: Execution flow of the inexact search kernel.

6.5 Complete search algorithm allowing up to 1 error

Table 3: Parameters for algorithm 3

Parameter	Description
W	An array of search strings (e.g. reads)
nW	The length of each read
C	Vector C
O	Matrix O with the shifted index
k_ini	Start value of $\lfloor R \rfloor$(default: 0)
l_ini	Start value of $\lceil R \rceil$(default: $\|B\| - 1$)
$results$	Array of results, which will contain the SA-interval obtained by each thread respectively

In this section we describe an algorithm that achieves 1 error alignment in GPU. So far the program only returns a result to a read, when it exactly matches a section in the reference string. Now we also want to return a result, where one symbol of the read may differ in a defined (Table 1) form. This complicates the algorithm significantly, because it needs to check in every position of W for all types of errors respectively.

As shown in figure 3 the inexact search will process W symbol by symbol, searching for matches, until it finds a deletion, mismatch or insertion. If so, it executes the exact search discussed beforehand (Section 6.4) from there on to validate the found sequence as a sequence with exactly one error and to save the result. If

Algorithm 3 GPU search algorithm allowing up to 1 error

```
1: function GPU_1_ERROR(W, nW, C, O, k_ini, l_ini, result_list)
2:     offset ← blockIdx.x * blockDim.x + threadIdx.x
3:     if threadIdx.x<4 then
4:         Cshared[threadIdx.x] ← C[threadIdx.x]
5:     end if
6:     __syncthreads()
7:     k_next ← k_ini
8:     l_next ← l_ini
9:     for i ← nW[offset] − 1 ... 0 do
10:        k ← k_next
11:        l ← l_next
12:        if k > l then return
13:        symbol ← W[offset][i]
14:        k_next ← Cshared[symbol] + O[symbol, k] + 1
15:        l_next ← Cshared[symbol] + O[symbol, l + 1]
16:        result ← exact_search(W, nW, start, i − 1, C, O)
17:        if result.k <= result.l then                          ▷ Deletion
18:            add_to_results(deletion, k, l, i)
19:        for symbol ← [A, C, G, T] do
20:            k_aux ← Cshared[symbol] + O[symbol, k] + 1
21:            l_aux ← Cshared[symbol] + O[symbol, l + 1]
22:            if k_aux > l_aux then continue
23:            if symbol ≠ W[offset][i] then                     ▷ Mismatch
24:                result ← exact_search(W, nW, start, i − 1, C, O)
25:                if result.k <= result.l then
26:                    add_to_results(mismatch, k_aux, l_aux, i, symbol)
27:            end if
28:            result ← exact_search(W, nW, start, i, C, O)
29:            if result.k <= result.l then                      ▷ Insertion
30:                add_to_results(insertion, k_aux, l_aux, i, symbol)
31:        end for
32:    end for
33:    if result.k <= result.l then                              ▷ Match
34:        add_to_results(match, k_next, l_next, i)
35: end function
```

this is not the case, the algorithm carries on until the last symbol of W to either save the SA interval of matches or just exits the function without finding any result.

This opens 9 possible branches for every symbol of W, which are seperated by conditional statements in the kernel code.

Description	**Number of possibilites**
A symbol matches exactly the reference	1
A symbol is replaced by one of the other symbols	3
A symbol is deleted	1
Any symbol is inserted	4

So with each symbol of W the program code can follow a different path, which limits the potential parallelism. Let us take a look at the algorithm 3. Each condition in the algorithm can force the threads to take two different code paths, meaning that these sections may be executed sequentially. Take for example line 19, here the algorithm checks for every of the 4 possible symbols if the query appears in the reference or not. After this iteration, it checks if the found SA-interval is valid in line 22. Now let us assume the following:

$$\begin{array}{ll} \text{Reference:} & W = A \\ \text{Reads:} & X1 = A \\ & X2 = G \\ & X3 = C \end{array}$$

We start the execution at line 19 with $symbol \leftarrow A$. When the execution reaches line 22, thread 1 will continue with the next line of code, whether thread 2 and 3 return to the start of the loop and their execution is stopped. Later, thread 1 also reaches again the beginning of the loop. From there on all 3 threads will be once again executed in parallel until the next condition branch, which they do not comply equally.

This algorithm guarantees to find every possible type of result along the search process. During the search, the algorithm splits up the threads. Half of the group enters a condiconal statement and is executed in parallel, while the other half jumps over the condition and is paused to be executed together with the others again.

7 Experimental results

7.1 Hardware Specification

All the tests in this section were executed on a machine with two Intel® Core™2 Duo E7300 CPUs running at 2.66GHz and 2GB of RAM. It features a GeForce GTX280 (See table 4) with 30 multiprocessors running at 1.30 GHz. In the experiments we use only one Intel core with one thread.

The disk drive is a 250GB Seagate ST3250310AS, with 7200 RPM spin speed, SATA 3GB/s serial connection, 8MB cache and a sustained data transfer rate of 106MB/s.

```
Device 0: "GeForce GTX 280"
CUDA Driver Version / Runtime Version          4.0 / 4.0
CUDA Capability Major/Minor version number:    1.3
Total amount of global memory:                 4096 MBytes (4294770688 bytes)
(30) Multiprocessors x ( 8) CUDA Cores/MP:     240 CUDA Cores
GPU Clock Speed:                               1.30 GHz
Memory Clock rate:                             800.00 Mhz
Memory Bus Width:                              512-bit
Max Texture Dimension Size (x,y,z)             1D=(8192), 2D=(65536,32768), 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers        1D=(8192) x 512, 2D=(8192,8192) x 512
Total amount of constant memory:               65536
bytes Total amount of shared memory per block: 16384 bytes
Total number of registers available per block: 16384
Warp size:                                     32
Maximum number of threads per block:           512
Maximum sizes of each dimension of a block:    512 x 512 x 64
Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
Maximum memory pitch:                          2147483647 bytes
...
```

Figure 4: Device query of the GeForce GTX280, which shows a brief hardware summary of the graphics card.

7.2 Profiling of the GPU algorithm

In this section we perform a profiling of the algorithm, measuring separately the different steps of the mapping process. These steps are the following:

- Read vector O from disk
- Copy O to GPU
- Read search strings from disk
- Copy search strings to GPU
- GPU kernel execution
- Copy results to CPU
- Write results to disk

Studying the time that takes each step, allows us to detect bottlenecks. Also, this study allows us to separate the tasks that rely on the input/output capabilities of the system from the time needed by the algorithm (Figure 6).

We executed a single threaded GPU version of the algorithm and aligned 2 million reads with a length ranging from 30 to 200 nucleotides against the *Drosophila melanogaster* genome. The tests were performed, allowing a maximum number of 50 hits per read. The reason for this is the unreasonably high number of hits, which occur for a few reads. The majority of threads only obtains $20-25$ results as shown in figure 5, nevertheless we need to allocate the same amount of memory for a maximum number of hits for all threads. Consequently it is necessary to allocate the same amount of memory for all threads depending on the maximum number of hits that one read can return. This is not the most optimal approach but sufficient, as seen in figure 5. It would be more convenient to allocate memory for the hits dynamically (which is not an option) or to organise the results in a different way.

For example, we could save every result consecutively in the results array and sort them post search. This opens the possibility to perform more parallel searches with one kernel launch, but the results need to be sorted afterwards. This is because the results of one thread can be distributed all over the results array, making it computationally intensive to retrieve them. The time needed for sorting the results afterwards however needs to be reasonable to still leave the whole process effective, which has to verified in that case. Also, atomic operations are needed to access the global array positions. These operations are very expensive.

In the test case shown (Figure 5), we find ~12200000 results with a max hits parameter of 25. During every kernel launch we allocate ~205MB on the device to save the results. Increasing the max hits parameter only gives us a few more

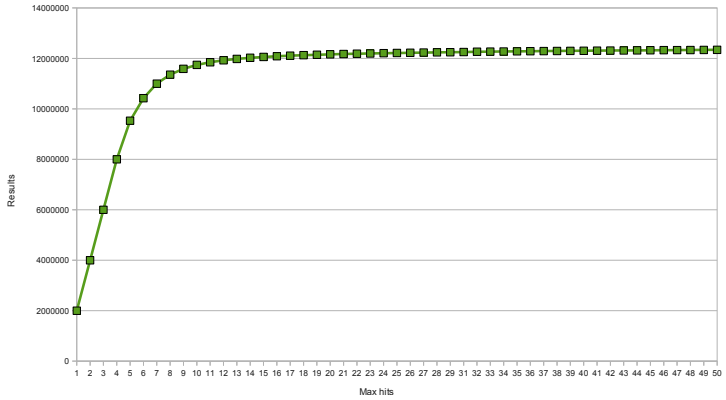


Figure 5: Number of results obtained with increasing number of max hits.

results, which means that the majority of reads does not show more tan 25 results. Trying to return all possible hits (Tests show that there are reads which show 209 hits) of every read obligates us to allocate ~1.700MB of memory. As the majority of reads does not show more that ~25 results, we allocate ~1.500MB and copy it back to the host afterwards for no reason.

The results depicted in figure 6 show that reading the search strings from disk and writing the results to disk together consume about 40% of the execution time. The kernel execution with about 55% is he most time consuming part. To save time, we could use a multi-threaded version of the program presented in [18]. One thread for reading, one for writing operations and one to control the kernel. Using 2 different hard disks, one for read and another for write operations respectively would result in even more speed-up. With these improvements we could reduce the time consumed by input/output operations to a minimum as they would overlap with the time consumed by the kernel execution.

Figure 7 shows a comparison of kernel execution times when launched with different block sizes. A blocksize of 32 gives the best result with a time of ~30 seconds.

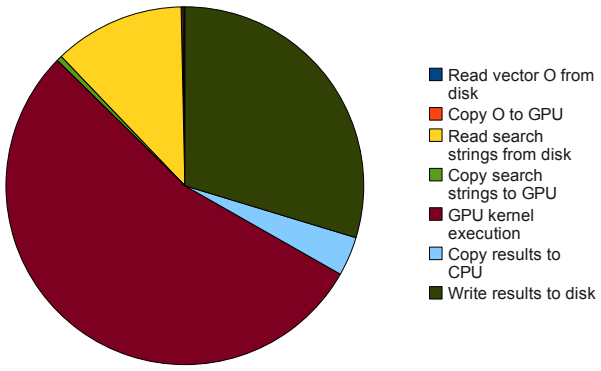


Figure 6: Profiling the algorithm with operative disk cache, permitting a maximum number of 50 hits.

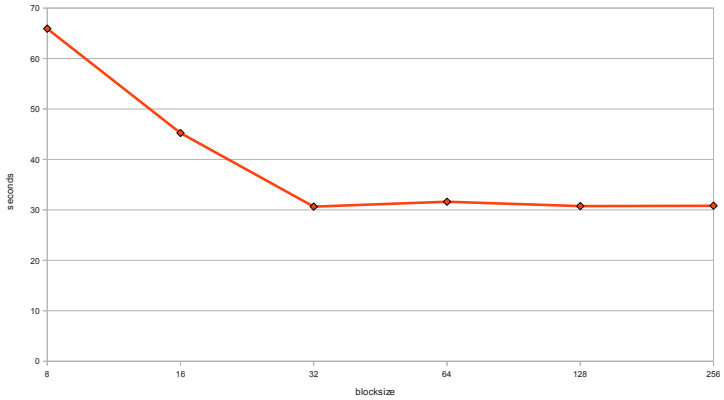


Figure 7: Kernel execution times with different blocksizes.

7.3 Speed-Up analysis

In this section we present the results of the speed-up analysis. The setting is the same as in the Profiling section 7.2, but we ran different tests. One with a max hits parameter of 25 and one with 50. The times for I/O operations are not taken into consideration, because they do affect both versions in the same way.

On the one hand we have the 1-thread CPU version and on the other hand the GPU-CPU hybrid version. The times of the GPU version include copying the search strings to the device, as well as retrieving the results. The measured speed-up of the GPU version lays between 7,32 and 7,60.

The speed-up measured without copy operations shows, that these operations do not play a huge role here.

CPU		**GPU**	
Algorithm	243430s	Copy to device	284s
		Kernel execution	30658s
		Copy to host	1013s
Speed-Up		7,62	
Without copies		7,94	

Table 4: Speed-Up between CPU and GPU with a max hits parameter of 25.

CPU		**GPU**	
Algorithm	244703s	Copy to device	284s
		Kernel execution	31049s
		Copy to host	2026s
Speed-Up		7,34	
Without copies		7,88	

Table 5: Speed-Up between CPU and GPU with a max hits parameter of 50.

8 Conclusions

In this work we have implemented an inexact DNA aligner based on the BWT in GPUGPU. The program presents a solution to rapidly align billions of short-reads to a reference sequence. Apart from the exact match, errors such as a mismatch, insertion or a deletion of a symbol are supported. This work proves, that a parallel GPGPU approach is justified. It shows a reasonable speed-up of 7+ compared to the CPU version, excluding input/output operations.

The project was realised with quite low-level computing, which is mandatory, when dealing with data volumes in the area of tera- to petabytes. Starting point of this final year project was a working implementation of the used algorithm (inexact mapping - 1 error) in CPU. This algorithm shows maximum sensitivity, because it is a non-heuristic approach. It would therefore perfectly serve for data-pre-processing in a multi-stage processing environment, as it does not distort the result.

It is difficult to compare this approach in an objective way with already existing solutions such as Bowtie or SOAP as they function according to heuristic principles. In addition their result sets are smaller because they consider lesser types of errors. Still our program can compete with the just mentioned traditional tools in terms of the speed with which the alignment is performed.

To give a general vision of possible future works to this field of work, we can suggest the following. First of all the use of fast hard-disk drives is recommended to reduce the dependence of input/output operations, which consume about 45% of the alignment process, as stated in section 7.2. Furthermore the use of different threads (one for input, one for output operations and one to control kernel launches) would cause significant speed improvements. It is also conceivable to employ various graphics cards, each processing a different block of reads at the same time, to increase speed. The next step will be developing an algorithm allowing more than the change of a single symbol (more than one error), that terminates in a reasonable amount of time.

References

[1] Bowtie Homepage. `http://bowtie.cbcb.umd.edu`, 2012. [Online; accessed 12-July-2012].

[2] SOAP Homepage. `http://soap.genomics.org.cn`, 2012. [Online; accessed 10-July-2012].

[3] Altschul et al. Basic local alignment search tool. *J. Mol. Biol.*, pages 403–410, 1990.

[4] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped Blast and Psi-Blast: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

[5] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, DEC SRC, 1994.

[6] Ferragina and Manzini. Opportunistic data structures with applications. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 2000.

[7] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.

[8] 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*, 10(R25), Mar. 2009.

[9] Larsson and Sadakane. Faster suffix sorting. *TCS: Theoretical Computer Science*, 387, 2007.

[10] H. Li and R. Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[11] R. Li, Y. Li, K. Kristiansen, and J. Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[12] R. Li, C. Yu, Y. Li, T. W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[13] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Sci.*, 227:1435–1441, 1985.

[14] C.-M. Liu, T. K. F. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, and T. W. Lam. SOAP3: ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics*, 28(6):878–879, 2012.

[15] Manzini. An analysis of the burrows-wheeler transform. *JACM: Journal of the ACM*, 48, 2001.

[16] NVIDIA. Documentation in form of slides and videos. `http://www.nvidia.com/content/cuda/cuda-developer-resources.html`, 2012. [Online; accessed 19-July-2012].

[17] W. R. Pearson and D. J. Lippman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. USA*, 85:2444–2448, Apr. 1988.

[18] J. Salavert Torres, I. Blanquer Espert, A. Tomas Dominguez, V. Hernendez, I. Medina, J. Terraga, and J. Dopazo. Using gpus for the exact alignment of short-read genetic sequences by means of the burrows–wheeler transform. *IEEE/ACM transactions on computational biology and bioinformatics / IEEE, ACM*, 2012 Mar 20 2012.

[19] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, Vol. 147:195–197, 1981.