BIOINFORMATICS ORIGINAL PAPER Vol. 27 no. 10 2011, pages 1351–1358
doi:10.1093/bioinformatics/btr151

Sequence analysis

Advance Access publication March 30, 2011

Exact and complete short-read alignment to microbial genomes using Graphics Processing Unit programming

Jochen Blom[1,†,*], Tobias Jakobi[1,†], Daniel Doppmeier[2], Sebastian Jaenicke[1], Jörn Kalinowski[2], Jens Stoye[3,4] and Alexander Goesmann[1,4,5]

[1]Computational Genomics, [2]Institute for Genome Research and Systems Biology, CeBiTec, [3]Genome Informatics, Faculty of Technology, [4]Institute for Bioinformatics and [5]Bioinformatics Resource Facility, CeBiTec, Bielefeld University, Bielefeld, Germany

Associate Editor: Alfonso Valencia

ABSTRACT

Motivation: The introduction of next-generation sequencing techniques and especially the high-throughput systems Solexa (Illumina Inc.) and SOLiD (ABI) made the mapping of short reads to reference sequences a standard application in modern bioinformatics. Short-read alignment is needed for reference based re-sequencing of complete genomes as well as for gene expression analysis based on transcriptome sequencing. Several approaches were developed during the last years allowing for a fast alignment of short sequences to a given template. Methods available to date use heuristic techniques to gain a speedup of the alignments, thereby missing possible alignment positions. Furthermore, most approaches return only one best hit for every query sequence, thus losing the potentially valuable information of alternative alignment positions with identical scores.

Results: We developed SARUMAN (Semiglobal Alignment of short Reads Using CUDA and NeedleMAN-Wunsch), a mapping approach that returns all possible alignment positions of a read in a reference sequence under a given error threshold, together with one optimal alignment for each of these positions. Alignments are computed in parallel on graphics hardware, facilitating an considerable speedup of this normally time-consuming step. Combining our filter algorithm with CUDA-accelerated alignments, we were able to align reads to microbial genomes in time comparable or even faster than all published approaches, while still providing an exact, complete and optimal result. At the same time, SARUMAN runs on every standard Linux PC with a CUDA-compatible graphics accelerator.

Availability: http://www.cebitec.uni-bielefeld.de/brf/saruman/saruman.html.

Contact: jblom@cebitec.uni-bielefeld.de

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on October 27, 2010; revised on March 15, 2011; accepted on March 22, 2011

1 INTRODUCTION

1.1 Challenges of next-generation sequencing

Together with the advent of new high-throughput sequencing technologies, the amount of generated biological data steadily increased. These techniques allow for cost-effective sequencing of complete libraries of different bacterial strains that may provide new insights e.g. into microevolution, but experimental data need to be processed before any conclusion can be drawn. Given an Illumina Solexa setup with a 36-bp read length, even one lane on the flow cell will suffice for deep coverage sequencing of several bacterial genomes. Typically, the generated reads are mapped on a closely related reference genome to perform a targeted re-sequencing, an in-depth SNP analysis, or to gain knowledge about gene expression when performing transcriptomic experiments using cDNA sequencing. Different tools for mapping reads against reference genomes are available at this time including MAQ (Li *et al.*, 2008), BWA (Li and Durbin, 2009), Bowtie (Langmead *et al.*, 2009), PASS (Campagna *et al.*, 2009), SHRiMP (Rumble *et al.*, 2009) and SOAP2 (Li *et al.*, 2009). MAQ was one of the first mapping tools available and uses a hash data structure to keep reads in memory while traversing the reference genome, resulting in a comparably low memory footprint. But it lacks support for the output of more than one mapping position per read and is not able to compete with more recent approaches in terms of speed (Langmead *et al.*, 2009). BWA, Bowtie and SOAP2 are indexing the complete reference genome using a Burrows-Wheeler-Transformation (BWT) (Burrows and Wheeler, 1994) and process all the reads sequentially, resulting in a considerable speedup of the mapping process. PASS and SHRiMP also perform an indexing of the reference sequence, but use a spaced seed approach (Califano and Rigoutsos, 2002) instead of BWT. All mentioned approaches except SHRiMP are heuristic and do not guarantee the mapping of all possible reads. Especially reads that show insertions or deletions (indels) compared to the reference sequence are often missed. Bowtie is unable to identify such alignments by design, while SOAP2 only supports gapped alignments in paired-end mode. As high accuracy and confidence of short-read alignment is highly desirable, especially in the analysis of single nucleotide polymorphisms (SNPs) or small-scale structural variations, we decided to design an exact short-read alignment approach that identifies all match positions for each read under a given error tolerance, and generates one optimal alignment for each of these positions without any heuristic. To obtain an adequate runtime even for large-scale, whole-genome applications, we employ the massively parallel compute power of modern graphics adapters [Graphic Processing Unit (GPUs)]. A qgram-based (Jokinen and Ukkonen, 1991) filter algorithm was designed to find auspicious alignment positions of short reads within the reference sequence. After the localization of possible

*To whom correspondence should be addressed.

†The authors wish it to be known that, in their opinion, the first two authors should be regarded as joint First Authors.

mapping positions, we use a Needleman–Wunsch (Needleman and Wunsch, 1970) algorithm to compute optimal alignments of the candidate sequences to the reference sequence. The alignments are processed in parallel on a NVIDIA graphics card using the NVIDIA CUDA (Compute Unified Device Architecture) framework. This combination of a filter step with the parallel computation of optimal alignments on graphics hardware allows us to compute an exact and complete mapping of all reads to the reference in a time comparable to existing heuristic approaches.

1.2 Parallelization in biosequence analysis

In computational biology, huge amounts of data need to be processed and analyzed, therefore suggesting the exploration and evaluation of new computational technologies like parallel programming. One basic idea behind parallel programming is to divide a problem into smaller, easier to solve subproblems, a technique known as 'divide and conquer'. Another approach is to solve a huge number of small independent tasks by parallelization such that each problem can be solved on a different processing unit, either cores of one computer or a compute cluster. Once all calculations are finished, they are combined into a solution of the original problem. While server systems with 32 and more CPU cores are available today and can be used to efficiently speedup multithreaded software, they still pose a significant capital investment. Therefore, specialized hardware for parallel processing has been employed, such as Celeras GeneMatch™ASIC (application-specific integrated circuit) approach which uses specialized processors to accelerate several bioinformatics algorithms. A few years later, TimeLogic developed Field Programmable Gate Arrays (FPGAs) to run adapted versions of the Smith–Waterman, BLAST and HMMer software with significant performance gains. Unfortunately, the prices for such optimized special purpose hardware together with appropriate licenses lie in the same expensive investment range as large servers. A more cost-efficient possibility is the use of existing hardware which can be employed for scientific computing through different frameworks. The MMX technology introduced by Intel in 1997 is a SIMD (single input multiple data) approach which allows for parallel processing of large amounts of data with specialized CPU instructions. The MMX instruction set as well as its successor SSE (Streaming SIMD Extensions) have been used to accelerate implementations (Farrar, 2007; Rognes and Seeberg, 2000) of the Smith–Waterman algorithm (Smith and Waterman, 1981). In 2008, also a first attempt on non-PC hardware has been published. SWPS3 (Szalkowski *et al.*, 2008) employs the Playstation 3's cell processor to speedup an adapted version of the Smith–Waterman algorithm. Another trend in the recent past is the use of modern graphics adapters in scientific computation due to their immense processing power which still increases at a much faster rate than the processing power of general purpose processors. While there are several implementations available (Liu *et al.*, 2009; Manavski and Valle, 2008) that focus on the alignment of one query sequence to a database of reference sequences, an algorithmic adaptation of massively parallel pairwise alignments, as it could be used for short-read alignments, is still missing. First approaches using graphics cards as hardware accelerators for bioinformatics algorithms (Liu *et al.*, 2006) relied on OpenGL, resulting in a difficult and limited implementation. Today, frameworks simplify software development by hiding the layer of 3D programming behind a more general Application Programming Interface (API). Thus, the focus of development shifts from fitting a given algorithm to OpenGL operations to the development of the best implementation. The CUDA platform developed by the NVIDIA cooperation (http://www.nvidia.com/object/cuda_home.html) and ATI's Stream framework (http://www.amd.com/stream) are novel approaches to use the huge computational power of modern graphics cards not only for games but also for scientific applications. Contemporary graphics processing units are built as massively parallel computational devices, optimized for floating point operations. Compared to universal central processing units (CPUs) used in every computer, GPUs are specialized for parallel execution of many small tasks while CPUs are designed to execute fewer large tasks sequentially. As such, GPUs are also well suited for the highly parallel computation of small-scale alignments.

2 METHODS

2.1 Problem

The short-read matching problem can be defined as follows: we have given a short sequencing read f of length $|f|=m$, a (in most cases genomic) reference sequence g of length $|g|=n$, and an error threshold $e\geq 0$ defined by the user. Then we want to calculate all starting positions i in g, such that there exists an alignment of f and a prefix of $g[i...]$ with at most e errors (mismatches and/or indels). The algorithm shall be capable to export an optimal alignment for every such match position.

This problem has been studied widely in the past, and most solutions are based on one of the two following principles, or combinations thereof: the *qgram lemma* states that two strings P and S with an edit distance of e share at least t qgrams, that is substrings of length q, where $t=max(|P|,|S|)-q+1-q\cdot e$.

That means that every error may destroy up to $q\cdot e$ overlapping qgrams. For non-overlapping qgrams, one error can destroy only the qgram in which it is located, which results in the applicability of the *pigeonhole principle*. The pigeonhole principle states that, if n objects (errors) are to be allocated to m containers (segments), then at least one container must hold no fewer than $\lceil \frac{n}{m} \rceil$ objects. Similarly, at least one container must hold no more than $\lfloor \frac{n}{m} \rfloor$ objects. If $n<m$, it follows that $\lfloor \frac{n}{m} \rfloor = 0$, which means that at least one container (segment) has to be empty (free of errors). Moreover, if $n<m$ this holds for at least $m-n$ segments. In our algorithm, presented in the following section, we first use a 2-fold application of the pigeonhole principle to find regions of interest in the genome, before we apply parameters derived from the qgram lemma to make our final selection of positions to pass the filter.

2.2 Solution

Assumptions and definitions: as a basic assumption, we require a minimal length of reads in relation to the given error threshold: $m>e+1$. Given this assumption, we calculate the length of the qgrams for our filter algorithm as follows. We define the length q of the qgrams as the largest value below $\frac{m}{e+1}$ such that

$$q' := \lfloor \frac{m}{e+1} \rfloor, \qquad q := \begin{cases} q' \text{ if } (e+1)q' < m, \\ q'-1 \text{ otherwise.} \end{cases}$$

This guarantees that a read f can be split into $e+1$ intervals with an additional non-empty remainder of length $R := m-(e+1)q$.

Given the calculated qgram length, we create an index I of the starting positions of qgrams in sequence g, such that for each possible qgram x, $I(x)$ contains the starting positions of x in g.

In the following step of the filter algorithm, every read is segmented into pieces of length q (Fig. 1). We choose a set S of $c=\lfloor \frac{m}{q} \rfloor$ segments $S=s_1...s_c$ of length q from f, such that for $i=1,...,c: s_i = f[(i-1)*q+1...i*q]$. As in

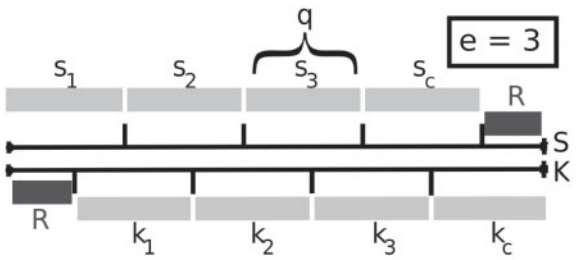

Fig. 1. The two sets of segments S and K. The two sets are shifted by the distance of R.

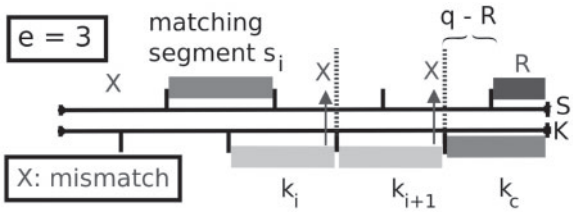

Fig. 2. A matching segment $s_i \in S$ where no other segment of S has a match in correct distance. Segments of set K are checked in this case. If none of the segments $k_i ... k_c$ matches, the read cannot match at the respective genome position.

most cases $q = \lfloor \frac{m}{e+1} \rfloor$, it can easily be shown that c converges against $e+1$ for increasing read length m. We additionally choose a second set K of c segments $K = k_1 ... k_c$ of length q, such that for $i = 1, ..., c : k_i = f[(i-1)*q+1+R...i*q+R]$, a set shifted by the remainder R.

The simple version of our algorithm allows for mismatches only, not for insertion or deletions. In this case, it can be shown by the following observations that a matching read f must have at least two matching segments from the sets S and/or K. There are two cases, in the first case two segments of set S will match, in the second case one segment from S and one segment from K will match. The term 'matching segment' in the following indicates that a starting position for the respective segment can be found in the genome via exact occurrence within index I. The first segment found in the index is used as seed, the second 'matching' segment has to be listed in the index in appropriate distance to this seed. The algorithm is based on the assumption that S and K comprise $c = e+1$ segments. In cases where $c > e+1$, we know by the pigeonhole principle that at least two segments of S must match, which fulfills our filter criterion directly. The segment set K is not needed in such cases, and all algorithmic steps after the first can be skipped.

Filter algorithm: by the pigeonhole principle, we know that one segment of set S 'must' match, as we allow e errors and have $e+1$ segments. We can identify all match positions of segments $s_i \in S$ for the given read in the qgram index I and use them as starting points for the filter algorithm.

(1) For every segment s_i matching at a position b in the genome g we check in I if there is another segment $s_j \in S, j > i$, that starts at the expected position $b+(j-i)*q$. If we find such a segment, we identified the two matching segments we expect.

In the case that only one segment $s_i \in S$ matches, there has to be exactly one error in every remaining segment $s_1 ... s_{i-1}, s_{i+1} ... s_c$. Otherwise, a second segment of set S would match. We can infer that not more than $c-i$ errors are remaining in the segments to the right of read s_i, but due to the overlapping construction of our segment sets we have one segment more of K to the right of read s_i to check. Hence, if read f is a possible hit, one of these $c-(i-1)$ segments $k_i ... k_c$ has to match, and we can start the checks for set K at position k_i.

(2) Check if segment k_i overlapping s_i matches. If it does, we have successfully matched 2 qgrams and passed the filter.

If segment k_i does not match, we know that k_i overlaps s_i on $q-R$ positions. These positions are free from errors (as s_i matched without errors). Thus, the error causing k_i not to match must have been on the last R positions of k_i which are the first R positions of s_{i+1}. As there is exactly one error in every segment of S, we can conclude that the last $q-R$ positions of s_{i+1} are free of errors, which are the first $q-R$ positions of k_{i+1}. So if k_{i+1} does not match, the next error is in the first R positions of s_{i+2} and so on.

(3) Iteratively check all remaining segments $k_{i+1} ... k_c$ until one segment matches and the read f passes the filter or until k_c is reached.

If we reach k_c, that means that k_{c-1} did not match, we know that the error of s_c was in the first R positions. So the last $q-R$ positions of s_c must be correct, and so must be the first $q-R$ positions of k_c. The remaining R positions of k_c must also be correct because all errors are within segments $s_1 ... s_c$. The last R positions of k_c are not part of one of these segments, so any error within them would be the $e+1$st one. If k_c does not match, it can be excluded that read f can be aligned to the reference sequence g with a maximum of e errors at the actual position. See Figure 2 as illustration of this matching process.

Insertions and Deletions: it is possible to allow indels in the matching process by checking every segment not only on one position, but also on several positions by shifting the segment to the left or right by up to t positions, where $t = e-(i-1)$ for an initial matching segment s_i as at least $i-1$ errors have already occurred in the previous $i-1$ segments. Hence, we conclude that if there are only e errors in f, it is always possible to match (i) two segments of S or (ii) one segment of S and one segment of K. Algorithm 1 identifies reads as candidates for a Needleman–Wunsch alignment by checking for this condition.

Alignment phase: as soon as a second qgram hit has been found for a given start index B, the alignment of f to $g[B-e, B+m+e]$ is enqueued for alignment and the rest of the filter phase for this value of B can be skipped. To speed up this alignment procedure in practice, the verification by alignment is computed on graphics hardware. Due to the parallel computation of alignments on graphics hardware, a huge number of possible alignment positions is collected before being submitted to the graphics card. The actual number depends on the amount of memory available on the graphics card (VRAM). The alignment of the read sequence f to the possibly matching part $g[B-e, B+m+e]$ of the reference sequence identified in the filter step is computed by a Needleman–Wunsch algorithm. On both sides of this template sequence, e, additional bases are extracted from the reference sequence to allow for indels in an end-gap free alignment. We use unit edit costs for the alignment matrix. As soon as the error threshold is exceeded in a complete column of the distance matrix, the calculation is stopped, the read is discarded and backtracing is omitted. If the complete distance matrix is computed, an optimal alignment path is calculated in the backtracing step and the read is reported as hit together with its optimal alignment. SARUMAN does not use banded alignments, yet, this is planned for future releases.

3 IMPLEMENTATION

The feasibility of sequence alignments using GPU hardware was demonstrated by different tools like SW-Cuda (Manavski and Valle, 2008) or CUDASW++ (Liu *et al.*, 2009). But, compared to our solution, existing implementations focused on the search for similar sequences in a huge set of other sequences, which corresponds to a BLAST-like use. In contrast, SARUMAN searches for local alignments of millions of short sequences against one long reference sequence. Employing the filtering algorithm described in the previous section, all possible alignment positions in the reference genome can be identified, and thereby the problem is reduced to

Algorithm 1 (Mapper)

Require: A read f of length m
Require: A reference sequence g of length n
1. ## Set of candidate positions ##
2. $C \leftarrow \emptyset$
3. ## q calculated as described in Section 2.2 ##
4. $c \leftarrow \lfloor \frac{m}{q} \rfloor$
5. $R \leftarrow m - c \cdot q$
6. **for** $i \leftarrow 1, \ldots, c$ **do**
7. $u \leftarrow (i-1) \cdot q + 1$
8. ## initial matching segments ##
9. **for each** b **in** $I(s_i)$ **do**
10. $B \leftarrow b - u$
11. ## check remaining segments of S ##
12. **for** $j \leftarrow i+1, \ldots, c$ **do**
13. $v \leftarrow (j-1) \cdot q + 1$
14. ## shift by up to $e-(i-1)$ positions ##
15. **for** $t \leftarrow -(e-(i-1)), \ldots, +(e-(i-1))$ **do**
16. ## if distance fits, add to alignment queue ##
17. **if** $B+v+t \in I(s_j)$ **then**
18. add B to C
19. **end if**
20. **end for**
21. **end for**
22. **if** not $B \in C$ **then**
23. ## check segments of K ##
24. **for** $j \leftarrow i, \ldots, c$ **do**
25. $v \leftarrow (j-1) \cdot q + 1 + R$
26. **for** $t \leftarrow -e-(j-1), \ldots, e-(j-1)$ **do**
27. **if** $B+v+t \in I(k_j)$ **then**
28. add B to C
29. **end if**
30. **end for**
31. **end for**
32. **end if**
33. **end for**
34. **end for**
35. ## Send candidates to alignment ##
36. **for each** $B \in C$ **do**
37. align f to $g[B-e, B+m+e]$ and report hit if successful
38. **end for**

global alignments of a read sequence with a short substring of a reference genome. Thus, compared to SW-Cuda, SARUMAN does not align sequences against a database of templates, but is designed as an alignment application to perform thousands of short pairwise global sequence alignments in parallel.

The CUDA API is an extension to the C programming language. Thus, the mapping part of SARUMAN was implemented in the C programming language to simplify the integration of CUDA code. Our software is divided into two consecutive phases, namely mapping and aligning. Phase one, the creation of the qgram index together with the following mapping of reads through qgrams, is completely processed on the host computer. During phase two, CUDA is used to compute the edit distance for candidate hits on the graphics card using a modified Needleman–Wunsch algorithm. If the computed edit distance of read and genome lies below a given error threshold, the optimal alignment is computed in the backtracing step on demand. The complete workflow of SARUMAN is illustrated in Figure 3.

3.1 Mapping phase

The memory usage of the qgram index depends on the qgram length and the number of replicates per qgram number, but is mainly dependent on the size of the reference genome. To process large reference genomes with limited resources, the reference sequence is divided into several chunks that are processed iteratively. The size of a sequence that can be processed in one iteration is restricted by the available memory. Using this technique, it is possible for SARUMAN to run on computers with small amounts of RAM by dividing the qgram index into chunks perfectly fitting into available memory. Our tests show that a standard computer with 4 GB of RAM and a recent dual core CPU is able to read and process even large bacterial genomes like *Sorangium cellulosum* (13 033 779 bp) without any difficulties. However, this approach is not feasible for large eukaryotic genomes as the number of needed iterations would be too high. Supported read input formats are FASTA as well as FASTQ. While the reads are stored in memory, all sequences containing more than e ambiguous bases (represented as N) are filtered out, due to the fact that an N would nevertheless be treated as a mismatch by SARUMAN. Subsequently, perfect matches are determined by exact text matching and exported as hits. Preceding the actual filtering step, all reads are preprocessed. During this phase, all located start positions of the $2*c$ qgrams of a read in the reference genome are extracted from the qgram hash index. This list of positions is stored in an auxiliary data structure in order to minimize access to the hash index in later stages and therefore speedup the following filter step. Reads with perfect hits on the reference genome are still further processed as there may exist imperfect hits elsewhere on the reference, but for such reads the starting positions of qgrams representing the perfect hit are removed from the auxiliary data structure. After the first steps, each read is mapped onto the reference genome using the algorithm described in Section 2, whereas a read is only mapped once to a given start position to avoid redundancy. While a combination of first and last qgram (e.g. s_1 and k_c) exactly determines start and end position of the read, two 'inner' qgrams (e.g. s_2 and k_{c-1}) would result in a longer alignment in order to find the correct start and stop position. Start positions for possible mappings are stored and transferred to the CUDA module in a later stage. In order to not only exploit the parallel architecture of graphics cards but also the availability of multicore CPUs, the matching phase uses two threads. The first thread handles mapping on the sense strand, whereas the second thread processes mapping on the antisense strand. In contrast to many other approaches, which employ a 2-bit encoding for the four DNA letters, SARUMAN is able to handle Ns in reads as well as in the reference genome. Since many genomes contain a small number of bases with unknown identity, it is of advantage to treat these bases as N. Replacing these positions with random or fixed bases to maintain the 2-bit encoding may lead to wrong and in case of random replacements even to irreproducible results.

3.2 Alignment phase

To efficiently use CUDA, it is of great advantage to understand the underlying hardware of CUDA capable graphics adapters. A GPU consists of a variable number of multiprocessors reaching from 1 in

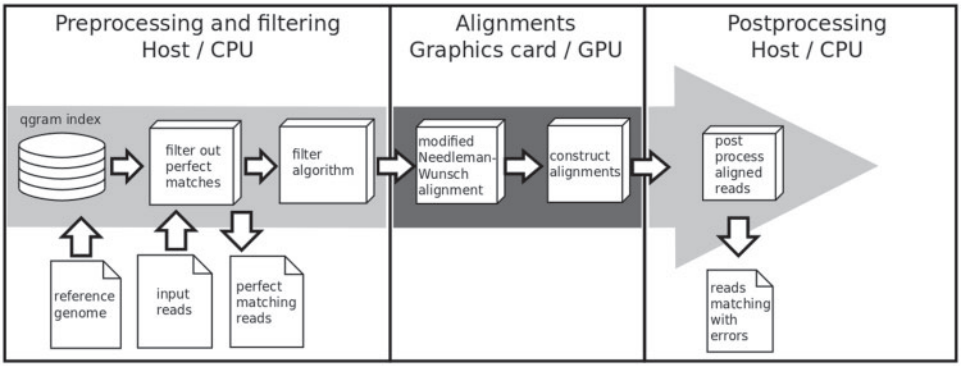

Fig. 3. Program flow within SARUMAN. The three different application phases have been highlighted in different shades of gray. After reading the reference genome, a qgram index is created, followed by the prefiltering of perfect hits. Consecutively, all reads are processed by the filtering algorithm which reports all candidate hits and copies all necessary data to the GPU. Edit distances for candidates are computed and alignments for promising hits calculated, which in the last step are postprocessed and printed out.

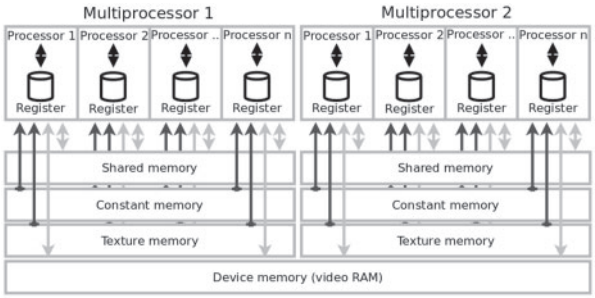

Fig. 4. CUDA hardware layout and memory organization. Each multiprocessor consists of eight processors, each with an own set of registers. Shared, constant and texture memory can be used by each of the eight processors, while device memory is globally accessible between multiprocessors.

entry level graphics adapters up to 60 multiprocessors in high end video cards. Each of these multiprocessors has access to registers of 8 or 16 kb size and is divided into 8 small processors. The available registers are divided and equally assigned to processors. This small amount of writable memory should be used for data processed in the currently active thread while texture memory and constant memory are read only and can be used to prefetch data from the much slower device memory. An overview of the CUDA hardware and memory model is given in Figure 4. Implementing code for the execution on a GPU is very similar to standard application development. Some additional keywords extending the C programming language are used to define on which device a function should be executed. A function on the GPU is mapped to one thread on one processor located on the graphics card, whereas one function can and should be executed on many different datasets in parallel. This execution scheme is called SIMT (single input multiple threads) due to its relation to the similar SIMD scheme used in classical parallel programming. Parallel programming with CUDA is transparent and (within NVIDIA products) device independent for developers and users. Launch of a CUDA application is controlled using only a few parameters defining the total number of threads. Those threads are organized hierarchically into *grids*, which themselves consist of threadblocks. Each threadblock is a collection of a given number of threads. A threadblock must be executable in any order and therefore must not have any dependencies on other blocks of the same grid. Once a sufficient number of candidate hits has been found by the filter algorithm, all necessary data for performing the alignments is collected. Read and genome sequences are stored together with auxiliary data structures. Afterwards, all required data for the alignment phase is copied to the GPU as one large unit in order to minimize I/O overhead. The maximal number of alignments fitting in the GPU memory heavily depends on read length. SARUMAN automatically calculates a save value for the number of parallel alignments. As the memory of the graphics adapter is also a limiting factor, SARUMAN does not use the quality information of reads as this would double up the memory usage. For datasets with sufficient coverage and quality, we recommend to simply remove all reads with low-quality bases. For 36 bp reads, a value of 200 000 alignments (100 000 for each mapping thread and direction) can be achieved on a standard GPU with 1 GB of graphics memory [Video RAM (VRAM)]. Once all data of the candidate hits has been copied to the GPU for each pair of genome and read sequence, the edit distance is computed using the desired values for match and mismatch positions. By comparing the distance with the supplied maximal error rate, all candidates with values above this threshold are discarded. Complete alignments are only computed in a second backtracing step for candidates with a tolerable edit distance. Typically, the alignment phase for one batch takes only a few seconds to complete, including the whole process of copying raw data to and processed alignments from the GPU. Before any output is written, alignments are postprocessed by clipping gaps at the start and end. For each possible start position of each read, the optimal alignment is reported, in contrast to other available tools which only deliver a fixed number of positions or do not even guarantee to report any optimal alignment. SARUMAN produces tab separated output by default which includes start and stop position of the mapping together with the edit distance and the complete alignment. The package includes an easy to use conversion tool to generate the widely used SAM alignment format. The SARUMAN software is available for download at http://www.cebitec.uni-bielefeld.de/brf/saruman/saruman.html.

4 RESULTS

4.1 Evaluation methods

The evaluation was accomplished on a standard desktop PC with an Intel Core2Duo E8400 3 GHz dual core processor with 8 GB DDR2 RAM and a GeForce GTX280 graphics card with 1 GB of VRAM. All programs were run multithreaded on both processor cores, the operating system was Ubuntu Linux. We used four datasets for the performance evaluation, three synthetic read sets of roughly 18 mio. reads generated from the *Escherichia coli K12 MG1655* genome (GenBank accession NC_000913), and a real dataset with data from one lane of a re-sequencing run of *Corynebacterium glutamicum ATCC 13032* (GenBank Accession BX927147) using the Illumina Solexa GAII sequencer, comprising 18 161 299 reads of 35 bp length. The settings for all programs used in the comparisons were adjusted to make the run parameters as comparable as possible. To achieve this, we allowed two mismatches/indels for each read and set all programs to support multithreading. Furthermore, we allowed gapped alignment of reads where possible and adjusted the alignment scoring matrix to simple unit costs. Detailed settings for all tool runs are given in the Supplementary Material.

4.2 Performance evaluation

In order to prove the exactness and completeness of the presented approach and to measure the discrepancy between exact and heuristic implementations, we used the synthetic read sets described in Section 4.1. Synthetic reads were generated with 36, 75 and 100 bp length in both directions with up to two errors of different types, i.e. mismatches, insertions, deletions and combinations thereof. About three million of the artificial reads contained indels. These reads were mapped to the original source genome *Escherichia coli K12 MG1655*. The dataset is available on the project homepage. Table 1 compares the mapping ratios of the different tools together with their respective running time for 75 bp reads. Data for 36 bp reads and 100 bp reads are provided in the Supplementary Material. The goal was to map all artificial reads on the genome at the exact position without missing any mappings. As expected, using SARUMAN we were able to map all artificial reads to the genome. Furthermore, nearly all reads were mapped to the correct position in the reference genome. In some rare cases (ca. 0.45%), SARUMAN returned optimal alignments that are shifted from the reads original position by up to e bases (Fig. 5). Such cases cannot be resolved, this is a general problem of the edit cost function and not a flaw of SARUMAN. Additionally, SARUMAN reported a large number of other matches on different sites of the genome. Among them were six matches that placed a read to an alternative position in the reference genome with a better score than the alignment to the original position. In this case, the incorporation of errors led to a read that just by chance fits better to a wrong genomic position. While alternative hits can be identified as misplaced in synthetic data, this behavior is preferable for real data as one can not determine the correct mapping position among several equally good locations.

The evaluation shows a clear separation of the programs into two classes, depending on their ability to handle gaps. Neither SOAP2 nor Bowtie are able to perform gapped alignments, both tools were

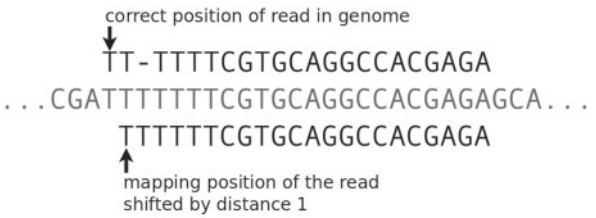

Fig. 5. Example for an ambiguous mapping. The read has a deletion in a poly-T region compared to the reference. This results in a shift of the mapping position by one base as it is not possible to determine in which position the deletion event happened.

Table 1. Sensitivity evaluation with an artificial dataset of 17.980.142 reads (75 bp) generated from *Escherichia coli K12 MG1655* with up to two errors

	SARUMAN	SOAP2	Bowtie	BWA	SHRiMP	PASS	Reference
Mapped	17 980 142	15 142 908	15 123 838	17 746 484	17 980 142	16 873 044	17 980 142
Not mapped	0	2 837 234	2 856 304	233 658	0	1 107 098	0
Perfect	4 999 944	4 999 942	4 999 944	4 999 944	4 999 944	4 999 944	4 999 944
With errors	12 980 198	10 142 966	10 123 894	12 746 540	12 980 198	11 873 100	12 980 198
1 mismatch	4 999 908	4 999 906	4 999 908	4 999 908	4 999 908	4 999 908	4 999 908
2 mismatches	4 999 936	4 999 936	4 999 936	4 999 936	4 999 936	4 999 936	4 999 936
1 Insertion	499 992	34 648	29 900	489 788	499 992	478 754	499 992
2 Insertions	499 998	1243	496	405 563	499 998	496	499 998
1 Deletion	493 560	46 740	46 740	491 454	493 560	475 916	493 560
2 Deletion	493 354	4828	4828	433 605	493 354	452 714	493 354
1 Ins, & 1 Mism,	500 000	21 374	12 308	458 957	500 000	18 334	500 000
1 Del, & 1 Mism,	493 450	34 291	29 778	467 329	493 450	447 042	493 450
MCP	17 899 092	15 070 286	15 061 178	17 432 842	17 785 567	16 577 180	–
BMCP	17 899 086	15 070 286	15 061 178	17 430 632	17 784 713	16 577 173	–
Total alignments	19 971 674	16 425 886	16 542 168	18 022 317	19 423 932	18 447 418	–
Runtime (min)	12:03	06:40	18:56	15:09	95:06	27:42	–
RAM usage (kb)	3 375 236	702 964	14 420	117 172	1 313 944	399 278	–

MCP (Mapped to Correct Position) denotes the number of mapped reads that had a match to their original position. BMCP (Best Match at Correct Position) denotes the number of reads where the best match was located at the correct position.

Table 2. In-depth analysis of reported mappings of 18 161 299 *C.glutamicum* re-sequencing reads with an error threshold of two

	SARUMAN	SOAP2	Bowtie	BWA	SHRiMP	Pass
Mapped	18 025 584	18 001 767	17 999 961	18 023 109	16 558 248	17 859 475
Not mapped	135 715	159 532	161 338	138 190	1 603 051	301 824
1 alignment	17 406 040	17 467 485	17 388 188	17 732 158	16 057 777	17 284 434
2 alignments	184 224	153 627	181 476	171 385	144 175	167801
3 alignments	72 679	44 237	73 534	58 298	42 699	57630
$\geq$ 4 alignments	362 641	336 418	356 763	61 268	313 597	349610
Alignments total	20 006 760	19 755 348	19 936 446	18 494 894	17 991 074	19 713 095
Runtime (min)	6:08	9:29	19:40	8:30	32:03	23:59

Matches reported by SOAP2 include reads containing Ns, which are treated as mismatch by other programs.

not able to map more than a small portion of sequences generated with indels to their correct position by using mismatches instead of gaps. The second group of tools consists of BWA, PASS, SHRiMP and SARUMAN, which are all capable of aligning reads containing gaps, although PASS shows a poor mapping ratio for reads with more than one indel. BWA shows a very good performance, but still misses more than 200 000 reads. SHRIMP shows a complete mapping of all reads, but places less reads than SARUMAN to the correct position. SARUMAN shows also the best performance on the real *C.glutamicum* dataset presented in Table 2. As this dataset originates from a re-sequencing of the identical strain, only a very low number of errors is expected. Therefore, the differences between the tools are quite small. Nevertheless, SARUMAN shows the highest mapping ratio of all tools, and furthermore produces the highest number of valid alignments. The dataset was evaluated with an error threshold of two: data for one and three allowed errors are given in the Supplementary Material.

Besides sensitivity, another major requirement of short-read alignment approaches is the performance in terms of running time. The runtime evaluation of the artificial reads shows that SOAP is nearly double as fast as SARUMAN, but maps significantly less reads. All other tools are comparably fast (BWA, Bowtie) or considerably slower (SHRIMP, PASS) than our approach. For the *C.glutamicum* re-sequencing dataset, SARUMAN performs best with a running time of 06:08 min, while SOAP and BWA take slightly longer with 9:29 and 8:30 min, respectively. Bowtie, PASS and SHRiMP are three to five times slower. Considering all calculated datasets, SOAP2 shows the best performance of all compared approaches, being the only algorithm that is faster than SARUMAN in most cases. BWA shows a runtime comparable to SARUMAN, while Bowtie is highly dependent on the used error ratio. For low error rates, it can compete with other approaches, for higher error rates the runtime rises significantly. SHRiMP and PASS are the slowest of the compared approaches for all evaluated datasets. In summary, SARUMAN is the only approach that provides exact and complete results, while still being nearly as fast or even faster than all compared approaches. Thereby, it shows the best overall performance for short-read alignment against prokaryotic genomes.

4.3 Performance of filter and alignment components

The performance of the filter algorithm mainly depends on the read length and the qgram length. We compared the number of candidates resulting from the filter algorithm with the number of alignments successfully verified by the alignment step. The results are shown in Table 3. The efficiency of the filter algorithm is slightly increasing with longer read lengths due to the bigger qgram size. The performance of the alignment step is decreasing as the alignments of the longer reads are more memory consuming on the graphics adapter. But even for 100 bp reads, the GPU implementation shows a more than 5-fold speedup compared to the CPU implementation of the same algorithm.

Table 3. Sensitivity of the filter algorithm and performance gain of the GPU implementation, tested on *E.coli* artificial data

Read length	36 bp	75 bp	100 bp
Candidates	23 0405 03	15 405 284	14 991 106
Aligned	14 918 066	14 553 824	14 451 680
Filter step (min)	5:26	4:53	4:49
Alignment on GPU (s)	42	430	1046
Alignment on CPU (s)	1085	3282	5814
GPU:CPU	1:25.83	1:7.63	1:5.55

The upper part shows the runtime of the filter step, the alignment candidates passing the filter and the number of reads that where successfully aligned. The lower part compares the runtime of the alignment step on a GPU versus the runtime on a CPU.

5 DISCUSSION

For short-read alignments against microbial genomes, SARUMAN has proven to be as fast or even faster than other available approaches like SOAP2, Bowtie, BWA, SHRiMP and Pass (Tables 1 and 2). In contrast to heuristic approaches, SARUMAN guarantees to find optimal alignments for all possible alignment positions. Compared to SHRiMP, another exact aproach, SARUMAN shows a better runtime performance. Thus, it is the first non-heuristic approach providing full indel support in competitive computing time. The completeness of the SARUMAN mapping was demonstrated using a constructed dataset, where all reads could be mapped correctly (Table 1). SARUMAN is designed as a short-read alignment approach; nonetheless, it works properly and with reasonable running times for reads up to lengths of 125 bp. The qgram index used in the filter step has a memory usage that in worst case increases linearly with the size of the reference genome. Furthermore, as SARUMAN provides a proper handling of Ns in the reference sequence as well as in the reads, it cannot use two bit encoding

of nucleotides, which further increases the memory footprint. On a standard desktop PC with 4 GB RAM, it can map reads to all bacterial genomes in a single iteration, with 8 GB RAM fungal genomes of up to 50 Mb in size were processed in a single run. If a reference genome is too large for the available memory, it is automatically split into two or more parts as described in Section 3. Of course the running time of the algorithm has to be multiplied by the number of iterations that are needed. Due to this limitation, we propose SARUMAN as dedicated solution for read mapping on microbial reference genomes or other reference sequences of comparable size. SARUMAN does not natively support paired end sequencing data, but as all possible alignments are returned this can be handled by post-processing the results to flag read pairs as matching either in compatible distance or not. A post-processing tool is under development. Several further improvements to SARUMAN are planned for the future. One idea is to combine the CUDA alignment module with other filter algorithms. These may be heuristic solutions to establish an even faster short-read alignment, or algorithms based on compression techniques like the BWT to reduce the memory usage and make the approach more applicable to large reference genomes. The highest potential for a further speedup has the processing of the filtering algorithm on the graphics adapter. Unfortunately, this is not yet foreseeable as it is quite complicated to handle the reference sequence data with the limited amount of memory available on the graphics cards, but it may become feasible with future generations of graphics hardware. Another planned development is a native support for color space data as generated by the SOLiD sequencing system.

ACKNOWLEDGEMENTS

The authors wish to thank the Bioinformatics Resource Facility system administrators for expert technical support.

Funding: German Federal Ministry of Education and Research (grant 0315599B 'GenoMik-Transfer') to J.B. and S.J.

Conflict of Interest: none declared.

REFERENCES

Burrows,M. and Wheeler,D.J. (1994) A block-sorting lossless data compression algorithm. *Technical Report 124*, Digital Systems Research Center, Palo Alto, California.

Califano,A. and Rigoutsos,I. (2002) FLASH: A fast look-up algorithm for string homology. In *Computer Vision and Pattern Recognition, 1993. Proceedings CVPR'93., 1993 IEEE Computer Society Conference on*. IEEE, New York, NY, USA, pp. 353–359.

Campagna,D. *et al.* (2009) PASS: a program to align short sequences. *Bioinformatics*, **25**, 967.

Farrar,M. (2007) Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, **23**, 156–161.

Jokinen,P. and Ukkonen,E. (1991) Two algorithms for approximate string matching in static texts. *Lecture Notes in Computer Science*, **520/1991**, 240–248.

Langmead,B. *et al.* (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol*., **10**, R25.

Liu,W. *et al.* (2006) Bio-sequence database scanning on a GPU. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, IEEE, Rhodes Island, Greece, p. 8.

Liu,Y. *et al.* (2009) CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Res. Notes*, **2**, 73.

Li,H. and Durbin,R. (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, **25**, 1754–1760.

Li,H. *et al.* (2008) Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Res*., **18**, 1851–1858.

Li,R. *et al.* (2009) SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, **25**, 1966–1967.

Manavski,S.A. and Valle,G. (2008) CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, **9** (Suppl. 2), S10.

Needleman,S.B. and Wunsch,C.D. (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, **48**, 443–453.

Rognes,T. and Seeberg,E. (2000) Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, **16**, 699–706.

Rumble,S. *et al.* (2009) SHRiMP: accurate mapping of short color-space reads. *PLoS Comput. Biol.*, **5**, e1000386.

Smith,T.F. and Waterman,M.S. (1981) Identification of common molecular subsequences. *J. Mol. Biol.*, **147**, 195–197.

Szalkowski,A. *et al.* (2008) SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. *BMC Res. Notes*, **1**, 107.