Distrib Parallel Databases (2012) 30:385–399
DOI 10.1007/s10619-012-7099-x

High-performance short sequence alignment with GPU acceleration

Mian Lu · Yuwei Tan · Ge Bai · Qiong Luo

Published online: 10 August 2012

Abstract Sequence alignment is a fundamental task for computational genomics research. We develop G-Aligner, which adopts the GPU as a hardware accelerator to speed up the sequence alignment process. A leading CPU-based alignment tool is based on the Bi-BWT index; however, a direct implementation of this algorithm on the GPU cannot fully utilize the hardware power due to its irregular algorithmic structure. To better utilize the GPU hardware resource, we propose a filtering-verification algorithm employing both the Bi-BWT search and direct matching. We further improve this algorithm on the GPU through various optimizations, e.g., the split of a large kernel, the warp based implementation to avoid user-level synchronization. As a result, G-Aligner outperforms another state-of-the-art GPU-accelerated alignment tools SOAP3 by 1.8–3.5 times for in-memory sequence alignment.

Keywords Sequence alignment · GPGPU · Parallel systems

1 Introduction

The second-generation DNA sequencing devices have been widely used since 2007. They produce short DNA fragments, or short *reads*, at an ultra-high throughput.

Communicated by Judy Qiu and Dennis Gannon.

M. Lu (✉) · Y. Tan · G. Bai · Q. Luo
Hong Kong University of Science and Technology, Hong Kong, China
e-mail: lumian@cse.ust.hk

Y. Tan
e-mail: ytan@cse.ust.hk

G. Bai
e-mail: gbai@cse.ust.hk

Q. Luo
e-mail: luo@cse.ust.hk

Sequence alignment is a fundamental genomics application, where input reads are matched to a reference sequence. Short sequence alignment is challenging due to the large number of input reads. We propose G-Aligner, a short sequence alignment tool accelerated by graphics processors, or GPUs.

A number of short sequence alignment tools have been developed, including 2BWT [4], Bowtie [5] and Bowtie2 [1], BWA [6], SOAP2 [7], SOAP3 [9], WHAM [8], and BarraCUDA [3]. These tools employ various kinds of indexing techniques to speed up the alignment process. Additionally, some of them also take advantages of modern processors, such as multi-core CPUs and GPUs. In comparison with this collection of work, we make the following contributions.

First, we propose a hybrid algorithm employing a filtering-verification framework on the GPU. On the CPU, Bi-directional Burrows-Wheeler Transform (Bi-BWT) [4] based index search is the most efficient algorithm due to its low computation complexity. However, due to its high irregularity in algorithmic structure and memory access pattern, this algorithm cannot fully utilize the GPU hardware resource. Therefore, we combine this Bi-BWT search and the direct matching. The Bi-BWT search first generates matched position candidates, and then the original reads are directly compared with the reference based on the position candidates. As the direct matching is efficient on the GPU, the overall performance is improved significantly.

Second, we carefully optimize the GPU-based implementations through various techniques. Our GPU-based filtering phase is similar to SOAP3, but with more techniques considered, such as the split of large kernels, ordered reads, and the utilization of cached memory. For the direct matching, we implement it based on GPU thread *warps*, to avoid user-level synchronization.

Finally, we conduct fair comparison for G-Aligner and state-of-the-art alignment tools, including 2BWT [4], WHAM [8] and SOAP3 [9]. We consider the both alignment quality and speed. Our evaluation results show that G-Aligner outperforms all existing tools on human genome data sets.

The paper is organized as follows. We introduce the background in Sect. 2. We describe our algorithm and GPU-based implementation in Sect. 3. We perform evaluations in Sect. 4 and conclude in Sect. 5.

2 Background and related work

2.1 Short sequence alignment

A short read is typically tens of base-pair (*bp*) long. For a given reference sequence and a large number of reads, *sequence alignment* is to match each read against the reference. Mismatches are allowed for the alignment, e.g., typically up to two mismatches per read. As one read may be matched to multiple positions on a reference sequence, we call each match an *alignment*. We focus on the alignment for human genome data sets with up to two mismatches. The reference of human genome contains around three billion base pairs.

Popular short sequence alignment algorithms are based on either hashing or Burrows-Wheeler transform (BWT). A hashing-based implementation, such as

WHAM [8], constructs a hash index containing the positions of all subsequences of the reference. In comparison, a BWT index is constructed with all suffixes of the reference and stored in suffix arrays (SA). Alignment tools employing BWT are Bowtie [5] and Bowtie2 [1], BWA [6], 2BWT [4], SOAP2 [7] and SOAP3 [9], and BarraCUDA [3]. Particularly, a significant optimization for BWT is to have a reverse index to support bi-directional search (Bi-BWT), which is adopted in 2BWT, SOAP2, and SOAP3, as well as our G-Aligner.

The hashing-based algorithm is efficient when the number of alignments is small. When there are more alignments, the search space increases linearly and there are more hashing conflicts. Moreover, the hashing index is too large (tens of GBs) to fit into the GPU memory. In contrast, Bi-BWT is more efficient to find all valid alignments. The size of major data structures of the Bi-BWT index is around 2.6 GB, which can fit into the GPU memory. Therefore, we adopt Bi-BWT as our alignment algorithm in this work.

2.2 Burrows-Wheeler transform (BWT) and bi-directional BWT (Bi-BWT)

As this paper does not focus on the detail of BWT algorithm, we only present the algorithm outline that is helpful to understand our filtering-verification framework. We refer readers to the paper of Bi-BWT for details [4].

We first introduce the BWT index, which supports the backward search only. The major index structures of BWT are a suffix array (SA) and an occurrence array. An SA element stores the matched position of a subsequence on the reference sequence. In general, the BWT search works backward on a given short read, and the output is a pair of positions on the suffix array (denoted as l and u, and $l \leq u$). There are n steps of BWT search for a n-bp long short read. It starts from the suffix with size 1. Then each step increases the suffix by 1 and calculates the SA pair for the new suffix with the computation complexity $O(1)$. We abstract this process in Algorithm 1. We ignore the detail of SA computation in the functions *update_l* and *update_u*. After the BWT search, the final matched positions on the reference are collected from $SA[l], SA[l+1], SA[l+2], \ldots, SA[u]$. This step is called *SA conversion*.

For the BWT search with mismatches, for each possible position, it replaces the original base with the other bases (there are four base types in total: A, T, C, G) and continues the BWT search. Two-mismatch search is similar but more complex. Note

Algorithm 1: BWT_exact(string s, int l, int u)

```
   Input: string s[1, 2, ..., n]
   Output: int l, int u
1  l = 0, u = 0, n = s.size()
2  for i = n to 1 do
3      l = update_l(s[i], l), u = update_u(s[i], u)
4      if l > u then
5          return false
6  return true
```

Algorithm 2: **BWT_1MisBackward**(string s, vector⟨pair⟩ sa)

```
   Input: string s[1, 2, ..., n]
   Output: vector⟨pair⟩ sa
1  l = 0, u = 0, n = s.size()
2  if BWT_exact(s[n/2, n/2 + 1, n/2 + 2, ..., n], l, u) = false then
3      return
4  for i = n/2 − 1 to 1 do
5      if l > u then
6          foreach other bases B different from s[i] do
7              l = update_l(B, l), u = update_u(B, u)
8              if l ≤ u and BWT_exact(s[1, 2, ..., i − 1], l, u) then
9                  sa.push_back(l, u)
10     l = update_l(s[i], l), u = update_u(s[i], u)
```

that, the alignment with mismatches allowed may produce multiple SA pairs for an input read.

To improve the performance of BWT search with mismatches, Lam et al. [4] have proposed Bi-Directional BWT (Bi-BWT). A reverse BWT index is constructed to support the forward search. Furthermore, various mismatch positions are categorized into different cases for efficiency. For example, for one-mismatch search, backward and forward searches are used when the mismatch occurs in the left and right half of the input read, respectively. Algorithm 2 outlines the backward one-mismatch search, in which the mismatch occurs in the left half. For two mismatches, the algorithm contains four cases [4]. To simplify the presentation, we refer the four cases as case A, B, C and D.

2.3 Alignment result reporting schemes

Various alignment tools support different result reporting schemes. Considering such a difference is significant when conducting fair performance comparison. If all valid alignments can be found, such a scheme is called *all-valid*. However, some tools have the all-valid option, but in fact they miss alignments due to their design, such as WHAM. We denote such a nominally all-valid scheme all-valid*. Additionally, another useful scheme in practice is *random-best*. For a given read, the best alignments are those that can be matched with fewest mismatches. The random-best scheme reports one alignment from all best alignments for each read. 2BWT [4], WHAM [8] and SOAP3 [9] are used in our evaluations, thus we briefly describe their reporting schemes.

2BWT It can directly supports all-valid and random-best reporting schemes.

WHAM When building the index, WHAM discards the subsequences that appear more than m times (m is 100 by default). Therefore, it only supports all-valid*. Although all valid alignments can be found using a large m, the performance will be

Fig. 1 The example of filtering-verification framework for exact alignment. After the filtering, the matched positions are {3, 7} for the suffix *TC*. Then we can convert the positions to {1, 5} for the read *ATTC*. Finally, the original read is directly compared to the reference based on the positions of 1 and 5

slowed down remarkably. The random-best reporting scheme is not directly supported either. However, the same result can be achieved through combining two parameters.

SOAP3 It can support both *all-valid* and *random-best* reporting schemes.

Our G-Aligner supports all-valid and random-best reporting schemes. We also provide a parameter for users to specify the number of passes to support all-valid*, which can generate similar results as WHAM.

3 Implementation

3.1 Filtering-verification alignment framework

The basic parallelization strategy for Bi-BWT is to make one thread handle one input read. The Bi-BWT search is inherently sequential, thus it is difficult to further parallelize the search for one input read. GPUs are able to run thousands of concurrent threads in the SIMD style to fully utilize the hardware resource. However, the Bi-BWT algorithm contains a large number of branches and is easy to introduce divergences among threads. As a result, the GPU hardware resource will be underutilized. We propose a new filtering-verification framework to better utilize the GPU resource. In the filtering phase, the Bi-BWT search generates SA pairs for suffix or prefix. In the verification phase, SA pairs are converted to matched position candidates, and then the reads are directly compared to the reference based on the position candidates.

Recall that in the BWT search, for each step, we update a SA pair (l and u) for the new suffix. Therefore, if we stop the search at a certain step, we can obtain the matched positions for the current suffix. Suppose the set of matched positions for a given read is A, and the set of matched positions for its suffix is B, obviously we have $A \subseteq B$. In the verification phase, with a given l-bp read and one of its possible matched positions p on the reference R, we directly compare the read with the subsequence $R[p, p+1, p+2, \ldots, p+l)$ through checking every base. Figure 1 demonstrates an example for exact match. For sequence alignment with mismatches, the framework is the same.

Particularly, we have a threshold t for the filtering stage. For a given suffix, when its SA pair satisfies $(u - l) < t$, where l and u are the lower and upper bound of SA for that suffix, we stop the BWT search and record the suffix and its SA pair. This way, we limit the number of candidates for each SA pair smaller than t. Note that,

Algorithm 3: **filter_BWT_exact**(string s, int l, int u, int t)

Input: string $s[1, 2, \ldots, n]$, int t
Output: int l, int u

```
1 l = 0, u = 0, n = s.size()
2 for i = 1 to n do
3     l = update_l(s[i], l), u = update_u(s[i], u)
4     if u − l < t then
5         return true
6     if l > u then
7         return false
8 return true
```

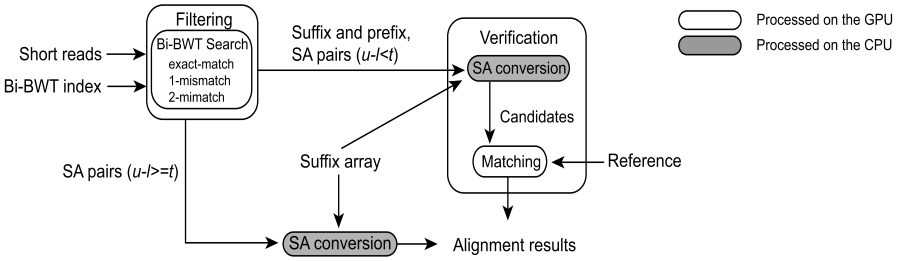


Fig. 2 The components and workflow of the filtering-verification algorithm on the GPU

some reads may finish the entire BWT search with $(u - l) \geq t$. In that case, these SA pairs are not required to be verified. Algorithm 3 shows the modified BWT exact search that works as a filter.

The filtering for the alignment with mismatches is more complex. For example, for the backward one-mismatch search (Algorithm 4), if the SA range becomes smaller than the threshold in the right half of the read using the exact match, the backward search for the entire read stops immediately (line 4–6 in Algorithm 4). In comparison, if it occurs in the left half after an mismatch, the search only stops for that branch (line 13–14 in Algorithm 4). The two-mismatch filtering is more complex, but with the same idea. Note that, for forward search in Bi-BWT, the position candidates are for prefix.

Figure 2 shows the workflow of our filtering-verification framework implemented with the GPU acceleration. Note that, the SA conversion has to be processed on the CPU, as SA is too large (12 GB) to fit into the GPU memory. In addition to SA, the overall GPU memory consumption (the occurrence array, reference, and other buffers) is around 5.6 GB.

Algorithm 4: **filter_BWT_1MisBackward**(string s, vector⟨pair⟩ sa, int t)

```
Input: string s[1, 2, ..., n], int t
Output: vector⟨pair⟩ sa
1  l = 0, u = 0, n = s.size()
2  if filter_BWT_exact(s[n/2, n/2 + 1, n/2 + 2, ..., n], l, u) = false then
3      return
4  if u − l < t then
5      sa.push_back(l,u)
6      return
7  for i = n/2 − 1 to 1 do
8      foreach other bases B different from s[i] do
9          l = update_l(B, l), u = update_u(B, u)
10         if u − l < t then
11             sa.push_back(l,u)
12             continue
13         if l ≤ u and filter_BWT_exact(s[1, 2, ..., i − 1], l, u) then
14             sa.push_back(l, u)
15     l = update_l(s[i], l), u = update_u(s[i], u)
16     if l > u then
17         return
18     if u − l < t then
19         sa.push_back(l,u)
20         return
```

3.2 Discussion of filtering-verification framework

For the filtering threshold t, it is difficult to build an accurate analytic cost model to choose the best value due to the complexity of the algorithm and hardware environment. From our experiments for data sets with various read lengths, the best or close to best performance can be achieved when the threshold is 10. Thus we fix the threshold parameter to 10 by default.

The major issue of pure GPU-based Bi-BWT search is many divergent branches introduced, which lead to imbalanced workload among threads. Our framework essentially reduces the branches in the filtering, and leaves the work for verification without branches. In general, Bi-BWT search for longer reads should introduce more branches. For a given threshold, and two reads with different lengths but a common suffix (suppose filtering will stop on that suffix), the saved Bi-BWT search workload of a longer read is more significant than a shorter read. Therefore, this algorithm should be more effective to improve load balance for longer reads.

For the overall improvement, longer reads should also benefit more from our algorithm. Considering two reads R_1 and R_2, suppose they have a common prefix with length l_c. The length of a shorter read R_1 is $(l + l_c)$, and another R_2 is $(nl + l_c)$, where

$n > 1$. Then we assume the pure Bi-BWT search time on R_1 and R_2 are $(t_1 + t_c)$ and $(nt_1 + t_c)$, respectively, where t_c is the time of search on the common prefix. Suppose we have a suitable threshold that makes the Bi-BWT search stop on the common prefix. Then the filtering time is t_c for both reads. Suppose the verification time on R_1 is t_1', and then it is nt_1' for R_2. In summary, the speedup of our algorithm compared with pure Bi-BWT search for R_1 is $X_1 = \frac{t_1+t_c}{t_1'+t_c}$, and for R_2 is $X_2 = \frac{nt_1+t_c}{nt_1'+t_c}$. As we have $t_1' < t_1$, we can have $X_1 < X_2$ after simplification. This indicates that our algorithm should be more effective for longer reads.

Finally, we focus on the GPU-based implementation for this algorithm in this study. The CPU-based implementation also slightly outperforms the CPU-based Bi-BWT, but the performance advantage is insignificant due to slow verification on the CPU.

3.3 Filtering: GPU-based Bi-BWT search

Our GPU-based Bi-BWT is similar to SOAP3 [9]. We describe the different techniques we have investigated.

Lock-free multi-pass execution This is similar to that adopted in SOAP3. The purpose is to avoid the write conflict when outputting results. The basic idea is to maintain a small buffer for each read. If the number of results of a read exceeds the buffer size, the read will be further processed in the next pass. Furthermore, G-Aligner supports a parameter to specify the number of passes, which is used to support the all-valid* reporting scheme.

Split of large kernel The major purpose of splitting a complex kernel to smaller ones is to reduce register spilling. Recall that the Bi-BWT algorithm has different cases for the alignment with mismatches. If we implement the entire Bi-BWT search algorithm in a large kernel, registers have high pressure. Therefore, we split the large kernel into seven smaller kernels (one for exact match, two for one-mismatch, and four for two-mismatch). This way, each smaller kernel consumes fewer registers to avoid or reduce register spilling.

Ordered reads Branch divergency happens due to different search paths among threads. We propose to arrange similar input reads together through sorting. The overhead of sorting is negligible compared with the total processing time. This way, threads can have the same search path for input reads with the common suffix or prefix. However, due to the randomness of reads, there still may be many divergent branches after sorting.

L1/shared memory configuration On the GPU, the same on-chip memory is used for both L1 cache and shared memory. They can be configured as either 16 KB or 48 KB. For G-Aligner, we make the L1 cache use 48 KB since a larger L1 cache is expected to have a higher hit ratio, and 16 KB is sufficient for the shared memory usage.

Additionally, we have also investigated other techniques, such as the shared memory and texture memory usage, kernel concurrent execution. However, these techniques either have very limited impact or even slightly hurt the performance. Therefore, we do not report them in the paper.

3.4 Verification: CPU-based SA conversion and GPU-based matching

As shown in Fig. 2, in verification, the SA pairs are first converted to the matched positions on the CPU. Particularly, there are four steps for verification: (1) copying SA pairs from the GPU, (2) converting SA pairs to matched position candidates on the CPU, (3) copying candidates to the GPU, (4) direct matching on the GPU.

The direct matching on the reference should be efficient on the GPU. We make one thread *warp* (32 threads) to handle a read. Thus the entire read is accessed by multiple passes, and each pass is for 32 bases. Within each warp, we maintain an array to indicate whether the corresponding position is a mismatch. In the end of each pass, we perform a reduction on the array to calculate the number of mismatches. Note that, the cached *shared memory* is used to hold the indicator array to reduce the memory access latency. As the threads within a warp are automatically synchronized by underlying hardware system, there is no user-level synchronization required in our implementation.

3.5 Alignment result compression

Existing tools store alignments either in customized formats or SAM format [2]. However, various formats essentially record similar information, which include the input read, the matched position, the mismatch information, and the quality score string.

To reduce the disk I/O overhead, we also adopt customized output formats. G-Aligner stores result in a binary file. We focus on compressing reads and quality scores, as these two attributes are the largest ones among all alignment attributes. We notice that for an alignment, if we record the matched position and mismatch information, the read can be recovered according to the reference. As the reference can reside in memory, the read attribute can be eliminated. For the quality score string, those alignments that correspond to the same read have the identical quality score string. Therefore, we only store the unique quality score strings in a table. Furthermore, an extra ID is appended to each alignment to fetch its quality score string from the table.

4 Evaluation

4.1 Experimental setup

We first study the performance of G-Aligner, and then compare it with state-of-the-art alignment tools, including 2BWT [4], WHAM [8] and SOAP3 [9].

Hardware setup We perform the evaluations on a server equipped with an NVIDIA Tesla C2070 GPU and two Intel Xeon E5630 2.53 GHz CPUs (8 cores, 16 threads). The C2070 GPU consists of 448 cores and has 6 GB GPU memory. The server has 32 GB main memory.

Implementation details We develop and evaluate G-Aligner using NVIDIA CUDA C 4.1 in 64-bit Linux system. All evaluations are based on the alignment with up to two mismatches. By default, the filtering threshold t in G-Aligner is fixed to 10, unless otherwise specified.

Data sets The reference sequence is the human genome NCBI 37.1. We choose three real-world data sets. The first one is NCBI SRR003092, and is 51-*bp* long. The second and third data sets are 67-*bp* and 100-*bp* long, respectively, provided by the genome institute BGI-Shenzhen. For each data set, we first randomly sample one million reads for performance studies. Finally, we use the complete 51-bp data set (around 16 million reads) for performance comparison.

Time metric The index loading time (from the disk to the main memory) is excluded in all evaluations. The read input and disk-based result output time will or will not be included, which will be specified in context. The in-memory alignment time of existing tools can be obtained either directly or through modifying the code by ourselves.

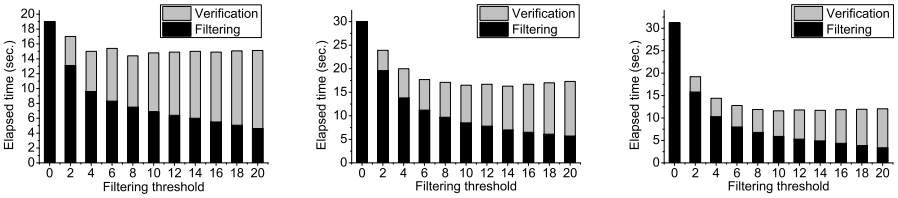
Software for comparison We compare G-Aligner with state-of-the-art alignment tools, including 2BWT [4], WHAM [8] and SOAP3 [9]. Particularly, SOAP3 employs the GPU-CPU coprocessing. 2BWT and WHAM are CPU-based and can support multi-threading. Furthermore, we also compare with a single-threaded CPU-based filtering-verification implementation (denoted as G-Aligner(CPU)), which is developed in-house.

4.2 Performance study of G-aligner

All experiments in this section are based on the all-valid reporting scheme. We focus on in-memory performance studies in this section, which exclude the disk-based read input and result output time.

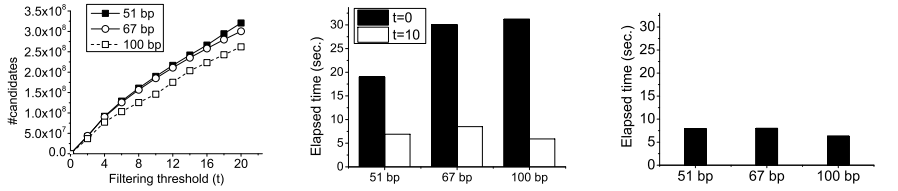
Performance impact of filtering threshold t We vary the filtering threshold t to study its performance impact. Figure 3 shows the overall time as well as the time for the filtering and verification phases with t varied. It shows that the overall time is significantly reduced when t increases from 0 to 6. After that, the elapsed time maintains almost a constant or becomes slightly longer. Note that, when $t = 0$, the filtering can generate all correct results, which is equivalent to the pure Bi-BWT search. As a result, on the GPU, our filtering-verification framework outperforms the pure Bi-BWT search by 1.3X, 1.8X, and 2.7X for the 51-bp, 67-bp, and 100-bp data sets, respectively.

Figure 3 also implies that longer reads take more advantage from our new algorithm. We further study this observation. Figure 4(a) shows that three data sets



(a) 51-*bp* (b) 67-*bp* (c) 100-*bp*

Fig. 3 Elapsed time of filtering and verification with the filtering threshold (t) varied



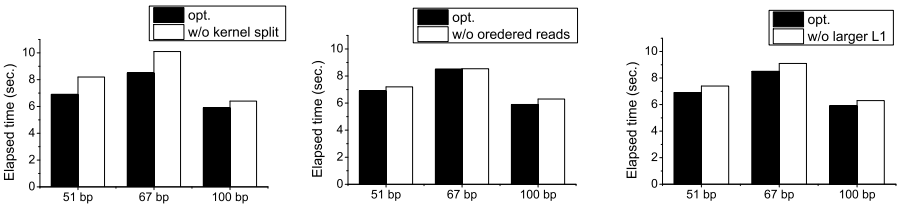
(a) Number of candidates (b) Time of filtering (c) Time of verification($t = 10$)

Fig. 4 Comparison of the filtering-verification framework for various read lengths

produce comparable number of candidates for the same t. Then when t is fixed to 10, Fig. 4(c) shows that the verification takes similar time for three data sets. However, Fig. 4(b) shows that, compared with the pure GPU-based Bi-BWT search ($t = 0$), the speedup of the filtering for 100-*bp* reads is 5.3X, while only 2.8X for 51-*bp* reads. In summary, as the time saving from the filtering is greater than the verification time, and this time saving is greater for longer reads, the overall performance improvement for longer reads is also more impressive. These results confirm the discussion in Sect. 3.2.

Performance impact of filtering-based optimizations We investigate the performance impact from our techniques for the filtering phase. Figure 5(a) shows that through splitting the large kernel, the filtering performance can be improved by up to 26 %. However, either the ordered reads (Fig. 5(b)) or larger L1 configuration (Fig. 5(c)) only slightly improves the filtering performance by up to 8 %. The insignificant improvement is due to the high irregularity of the algorithm structure and memory access pattern.

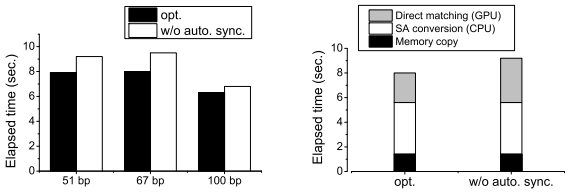
Performance impact of verification-based optimizations Figure 6(a) shows that the warp-based automatic thread synchronization can improve the verification performance by up to 13 %. Recall that the verification phase contains four steps (Sect. 3.4). Figure 6(b) shows that even though the optimization is effective for the GPU-based direct matching (around 1.5X speedup), the overall performance improvement is moderate as this step does not dominate the overall performance.



(a) Large kernel split (b) Ordered reads (c) L1 configuration

Fig. 5 The performance impact of filtering-based optimizations when $t = 10$

Fig. 6 The performance impact of verification-based optimizations when $t = 10$. (**a**) Performance impact of the warp-based synchronization for the verification phase. (**b**) Elapsed time of different steps of verification for 51-bp reads



(a) Warp-based synchronization (b) Time of verification steps (51-bp)

4.3 Comparison with existing software

2BWT, SOAP3 and G-Aligner can produce identical results for both all-valid and random-best schemes. WHAM cannot find all valid alignments. Therefore, we adopt all-valid* for its comparison. In G-Aligner, all-valid* is supported through specifying the number of passes (Sect. 3.3) (denoted as p). In our experiments, p is set to 1, in order to generate similar numbers of alignments as WHAM. Additionally, SOAP3 supports GPU-CPU coprocessing (at least one CPU thread used for coprocessing). As we focus on comparing the GPU-based implementation with SOAP3, we make SOAP3 use one CPU thread for coprocessing. Even though such a comparison is still unfair, it is sufficient to show the efficiency of G-Aligner. In this section, we first focus on the in-memory sequence alignment time. Then we study the end-to-end performance comparison, which includes the disk I/O time.

In-memory sequence alignment: G-Aligner vs. 2BWT and SOAP3 Figure 7(a) shows that for the all-valid scheme, G-Aligner is up to 8 and 2 times faster than the single-threaded and parallel 2BWT, respectively. On the other hand, G-Aligner(CPU) has comparable performance with single-threaded 2BWT. It also shows that SOAP3 employing the pure Bi-BWT search on the GPU only achieves a speedup of around 2-3X compared with single-threaded 2BWT. This confirms our conclusion that the original Bi-BWT algorithm is not suitable for GPU. Instead, our G-Aligner outperforms SOAP3 by up to 3.5 times. This speedup mainly benefits from the filtering-verification framework (as shown in Fig. 3), which reduces the time of GPU-based Bi-BWT search, but utilizes the fast GPU-based direct matching. For the random-best scheme, as only one alignment to be reported for each read, more reads will be processed in the filtering phase only. This may make the speedup less significant than

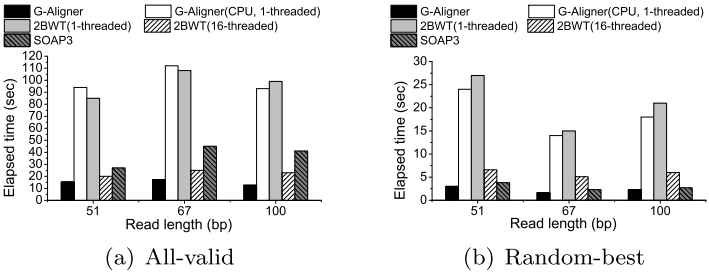


Fig. 7 Performance comparison among G-Aligner, 2BWT, and SOAP3.

The number of alignments (million)

Read length (bp)	50	67	100
G-Aligner (all-valid)	63.6	70.1	28.5
G-Aligner (all-valid*,p=1)	5.9	8.2	3.3
WHAM (all-valid*)	1.7	2.3	2.0
G-Aligner(random-best)	0.40	0.65	0.88
WHAM(random-best)	0.40	0.65	0.88

(a) Alignment results

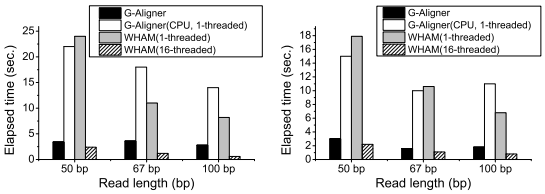


Fig. 8 Comparison between G-Aligner and WHAM

that of all-valid. Nevertheless, Fig. 7(b) shows that G-Aligner outperforms single-threaded and 16-threaded 2BWT by 9X and 2-3X, respectively, and is still faster than SOAP3.

In-memory sequence alignment: G-Aligner vs. WHAM. Figure 8(a) shows the number of alignments generated by G-Aligner and WHAM. It shows that WHAM(all-valid*) only produces 3–7 % of all valid alignments. They are fewer than that generated by G-Aligner(all-valid*) as well. Figures 8(b) and 8(c) show that G-Aligner outperforms single-threaded WHAM significantly in all cases. However, the single-threaded WHAM is faster than our G-Aligner(CPU) for longer reads. As the underlying alignment algorithms of G-Aligner and WHAM are significantly different, we consider WHAM is better for longer reads with the all-valid* scheme. Therefore, the performance improvement of G-Aligner mainly benefits from the GPU acceleration.

End-to-end performance comparison We show the output file size, as it affects the I/O performance. For the software supporting multiple output formats, we choose the format that is smallest. Specifically, 2BWT, SOAP3, and WHAM store their results in binary, plain text, and SAM respectively. Figure 9(a) shows their output sizes. Note that, both 2BWT and SOPA3 require original input read files when extracting complete information of alignments. Instead, our output file already contains all necessary information and does not need the original input file for decompression. Moreover, 2BWT does not record the quality scores, which makes its output file much smaller than SOAP3 and G-Aligner. Although we consider the quality scores are necessary.

Figure 9(b) shows that the parallel 2BWT is slightly faster than G-Aligner due to its smaller output file size. Compared with SOAP3, either the alignment or I/O time

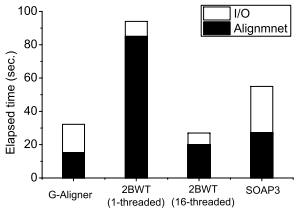
All-valid

G-Aligner	1.1
2BWT	0.3
SOAP3	1.5

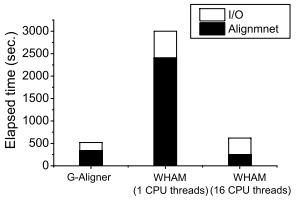
All-valid*

G-Aligner	0.14
WHAM	0.34

(a) Output file size (GB)



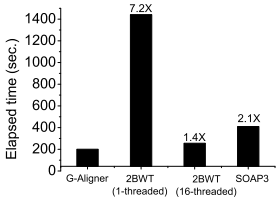
(b) Elapsed time for all-valid



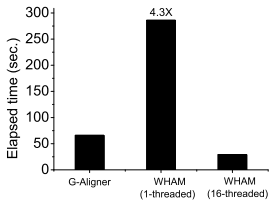
(c) Elapsed time for all-valid*

Fig. 9 End-to-end performance comparison for 51-*bp* reads. (**a**) Output file size. (**b**) Overall elapsed time of 2BWT, SOAP3 and G-Aligner(all-valid). (**c**) Overall elapsed time of WHAM and G-Aligner(all-valid*)

Fig. 10 Performance comparison on the complete 51-bp large data set



(a) All-valid



(b) All-valid*

of G-Aligner is faster than its counterpart of SOAP3. Figure 9(c) shows that due to the GPU acceleration and smaller output size, the overall performance of G-Aligner is faster than either single- or 16-threaded WHAM.

4.4 Performance comparison on a real-world data set

In this section, we evaluate G-Aligner on a complete real-world data set (51-*bp* NCBI SRR003092). This data set contains around 16 million reads, and produces around 1 billion alignments for all-valid. As this paper mainly focuses on in-memory alignment, we only demonstrate in-memory alignment time.

Figure 10(a) shows that for all-valid, G-Aligner outperforms the single-threaded 2BWT by 7.2X and is slightly faster than the 16-threaded 2BWT. Compared with SOAP3, it achieves a speedup of 2.1X. For all-valid* shown in Fig. 10(b), G-Aligner is 4.3X faster than single-threaded WHAM, but is slower than the 16-threaded WHAM. However, we should notice that WHAM only produces around 28 % alignments of that produced by G-Aligner(all-valid*).

5 Conclusion

We present the design and optimization of our GPU-accelerated short DNA sequence alignment tool G-Aligner. G-Aligner is based on a new filtering-verification alignment framework, which combines the traditional Bi-BWT index search and direct matching. Our new algorithm can better utilize the GPU hardware resource than the

pure GPU-based Bi-BWT implementation. We further optimize G-Aligner through various techniques, such as the split of a large kernel and warp-based implementation to avoid user-level synchronization. As a result, on a server equipped with an NVIDIA Tesla C2070 GPU and two Intel Xeon E5630 CPUs, compared with SOAP3, a leading sequence alignment system with GPU acceleration, G-Aligner is up to 3.5 times faster for in-memory alignment. The source code of G-Aligner is available at http://www.cse.ust.hk/gallop.

References

1. Bowtie 2. http://bowtie-bio.sourceforge.net/bowtie2/index.shtml
2. The SMA Format Specification: http://samtools.sourceforge.net/SAM1.pdf (2011)
3. Klus, P., Lam, S., Lyberg, D., Cheung, M.S., Pullan, G., McFarlane, I., Yeo, G., Lam, B.: BarraCUDA—a fast short read sequence aligner using graphics processing units. BMC Res. Notes **5**(1), 27+ (2012)
4. Lam, T.W., Li, R., Tam, A., Wong, S., Wu, E., Yiu, S.M.: High throughput short read alignment via bi-directional bwt. IEEE Int. Conf. Bioinform. Biomed. pp. 31–36 (2009)
5. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. **10**, 3 (2009)
6. Li, H., Durbin, R.: Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics **25**(14), 1754–1760 (2009)
7. Li, R., Yu, C., Li, Y., Lam, T.W.W., Yiu, S.M.M., Kristiansen, K., Wang, J.: SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics **25**(15), 1966–1967 (2009)
8. Li, Y., Terrell, A., Patel, J.: Wham: A high-throughput sequence alignment method. In: ACM SIGMOD (2011)
9. Liu, C.M., Wong, T., Wu, E., Luo, R., Yiu, S.M., Li, Y., Wang, B., Yu, C., Chu, X., Zhao, K., Li, R., Lam, T.W.: SOAP3: Ultra-fast GPU-based parallel alignment tool for short reads. Bioinformatics **28**(6), 878–879 (2012)