Bioinformatics Advance Access published May 9, 2012

CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform

Yongchao Liu[1,2,*], Bertil Schmidt[2,*] and Douglas L. Maskell[1]

[1]School of Computer Engineering, Nanyang Technological University, Singapore 639798

[2]Institut für Informatik, Johannes Gutenberg University Mainz, Germany

Associate Editor: Prof. Alfonso Valencia

ABSTRACT

Motivation: New high-throughput sequencing technologies have promoted the production of short reads with dramatically low unit cost. The explosive growth of short read datasets poses a challenge to the mapping of short reads to reference genomes, such as the human genome, in terms of alignment quality and execution speed.

Results: We present CUSHAW, a parallelized short read aligner based on the compute unified device architecture (CUDA) parallel programming model. We exploit CUDA-compatible graphics hardware as accelerators to achieve fast speed. Our algorithm employs a quality-aware bounded search approach based on the Burrows-Wheeler transform (BWT) and the Ferragina Manzini (FM)-index to reduce the search space and achieve high alignment quality. Performance evaluation, using simulated as well as real short read datasets, reveals that our algorithm running on one or two graphics processing units (GPUs) achieves significant speedups in terms of execution time, while yielding comparable or even better alignment quality for paired-end alignments compared to three popular BWT-based aligners: Bowtie, BWA and SOAP2. CUSHAW also delivers competitive performance in terms of SNP calling for an *E.coli* test dataset.

Availability: http://cushaw.sourceforge.net.

Contact: liuy@uni-mainz.de; bertil.schmidt@uni-mainz.de

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

1 INTRODUCTION

The emergence of high-throughput sequencing technologies have led to an explosive growth of the size and availability of short read datasets. It is therefore important and challenging to efficiently and effectively map the sheer volume of short reads to genomes as large as the human genome for existing short read aligners.

Existing aligners are mainly based on two approaches: hash tables and suffix/prefix tries. The aligners based on hash tables either hash the short reads or the reference genome. By hashing short reads, aligners including RMAP (Smith *et al.*, 2008), MAQ (Li *et al.*, 2008) SHRiMP (Rumble *et al.*, 2009) usually have a flexible memory footprint but may have the overhead introduced by scanning the whole genome when few reads are aligned. By hashing the reference genome, aligners such as SOAP (Li *et al.*, 2008), PASS (Campagna *et al.*, 2009) and BFAST (Homer *et al.*, 2009) can be efficiently parallelized using multiple threads, but require large amounts of memory to construct an index for the reference genome.

The aligners based on suffix/prefix tries essentially perform inexact matching based on exact matching. Compared to hash tables, the advantage of the use of tries is that only one search pass is needed to find the alignments to multiple identical copies of a substring in the reference genome, whereas the hash tables need to perform one search pass for each copy. The search using suffix/prefix tries must rely on some algorithms and data structures, one such combination is the Burrows-Wheeler transform (BWT) (Burrows and Wheeler, 1994) and the FM-index (Ferragina and Manzini, 2000, 2005). Several short read aligners have been developed based on the BWT and the FM-index, including the popular Bowtie (Langmead *et al.*, 2009), BWA (Li and Durbin, 2009) and SOAP2 (Li *et al.*, 2009). These three aligners provide support for two types of alignments: seeded alignment, where the high-quality end of a read is used as the seed, and end-to-end alignment. Moreover, paired-end mapping is usually also supported. Bowtie does not support gapped alignment, SOAP2 and BWA supports both ungapped and gapped alignments. The three aligners have been parallelized using multi-threading and are optimized for multi-core CPUs. Even though these aligners are fast, they are still time-consuming for large-scale re-sequencing applications.

The emerging many-core GPUs have evolved to be a powerful choice, in addition to multi-core CPUs and other accelerators like Cell/BE, for the high-performance computing world. Their compute power has been demonstrated to reduce the execution time in a range of demanding bioinformatics applications including sequence alignment (Liu *et al.*, 2009; Vouzis and Sahinidis, 2010), motif discovery (Liu *et al.*, 2010; Kuttippurathu *et al.*, 2011) and short read error correction (Liu *et al.*, 2011). These successes have motivated us to employ GPU computing to accelerate short read alignment.

As a first step, Blom *et al.* (2011) implemented a short read aligner based on the hash table approach using GPUs. However, this aligner can only work for small-scale microbial genomes and does not support paired-end mapping. In this paper, we present

*To whom correspondence should be addressed.

CUSHAW, a parallelized algorithm to align short reads to large genomes, such as the human genome. It employs a quality-aware bounded search approach based on the BWT and the FM-index, and exploits CUDA-compatible GPUs to accelerate the alignment process. The performance of our algorithm is compared with three aligners: Bowtie, BWA and SOAP2, using both simulated and real short read datasets. Our experimental results show that our algorithm achieves significant speedups in terms of execution time compared to the three aligners, while giving comparable or even better alignment quality for paired-end alignments. Furthermore, we have evaluated the performance of SNP calling from short read alignments using the different aligners.

2 METHODS

2.1 Burrows-Wheeler transform

Given a text defined over an alphabet, a BWT is a reversible permutation of the text. For a genome sequence G defined over Σ={A, C, G, T}, the forward BWT of G can be constructed in three steps. Firstly, a special character \$, which is lexicographically smaller than any character in Σ, is appended to the end of G to form a new sequence G\$. Secondly, a conceptual matrix M is constructed, whose rows are all cyclic rotations of G\$ (equivalent to all suffixes of G) sorted in lexicographical order, and each column is a permutation of G\$. Finally, the transformed text L (i.e. the forward BWT of G) is formed by taking the last column of M. A suffix array SA , where $SA[i]$ stores the starting position of the i-th smallest suffix of G, can be constructed from M using the one-to-one correspondence relationship between $SA[i]$ and the i-th row of M (Ferragina and Manzini, 2005; Grossi and Vitter, 2005).

The matrix M has a property called "last-to-first column mapping", which means that the i-th occurrence of a character in the last column corresponds to the i-th occurrence of the same character in the first column. This property constitutes the basis of string searching using BWT. A reverse BWT is constructed from the reverse orientation (not the reverse complement) of G and has the same properties as the forward BWT. A reverse BWT has its own SA.

BWT construction has been extensively studied, including techniques to reduce the execution time and the peak memory size of the working space. We use the algorithm proposed by Hon *et al.* (2007), which only takes $|G|$ bits of working space to construct the BWT (with a peak memory size of < 1GB for the human genome). Since BWT construction is a pre-processing step which can be performed offline, we do not discuss it further. Readers can refer to Lam *et al.* (2008) for more information. In CUSHAW, we have used parts of the source code from the open-source BWT-SW (Lam *et al*., 2008) and BWA algorithms.

2.2 Searching for exact matches

Given a sequence S that is a substring of G, each occurrence of S can be found using a backward search procedure based on the FM-index (Ferragina and Manzini, 2005). Because all suffixes of G that have S as a prefix are sorted together, the search for all occurrences of S is actually equivalent to the search for an interval in SA. We define $C(\bullet)$ to denote an array of length $|\Sigma|$, where $C(x)$ denotes the number of characters in G that are lexicographically smaller than $x \in \Sigma$, and define $Occ(x, i)$ as the number of occurrences of x in $L[0,i]$. Thus, the SA interval for all occurrences of S in G can be recursively calculated, using the forward BWT from the rightmost to the leftmost of S, as

$$\begin{cases} R_s(i) = C(S[i]) + Occ(S[i], R_s(i+1) - 1) + 1, & 0 \le i < |S| \\ R_e(i) = C(S[i]) + Occ(S[i], R_e(i+1)), & 0 \le i < |S| \end{cases} \quad (1)$$

where $R_s(i)$ and $R_e(i)$ are the starting and end indices of the SA interval for the suffix of S starting at position i, and R_s ($|S|$) and $R_e(|S|)$ are initialized as 0 and $|G|$ respectively. The calculation stops if it encounters $R_s(i+1) > R_e(i+1)$, and the condition $R_s(i) \le R_e(i)$ stands if and only if the suffix of S starting at position i is a substring of G. The total number of the occurrences of S in G is calculated as $R_e(0) - R_s(0) + 1$ if $R_s(0) \le R_e(0)$, and 0, otherwise. We can also perform the backward search using the reverse BWT with the difference that it calculates the SA interval from the leftmost to the rightmost of S.

The calculation in Equation (1) only relies on the occurrence array Occ, not requiring L. However, it requires up to $4|G|\lceil \log_2 |G| \rceil$ bits for storing Occ in memory, as the total number of elements is $4|G|$ and each element takes $\lceil \log_2 |G| \rceil$ bits. To reduce the memory footprint of Occ, we trade off the execution speed and memory space using a reduced occurrence array ($ROcc$), which only stores parts of elements in Occ and calculates the others with the help of L. In CUSHAW, $ROcc$ stores the elements whose indices in Occ are multiples of K (K=128 by default) in memory. After using 2 bits to represent each character in L, the total memory size can be reduced to $4|G|\lceil \log_2 |G| \rceil / K + 2|G|$ bits, which is significantly smaller than the whole occurrence array. The memory reduction is very important for our GPU implementation, due to the limited device memory and small cache sizes. For clarity, we define *bwt* to denote the combination of $ROcc$ and L from the forward BWT and define *rbwt* to denote the combination from the reverse BWT.

2.3 Searching for inexact matches

Inexact matches to a genome are indispensable in the consideration of sequencing errors and genuine differences between sequenced organisms and reference genomes. The inexact matches can be obtained by introducing substitutions, insertions and deletions in the query and the subject sequences.

In this paper, our algorithm only supports ungapped alignment, i.e. CUSHAW does not allow insertions and deletions. Thus, the search for inexact matches can be transformed to the search for exact matches of all permutations of all possible bases at all positions of a short read. All the permutations can be represented by a complete 4-ary tree (see Figure S1 in the supplementary data), where each permutation corresponds to a path routing from the root to a leaf (the root node is Ø, meaning an empty string). Each node in the path corresponds to a base in a read that has the same position. Hence, all inexact matches can be found by traversing all paths using either depth-first search (DFS) or breadth-first search (BFS) approaches. BFS requires a large amount of memory to store the results of all valid nodes at the same depth. However, the approach is infeasible for GPU computing, since we need to launch hundreds of thousands of threads to leverage the compute power of GPUs and thus can only assign a small amount of device memory to each thread. Instead, our aligner uses the DFS approach, whose memory consumption is small and directly proportional to the depth of the tree (i.e. the full length of a read). Even though recursive functions are supported for the Fermi architecture (NVIDIA, 2010), we use a stack framework to implement the DFS approach.

2.4 Locating the occurrences

After getting the SA interval, the position of each occurrence in G can be determined by directly looking up the SA. However, it will take $|G|\lceil \log_2 |G| \rceil$ bits if the entire SA is loaded into memory. This large memory consumption is prohibitive for large genomes (e.g. for the human genome, the SA takes about 12GB RAM). Fortunately, we can reconstruct the entire SA from parts of it. Ferragina and Manzini (2005) have shown

that an unknown value $SA[i]$, can be re-established from a known $SA[j]$ using Equation (2).

$$\begin{cases} SA[i] = SA[j] + t \\ j = \beta^{(t)}(i) \end{cases} \quad (2)$$

where $\beta^{(t)}(i)$ means repeatedly applying the function $\beta(i)$ t times. The function employs last-to-first column mapping for the i-th row of M and is calculated as

$$\beta(i) = C(L[i]) + Occ(L[i], i) \quad (3)$$

In Equation (2), t is actually the distance between the two starting positions (i.e. $SA[i]$ and $SA[j]$) in G. Thus, for any unknown i-th element of SA, we can calculate its value $SA[i]$ in at most k iterations if we store the SA elements whose values are multiples of a constant number k (i.e. the starting positions in G are multiples of k). However, this will complicate the storage of SA by introducing additional data structures, making it difficult to map to GPU memory. Instead, we construct a reduced suffix array (RSA) by simply storing $SA[i]$ whose index i is a multiple of k (k=32 by default). This approach reduces the total memory size of a suffix array to $|G|\lceil \log_2 |G| \rceil / k$ bits, but cannot guarantee to complete each computation within k iterations. This is because a maximal distance k between i and j does not mean a maximal distance k for starting positions $SA[i]$ and $SA[j]$ in G. The selection of k is a trade-off between look-up time and memory space. For a suffix array index i that is a not multiple of k, we repeat t iterations using Equation (3) until j is a multiple of k, where $SA[j]$ is equal to $RSA[j/k]$, and then calculate $SA[i]$ as $SA[j]+t$ following Equation (2).

2.5 Mapping using CUDA

To design an efficient short read alignment algorithm using CUDA, it is critical to fully understand the underlying hardware architecture and programming constraints. More than a computing architecture, CUDA is also a parallel programming language that extends the general programming languages, such as C/C++ and Fortran, with a minimalist set of abstractions to express parallelisms. A CUDA program is comprised of code for the host and kernels for the devices. A kernel is a program launched over a set of lightweight parallel threads on GPUs, where the threads are organized into a grid of thread blocks. All threads in a thread block are split into small groups of 32 parallel threads, called warps, for execution. These warps are scheduled in a single instruction, multiple thread fashion. Full efficiency and performance can be obtained when all threads in a warp execute the same code path. It is the programmers' responsibility to limit the amount of thread divergence.

A CUDA-enabled GPU is built around a fully configurable scalable processor array, organized into a set of streaming multiprocessors (SMs) using two different architectures: Tesla architecture (Lindholm *et al.*, 2008) and Fermi architecture (NVIDIA, 2010). For the Tesla architecture, the number of SMs per GPU device varies from generation to generation, and each SM comprises 8 scalar processors (SPs), sharing 8KB or 16 KB of 32-bit registers depending on compute capabilities and a fixed 16KB of on-chip shared memory. The local and global memory is not cached and the amount of local memory size per thread is 16 KB. For the newer Fermi architecture, each GPU device contains 16 SMs with each SM comprising 32 SPs, where each SM has a total number 32 KB of 32-bit registers and has a configurable shared memory size from the 64 KB on-chip memory. This on-chip memory can be configured, for each kernel at runtime, as 48KB of shared memory with 16 KB L1 cache or as 16 KB of shared memory with 48 KB L1 cache. The Fermi architecture has a larger local memory size of 512 KB per thread than the Tesla architecture. Furthermore, a L1 cache of configurable size per SM and a unified L2 cache of size 768 KB per device is introduced for local and global memory caching. The L1/L2 cache hierarchy offers the potential to significantly improve the performance compared to the direct access to the external device memory. The global memory caching in the L1 cache can be disabled at compile time, whereas the local memory caching in L1 cannot be disabled. Thus, it is useful to find out the best combination for a given kernel: 16KB or 48 KB per-SM L1 cache (vice versa for shared memory) with or without global memory caching in L1 and with more or less usage of local memory. Kernels that use a large amount of local and global memory might be able to benefit from the 48 KB L1 cache.

CUSHAW is optimized for the Fermi architecture, where a thread is assigned to align a short read as well as its reverse complement to the reference genome. Highest performance is achieved by using 64 threads per thread block. To alleviate the device memory pressure, CUSHAW organizes the input short reads into batches, and employs multiple passes to complete the computation, where each pass processes a batch of reads. The read batch is stored in texture memory from linear memory at start-up time, and then is loaded into shared memory when performing alignments since we see slight performance improvement after using shared memory. This performance gain comes from a new feature of shared memory in the Fermi architecture. More flexible mechanisms are supported to avoid bank conflicts when two or more threads access any bytes in the same 32-bit bank of shared memory. For read access, it broadcasts multiple words to the requesting threads in a single transaction, and for write access, it requires that each byte is written by only one of the threads.

As mentioned earlier, CUSHAW searches for inexact matches by traversing the tree in a DFS way. This traversing is implemented using a stack data structure. It is impractical to store the stack per thread in shared memory even though we configure a shared memory size of 48 KB. In this case, we implement the stack in the cached local memory for each thread. For the alignment of a read, both *bwt* and *rbwt* are used for the calculation. They are frequently accessed and do not show good data locality when performing alignments. Considering the large amount of device memory consumed by *bwt* and *rbwt* (e.g. the human genome requires about 2.2 GB memory), we store them in cached global memory, instead of cached texture memory. This organization is based on two considerations: (1) we do not need some other features, like address calculations or texture filtering, that can benefit from texture fetches; and (2) L1 cache has a higher bandwidth than the texture cache. Hence, for access to large memory, we can expect a higher performance gain through regular global memory loads cached in L1 compared to texture fetches. After tuning, CUSHAW achieves the best performance when the GPU is configured with a 48 KB per-SM L1 cache with global memory cached in L1.

CUSHAW uses a quality-aware bounded search approach to reduce the search space and guarantee alignment. It supports two types of alignments: seeded alignment and end-to-end alignment. The end-to-end alignment is considered as a special case of the seeded alignment which considers the full length of a read as the seed size. The reverse complement of a read is also incorporated and for clarity, the following discussions only refer to the forward strand. The quality-aware property exploits the numerical quality score of each base in a read. A quality score Q is assumed to be calculated from the probability p that the corresponding base is incorrect, following the PHRED (Ewing and Green, 1998) definition $Q = -10\log_{10} p$. Lower quality scores indicate higher probabilities of incorrect bases. The bounded search is able to reduce the search space by employing several constraints on sums of quality scores and maximal allowable number of mismatches. The use of these constraints can prune branches of the complete tree ahead of time and thus significantly reduce the number of backtracks. The constraints have been used in other aligners, like Bowtie, BWA and SOAP2, in part or whole and are detailed as

- *MMS*: the maximal number of mismatches allowed in the seed (default = 2);
- *MMR*: the maximal number of mismatches allowed in the full length of a read, which is calculated as $MMS + [err \times |S|]$, where *err* is the uniform base error rate (default = 4%).

- *QSS*: the maximal sum of quality scores at all mismatched positions in the seed (default = 70);
- *QSR*: the maximal number of quality scores at all mismatched positions in the full length of a read (default = 3×*QSS*).
- *QSRB*: the maximal *QSR* among the currently selected best alignments, which is updated as the aligning process goes on;

Increasing *MMS* and *MMR* might enable the alignment of more reads but will result in a longer execution time. Decreasing *QSS* and *QSR* focuses the aligner more on mismatches with low-quality scores and can thus facilitate earlier pruning of some branches. However, smaller values also carry the risk of pruning real alignments. Figure 1 shows the workflow of CUSHAW.

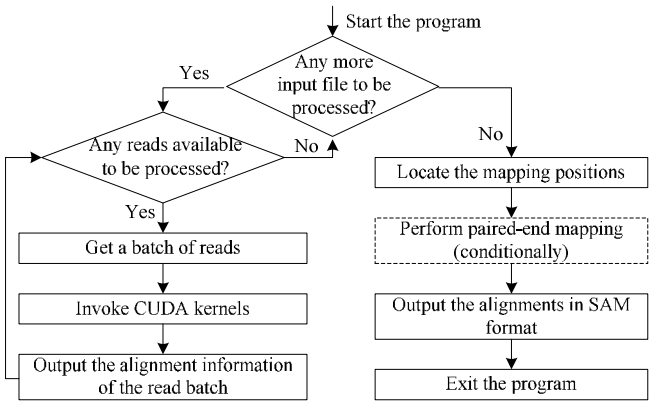

Fig. 1 Program workflow of CUSHAW

CUSHAW outputs the alignments with the smallest quality score sum over all mismatched positions in the full length read alignment. All possible alignments are compared and enumerated by employing the above constraints. Our alignment approach is different from the ones used in Bowtie and BWA. Bowtie allows any number of mismatches in the non-seed region and outputs the "best" alignment after a specified maximum number of search attempts. BWA does not use base quality scores when performing alignments but assigns different penalties on mismatches and gaps. It then reports the alignment with the best score that is calculated from the number of mismatches and gaps in the alignment.

When searching along a path from top to bottom using DFS, we check the quality score constraints: *QSS*, *QSR* and *QSRB* against the current sum of quality scores at all mismatched positions in the path from the root to the current node (including the current node). If the current quality score sum does not satisfy all the three constraints, we will stop searching the subtrees of the current node. Additionally, we must confine the number of mismatches within the allowable ranges. To facilitate the bounded search using the constraints with respect to the number of mismatches, we estimate the lower bound of the number of substitutions that are required to transform the substring, represented by the sequence of nodes in the subpath down from the current node to the leaf, to exactly match to the genome. This estimation has been used in BWA and is calculated based on the observation that if a sequence *S* exactly matches to a genome, any substring of *S* must exactly match to the genome. Hence, the lower bound of the number of substitutions for a string can be estimated by counting the number of all constituent non-overlapping substrings that do not exactly match to the genome. We define a vector *DIFFS*(•) for the full length of *S*, where *DIFFS*(*i*) represents the lower bound of the number of substitutions in the substring $S[i+1, |S|-1]$ ($0 \le i < |S|-1$). When using the seeded alignment with a seed size *W*, we calculate another vector *SDIFFS*(•) for the seed, where *SDIFFS*(*i*) represents the lower bound of the number of substitutions in the substring $S[i+1, W-1]$ ($0 \le i < W-1$). The current node corresponds to a base (mutated or not) at position *i* of *S*, and holds the number of mismatches in the sub-path routing to the root (*UPMM*), including the current node. We will not traverse the sub-trees of the current node if either *UPMM* + *SDIFFS*(*i*) > *MMS* when the current node is in the seed region or *UPMM* + *DIFFS*(*i*) > *MMR* for the full length. Figure 2 shows the pseudocode of the CUDA kernel for the search from the forward-strand of *S*.

```
#bwt: the forward BWT of genome G; rbwt: the reverse BWT of genome G; S: a forward-strand short
#read; W: seed length;

1. estimate the lower bound of the number of substitutions in substring S[i+1, |S|-1] for the full length;
   diffs = 0; f = 0; l = |G|;
   for i=|S|-1 to 0 do
      f = bwt.C(S[i]) + bwt.Occ(S[i], f-1) + 1; l = bwt.C(S[i]) + bwt.Occ(S[i], l); DIFFS[i] = diffs;
      if f > l then
         f = 0; l = |G|; ++diffs;
      fi
   done
2. estimate the lower bound of the number of substitutions in substring S[i+1, W-1] for the seed;
   diffs = 0; f = 0; l = |G|;
   for i=W-1 to 0 do
      f = bwt.C(S[i]) + bwt.Occ(S[i], f-1) + 1; l = bwt.C(S[i]) + bwt.Occ(S[i], l); SDIFFS[i] = diffs;
      if f > l then
         f = 0; l = |G|; ++diffs;
      fi
   done
3. initialize the stack from the first base of S;
   while (stack is not empty) do
      access to the top node (the current node);
      if have attempted all mutations for the current node then
         pop out the top node from the stack;
         continue;
      fi
      mutate the base corresponding to the current node;
      calculate the quality score sum at all mismatched positions down to the current node;
      check whether the constraints QSS, QSR, QSRB, MMS and MMR are satisfied;
      if not all constraints are satisfied then
         pop out the top node from the stack;
         continue;
      fi
      calculate the suffix array interval f and l from the mutation in the current node using rbwt.
      if f <= l then
         if reaching the end of S then
            store the hit and update QSRB;     #a hit is found
         else
            push the next base in S into the stack as well as other information;
         fi
      fi
   done
```

Fig. 2 Pseudocode of the CUDA kernel for the search from the forward strand

To calculate the mapping positions in *G* using *RSA*, an approach is to calculate the *SA* intervals of the best alignments on GPUs and the mapping positions using Equation (2) on CPUs. This approach requires loading an *RSA* and its corresponding *bwt* (or *rbwt*) into memory to improve execution speed, which introduces more memory overhead for the host. Moreover, the calculation of *j* and *t* in Equation (2) imposes more compute overhead, as well, and thus increases the execution time. Fortunately, the calculation of *j* and *t* is independent of the *RSAs* and can be directly computed using the corresponding *bwt* (or *rbwt*) after gaining the *SA* interval. Hence, we select the best alignments, calculate the *j* and *t* values on GPUs and then output the *j* and *t* instead of the *SA* interval. Thus, the mapping positions can be calculated very quickly on the host by only two operations: one table lookup and one addition, after loading the *RSAs* into memory.

As mentioned above, CUSHAW attempts to find the best alignments in the full length read alignment by enumerating and evaluating all possible alignments. Given a read of length *l* and a specific *MMR*, the total number of possible alignments is $\Sigma_{k=0}^{MMR} 3^k C_l^k$, which increases polynomially with *l* (postulating *MMR* is fixed) and increases exponentially with *MMR* (postulating *l* is fixed). Furthermore, to miss as few correct alignments as possible, we have to accordingly increase *MMR* as *l* becomes larger. Even though the search space can be significantly reduced by employing the other four constraints, the number of possible alignments that needs to be evaluated grows significantly for larger *l* and *MMR*. In this context, we have extended the basic version of CUSHAW (as discussed above) by introducing a new progressive constraint approach, with respect to *MMR*, in order to further reduce the search space for long reads. In this extended version, the progressive constraint approach works by employing an additional constraint

on the maximal allowable number of mismatches *MMR′* in the current alignment path of length l' ($1 \le l' \le l$), not only in the full length read alignment, as the alignment progresses. This approach is likely to lower the single-end alignment quality, since it excludes the evaluation of those possibly correct alignments with mismatches clustering in small regions, but does improve the execution speed by further reducing the search space. For each l', its *MMR′* is independent of reads to be aligned, and is pre-calculated before alignment using the following method. We consider the number of base errors m in a read as a random variable and postulate that all bases have the same error probability p (2.5% in our case), for simplicity. Thus, the probability of having k base errors in a sequence of length l' is $P(m=k) = C_{l'}^{k} p^{k} (1-p)^{l'-k}$, where m follows a binomial distribution. In our case, this probability can be approximated using a Poisson distribution with the mean $\lambda = l'p$. For each $l' \le$seed size, its *MMR′* is set to *MMS*; and for each $l' >$seed size, its *MMR′* is set to $max\{MMS+1, min\{k | P(m>k) < err\}\}$, where *err* is the uniform base error rate specified for the input reads. By default, CUSHAW supports a maximal read length of 128 (can be configured up to 256). The *MMR′s* for all l' ($1 \le l' \le 128$) are pre-calculated on the host prior to alignment, and are then loaded into cached constant memory on the GPU devices. CUSHAW releases the basic and the extended versions as two separate executable binaries, and uses the basic version to align <70-bp reads and the extended version to align ≥70-bp reads for all tests in this paper.

2.6 Paired-end mapping

CUSHAW supports paired-end mapping and completes it in three steps for a read pair S_1 and S_2 on the host. Firstly, if both S_1 and S_2 have matches to the reference genome, we iterate each mapping position of S_1 and calculate the distance to each mapping position of S_2. If a distance satisfies the maximal insert size, the read pair is considered paired and the corresponding mapping positions are output, finishing the pairing of S_1 and S_2. Secondly, if S_1 has matches to the genome (no matter whether S_2 has or not), it iterates some mapping positions (at most 2 by default) of S_1 and estimates the region in the genome to the right of S_1, where S_2 is likely to have a match, using the maximal insert size. The Smith-Waterman algorithm (Smith and Waterman, 1998) is used to find the optimal local alignment for S_2 and the genome region. If we find an alignment satisfying the constraints specified by the user, such as the maximal number of unknown bases in the short read and the minimal number of bases in the optimal local alignment for both the short read and the genome region, the read pair is considered paired and otherwise we will continue the pairing process. Thirdly, if S_2 has matches to the genome (no matter whether S_1 has or not), it uses the same pairing method as in the previous step except that the estimated genome region is to the left of S_2. If we still fail to find an optimal local alignment satisfying the constraints specified by the user in this step, the read pair is considered unpaired. The Smith-Waterman algorithm is the most time consuming in the pairing process, having a time complexity of $O(l_1 l_2)$ for a sequence pair with lengths l_1 and l_2 respectively, and thus the more reads that are paired in the first step, the shorter the overall execution time. Moreover, the maximal insert size also has an impact on the number of reads that are paired and the overall execution time. This pairing process has been optimized for multi-core CPUs using a multi-threaded design. Figure 3 shows the workflow of the multi-threaded paired-end mapping. For all paired-end alignments in this paper, CUSHAW uses four CPU threads.

3 RESULTS

We have evaluated the performance of CUSHAW by comparing to three popular short read aligners: Bowtie (version 0.12.7), BWA (version 0.5.9rc1) and SOAP2 (version 2.21) using simulated and real short read datasets (see Table 1). The first three datasets in Table 1 are paired-end datasets simulated from the human genome using the *wgsim* utility program in the SAMtools package (version 0.1.17) (Li *et al.* 2009), with 0.5% SNP mutation rate, 0.05% indel mutation rate and 1% uniform sequence base error rate. The insert size is drawn from a normal distribution $N(200, 10)$ for the SIM36 and SIM72 datasets and from a normal distribution of $N(500, 30)$ for the SIM120 dataset. The simulated datasets are available for download at http://sourceforge.net/projects/cushaw/files/data. The last five datasets are real datasets, named after their accession numbers in the NCBI Sequence Read Archive. All the tests are conducted on a workstation with an AMD Opteron 2378 2.4 GHz quad-core processor and 8 GB RAM running the Linux operating system. Two Fermi-based Tesla C2050 GPUs are installed in the workstation. A single GPU consists of 14 SMs (a total of 448 SPs) with a core frequency of 1.15 GHz and with 3 GB of user available device memory (after turning off error correcting code).

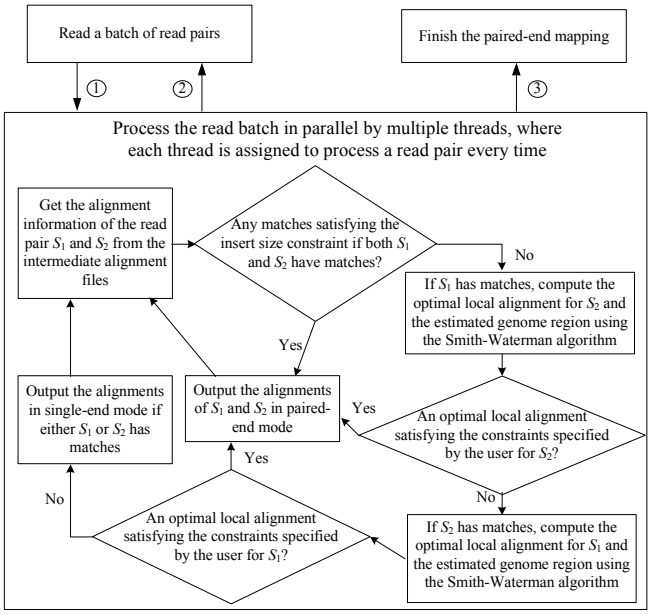

Fig. 3 Workflow of the multi-threaded paired-end mapping

Table 1. Datasets used for performance evaluation

Datasets	Type	Read length	No. of Reads	Insert size
SIM36	PE	36	2,000,012	200 bp
SIM72	PE	72	2,000,012	200 bp
SIM120	PE	120	2,000,012	500 bp
SRR002273	PE	36	8,552,428	200 bp
ERR000589	PE	51	24,277,796	200 bp
SRR033552	PE	75	20,483,110	200 bp
SRR034966	PE	100	41,010,976	500 bp
ERR024139	PE	100	53,542,962	311 bp

3.1 Alignment quality

The alignment quality is conventionally evaluated by computing how many single-end reads are found to have matches to the refer-

ence genome and how many paired-end reads are paired together using simulated or real short read datasets. Since we know the exact positions of each read in the simulated datasets, we introduce two more stringent measures to compare the aligners:

- *MCP*: meaning the number of reads that have a match to their correct positions with a maximal distance of d (d=5 in our evaluation) in the genome;
- *EMCP*: meaning the number of reads that have an exact match to their correct positions in the genome.

The distance d for the MCP measure is considered because of two reasons: (1) a few contiguous low-quality bases starting from the 3' end and a few contiguous leading or trailing unknown bases might be trimmed and (2) gaps might be introduced. In the following evaluations, the single-end alignments consider all aligned reads and the paired-end alignments take into account only the reads that have been paired to calculate the *MCP* and *EMCP*. Thus, the *MCP* and *EMCP* for paired-end alignments might be underestimated compared to when all aligned reads are considered. For all aligners, we have tuned the parameters (see the supplementary data for details) with the intention to gain the highest performance.

We first evaluate all aligners using the three simulated datasets (see Table 2). For both the single-end and paired-end alignments, BWA outperforms all other aligners, while Bowtie performs worst, in terms of all measures. SOAP2 performs slightly better than CUSHAW for the single-end alignment, but the latter outperforms the former for the paired-end alignment (with an exception of the SIM36 dataset). After using paired-end mapping, both CUSHAW and BWA were able to improve *MCP* and *EMCP*. However, both Bowtie and SOAP2 do not always follow this trend. We have also simulated three datasets with a higher uniform base error rate of 4%. The dataset information as well as the alignment results can be obtained from the supplementary data. Using these three datasets, the alignment quality varies much for the aligners, where Bowtie outperforms the other aligners for the single-end alignment and CUSHAW performs best for the paired-end alignment.

Finally, we have assessed the alignment quality (see Table 3) using the five real datasets in Table 1. For the single-end alignment, CUSHAW is inferior to both Bowtie and SOAP2 for all datasets. CUSHAW aligned more reads than BWA for the SRR002273 and ERR000589 datasets, but aligned fewer for the other three datasets. This is due to the progressive constraint approach, which is likely to discard a relevant part of the single-end alignment search space. For the paired-end alignment, CUSHAW performs best and BWA outperforms SOAP2, for all datasets. BWA is superior to Bowtie for the SRR002273, ERR000589, SRR033552 and ERR024139 datasets, and is slightly inferior to it for the SRR034966 dataset. Similar to the results using simulated datasets, the performance of both CUSHAW and BWA improves after using paired-end mapping, but the performance of both Bowtie and SOAP2 deteriorates. For a real dataset, it is difficult for us to prove the correctness of the alignments rescued in the paired-end mapping stage, but we can get some supportive evidences from the paired-end alignments of all simulated datasets to imply the effectiveness of our paired-end mapping policy.

Table 3. Alignment results for real datasets (in percentage)

Datasets	Type	CUSHAW	Bowtie	BWA	SOAP2
SRR002273	SE	92.58	94.69	90.26	94.85
	PE	95.77	87.56	90.85	87.89
ERR000589	SE	94.76	96.51	94.06	96.87
	PE	97.72	92.42	94.60	91.09
SRR033552	SE	88.71	91.70	89.12	92.03
	PE	94.46	86.86	92.35	82.17
SRR034966	SE	78.89	91.25	85.10	85.10
	PE	90.56	90.55	90.51	73.11
ERR024139	SE	89.69	92.09	93.28	92.68
	PE	95.11	87.58	94.46	86.25

3.2 SNP calling

To evaluate the SNP calling performance of the respective aligners, we first aligned the 20.8 million 200bp-insert-size paired-end reads (accession number SRR001665 in NCBI SRA, whose reference genome is *E. coli K12 MG1655* with accession number NC_000913 in GenBank) to a related reference genome *E.coli 536* (accession number NC_008253) using the paired-end alignment for each aligner. Secondly, we called SNPs from the aligned reads using the *mpileup* utility program in SAMtools. Finally, we compared the identified SNPs with the "correct" SNPs of the two genomes. Since the correct SNPs are unknown, we have employed a cross-match approach to predict the correct SNPs, where a SNP is

Table 2. Alignment results for simulated datasets

Type	Aligner	SIM36			SIM72			SIM120		
		Aligned (Paired)	MCP	EMCP	Aligned (Paired)	MCP	EMCP	Aligned (Paired)	MCP	EMCP
SE	CUSHAW	1,822,197	1,596,293	1,595,327	1,791,559	1,705,766	1,704,743	1,705,318	1,649,885	1,648,954
	Bowtie	1,822,191	1,593,327	1,592,366	1,775,714	1,686,975	1,685,982	1,648,097	1,597,055	1,596,288
	BWA	1,823,640	1,598,321	1,597,298	1,827,683	1,740,344	1,739,150	1,819,577	1,764,549	1,763,384
	SOAP2	1,825,550	1,598,291	1,597,316	1,799,550	1,706,894	1,705,859	1,765,747	1,705,692	1,704,724
PE	CUSHAW	1,841,802	1,691,464	1,689,415	1,844,816	1,769,581	1,767,417	1,838,152	1,787,936	1,785,860
	Bowtie	1,793,318	1,676,232	1,674,846	1,705,892	1,648,008	1,647,021	1,469,346	1,436,112	1,435,416
	BWA	1,832,606	1,757,384	1,755,500	1,843,374	1,796,106	1,794,208	1,844,454	1,809,769	1,807,890
	SOAP2	1,793,426	1,700,966	1,699,779	1,740,120	1,685,439	1,684,416	1,578,630	1,541,413	1,540,692

considered to be correct if and only if it has been identified by both the *dnadiff* utility program in MUMmer (version 3.23) (Kurtz *et al*., 2004) and the combination of BWA-SW (Li and Durbin 2010) and SAMtools, through whole genome alignments. The detailed procedure for identifying the "correct" SNPs, as well as the SNP files, is given in the supplementary data.

We evaluated the SNP calling performance of all aligners using the precision, recall and F-score measures (see Table 4). Precision is defined as TP/(TP+FP), recall as TP/(TP+FN) and F-score as 2×precision×recall/(precision+recall), where TP is a true positive, representing a match with a "correct" SNP, FP is a false positive representing a mismatch, and FN represents a "correct" SNP that was not identified. From the single example presented in Table 4, CUSHAW has the best recall, but gives the worst precision, whereas Bowtie yields the best precision, but gives the worst recall. In terms of the F-score, BWA is the best, CUSHAW is second and Bowtie is the worst.

Table 4. SNP calling accuracy comparison

	CUSHAW	Bowtie	BWA	SOAP2
TP	96,980	84,286	96,690	92,343
FP	10,780	5,889	8,611	9,314
FN	2,472	15,166	2,762	7,109
Precision	90.00%	93.47%	91.82%	90.84%
Recall	97.51%	84.75%	97.22%	92.85%
F-score	0.936	0.889	0.944	0.918

3.3 Execution speed

Besides alignment quality, another major concern about short read alignment is the execution speed considering the sheer volume of short reads produced from the high-throughput sequencing technologies. We have compared the speed of CUSHAW on one and two GPUs to the three aligners using multiple threads. In this paper, all the execution times (in seconds) are wall clock times that are taken to complete the whole computation for each aligner.

Table 5 shows the execution speed comparison between aligners for the single-end and paired-end alignments respectively using the five real datasets. The table shows the execution times of CUSHAW using a single GPU and two GPUs, and the execution times of the other three aligners using a single thread and four threads on a quad-core CPU. Each aligner uses the same parameters as in Table 3 with an additional parameter to specify the number of GPUs or threads.

For the single-end alignment, Bowtie runs much faster than BWA and SOAP2 for each dataset. Bowtie on a single CPU core (4 CPU cores) runs faster than CUSHAW on a single GPU (2 GPUs) for SRR034966, but the latter is faster than the former for the SRR002273, ERR000589 and ERR024139 datasets. For SRR033552, CUSHAW runs faster on a single GPU than Bowtie on a single CPU core, while Bowtie on 4 CPU cores is faster than CUSHAW on 2 GPUs. However, CUSHAW significantly outperforms both BWA and SOAP2 for all datasets. Compared to BWA (SOAP2) on a single CPU core, CUSHAW on a single GPU is about 12.0× (7.9×) faster for SRR002273, is about 10.0× (5.4×) faster for ERR000589, is about 6.2× (3.4×) faster for SRR033552, is about 6.4× (6.5×) faster for SRR034966, and is about 6.8× (4.0×) faster for ERR024139. When compared to BWA (SOAP2) on 4 CPU cores, our algorithm on two GPUs is about 6.9× (4.8×) faster for SRR002273, is about 4.8× (2.6×) faster for ERR000589, is about 3.2× (1.8×) faster for SRR033552, is about 3.4× (3.3×) faster for SRR034966, and is about 3.6× (2.1×) faster for ERR024139. From the above analysis, we can see that the speedups of CUSHAW over BWA and SOAP2 generally (not absolutely) decrease as the read lengths increase for the single-end alignment, but they are still considerable even for the two 100-bp datasets. Furthermore, CUSHAW on a single GPU (2 GPUs) achieves nearly stable speedups over BWA on a single CPU core (4 CPU cores) for the three datasets of read lengths ≥75 (i.e. SRR033552, SRR034966 and ERR024139), where the average speedup is about 6.5±0.2 (3.4±0.2).

For the paired-end alignment, CUSHAW achieves significant speedups over all other three aligners (with an exception that for SRR034966, CUSHAW on two GPUs executes only 1.3× faster than Bowtie on 4 CPU cores). On a single GPU (2 GPUs), CUSHAW achieves an average speedup of 5.7 (2.4) with a highest of 11.8 (4.7) over Bowtie, an average speedup of 8.3 (5.5) with a highest of 14.5 (12.2) over BWA, and an average speedup of 8.5 (4.1) with a highest of 24.3 (10.4) over SOAP2, where the three

Table 5. Execution speed comparison between aligners using real datasets

Type	Aligner	SRR002273 (36-bp)		ERR000589 (51-bp)		SRR033552 (75-bp)		SRR034966 (100-bp)		ERR024139 (100-bp)	
		1 core (1 GPU)	4 cores (2 GPUs)	1 core (1 GPU)	4 cores (2 GPUs)	1 core (1 GPU)	4 cores (2 GPUs)	1 core (1 GPU)	4 cores (2 GPUs)	1 core (1 GPU)	4 cores (2 GPUs)
SE	CUSHAW	226	129	1,365	852	3,096	1,704	9,532	5,051	7,077	3,930
	Bowtie	709	249	3,072	944	3,430	940	7,735	2,115	14,379	3,949
	BWA	2,715	896	13,662	4,057	19,055	5,515	60,957	17,308	48,128	14,176
	SOAP2	1,788	625	7,307	2,252	10,470	3,084	62,362	16,815	28,381	8,417
PE	CUSHAW*	292	203	1,700	1,067	3,441	2,004	10,716	6,528	8,179	5,031
	Bowtie	1,271	398	20,017	5,053	15,430	4,023	32,126	8,366	41,177	10,634
	BWA	4,246	2,469	15,713	6,118	20,123	6,569	63,121	19,412	49,989	16,157
	SOAP2	7,098	2,103	15,986	4,862	11,830	3,405	48,525	13,337	29,028	8,528

*CUSHAW uses four threads to perform the paired-end mapping on the CPU.

aligners run on a single CPU core (4 CPU cores) for all five datasets. Similar to the single-end alignment, the speedups of CUSHAW over BWA and SOAP2 generally decrease with the increase of read lengths for the paired-end alignment, but they are also still considerable for the two 100-bp datasets. Moreover, CUSHAW on a single GPU (2 GPUs) also achieves nearly stable speedups over BWA on a single CPU core (4 CPU cores) for the three datasets of read lengths ≥75, with an average speedup of about 5.9±0.1 (3.2±0.1).

As mentioned above, with the increase of read lengths, the number of possible alignments that needs to be evaluated by CUSHAW grows significantly. With the progress in high-throughput sequencing technologies, longer reads (e.g. ≥150 bps) will become more frequent in the future. This will pose challenges to CUSHAW in terms of execution speed, since our aligner attempts to evaluate all possible alignments to find the best alignments by employing some constraints.

4 CONCLUSIONS

We have presented CUSHAW, a parallelized BWT-based short read aligner to the human genome. The algorithm is based on the CUDA C++ parallel programming model and employs CUDA-compatible graphics hardware as accelerators to achieve fast execution speed. It uses a quality-aware bounded search approach to reduce the search space and to guarantee alignment quality. Evaluation using simulated and real short reads reveals that our algorithm is able to achieve significant speedups in execution time over three popular BWT-based aligners: Bowtie, BWA and SOAP2, while producing comparable or even better alignment quality for paired-end alignments. The performance of SNP calling from short read alignments was also examined. While the single example presented is insufficient to fully evaluate the performance of all the aligners, it still sheds some light on the impact of the different aligners in terms of their SNP calling performance.

At present, CUSHAW only supports ungapped alignment for single-end and paired-end reads, where it supports a maximal read length of 128 by default (can be configured up to 256) and a maximal genome length of 4 billion bases. For longer reads that tend to contain indels, the introduction of gapped alignment might be able to increase the probabilities that reads are matched to the reference genome. CUSHAW outputs the aligned (or paired) reads in the SAM format (Li *et al.*, 2009) to take advantage of the SAMtools software package to facilitate the downstream analysis of alignments. The major challenges for short read alignment using CUDA are the frequent accesses to global memory with poor data locality and the divergence of alignment paths for different short reads. The poor data locality will lead to more misses in the L1/L2 caches for global memory accesses and the divergence of alignment paths cause the execution paths of the threads in a warp to diverge frequently. The efficiency of accesses to global memory might be improved by increasing the cache size in the future generation GPU devices, but the divergence of alignment paths for threads in a warp still remains a challenge.

ACKNOWLEDGEMENTS

The authors thank Dr. Liu Weiguo for providing the experimental environments, and thank the editor and the anonymous reviewers for their helpful and constructive comments which helped to improve the manuscript. We acknowledge the BWT-SW authors (T.W. Lam, W.K. Sung, S.L. Tam, C.K. Wong, and S.M. Yiu) and the BWA authors (Heng Li and Richard Durbin) for the use of parts of source code from their open-source BWT-SW and BWA algorithm respectively.

Conflict of interest: none declared

REFERENCES

Blom, J. *et al.* (2011) Exact and complete short read alignment to microbial genomes using GPU programming. *Bioinformatics*, **27**(10), 1351-1358

Burrows, M. and Wheeler, D.J. (1994) A block sorting lossless data compression algorithm. *Technical Report 124 Palo Alto*, CA, Digital Equipment Corporation

Campagna, D. *et al.* (2009) PASS: a program to align short sequences. *Bioinformatics*, **25**(7), 967-968

Ewing, B. and Green, P. (1998) Base-calling of automated sequencer traces using phred. II. Error probabilities. *Genome Res.*, **8**(3), 186-194

Ferragina, P. and Manzini, G. (2000) Opportunistic data structures with applications. *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*

Ferragina, P. and Manzini, G. (2005) Indexing compressed text. *Journal of the ACM*, **52**, 4

Grossi, R. and Vitter, J.S. (2005) Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, **35**, 2

Homer, N. *et al.* (2009) BFAST: an alignment tool for large scale genome resequencing. *PLoS One*, **4**(11), e7767

Hon, W.K. *et al.* (2007) A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays. *Algorithmica*, **48**, 1

Kurtz, S. *et al.* (2004) Versatile and open software for comparing large genomes. *Genome Biol.* **5**, R1

Kuttippurathu, L. *et al.* (2011) CompleteMOTIFs: DNA motif discovery platform for transcription factor binding experiments. *Bioinformatics*, **27**(5), 715-717

Langmead, B. *et al.* (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, **10**(3), R25

Lam, T.W. *et al.* (2008) Compressed indexing and local alignment of DNA. *Bioinformatics*, **24**(6), 791-797

Li, H. *et al.* (2008) Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Res.*, **18**(11), 1851-1858

Li, H. *et al.* (2009) The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, **25**(16), 2078-2079

Li, H. and Durbin, R. (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, **25**(14), 1755-1760

Li, H. and Durbin, R. (2010) Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, **26**(5): 589-595

Li, R. *et al.* (2008) SOAP: short oligonucleotide alignment program. *Bioinformatics*, **24**(5), 713-714

Li, R. *et al.* (2009) SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, **25**(15), 1966-1967

Lindholm, E. *et al.* (2008) NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, **28**(2), 39-55

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

Liu, Y. *et al.* (2010) CUDA-MEME: accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units. *Pattern Recognition Letters*, **31**(14), 2170-2177

Liu, Y. *et al.* (2011) DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI. *BMC Bioinformatics*, **12**, 95

NVIDIA (2009) NVIDIA's next generation CUDA compute architecture: Fermi. *NVIDIA Corporation Whitepaper*

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

Smith, A.D. *et al.* (2008) Using quality scores and longer reads improves accuracy of Solexa read mapping. *BMC Bioinformatics*, **9**, 128

Smith, T.F. and Waterman, M.S. (1998) Identification of common molecular subsequences. *J. Mol.Biol.*, **147**, 195-197

Vouzis, P.D and Sahinidis, N.V. (2010) GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, **27**(2), 182-188