

SOAP3: GPU-based Compressed Indexing and Ultra-fast Parallel Alignment of Short Reads*

Chi-Ming Liu, Tak-Wah Lam, Thomas Wong, Edward Wu, Siu-Ming Yiu
Department of Computer Science, University of Hong Kong

Zhiheng Li, Ruibang Luo, Bingqiang Wang, Chang Yu
Beijing Genome Institute, Shenzhen, China

Xiaowen Chu, Kaiyong Zhao
Department of Computer Science, Hong Kong Baptist University

Ruiqiang Li
Novogene Inc. Beijing, China & University of Hong Kong

Abstract

As the cost efficiency of the next generation DNA sequencing technology keeps improving, there is an ever-increasing demand for high-throughput software to align the enormous number of short reads (patterns) with reference genomes (such as the human genome). In the past few years, a number of very fast alignment software (e.g., Maq, SOAP2, ZOOM, Bowtie, BWA) have been developed; most of them are exploiting some kind of compressed index (like BWT) of the reference sequence. These tools can align at a speed of a few hundred seconds per one million reads. Yet such speed is still behind the throughput of the next generation sequencing machines. In this paper we show the first implementation of a compressed index on the GPU. The technical issues include how to reduce the memory accesses to the index from individual threads (cores) and how to control the branching and divergence of the threads to avoid unnecessary idle time. Based on this new index, we are able to exploit the parallelism of GPU to speed up the alignment of short reads. Our experiments show that with respect to the human genome, the GPU takes only a few seconds to perform exact alignment of one million length-100 reads, and tens of seconds when a few mismatches are required. Further improvement has been obtained by utilizing the host CPU to share the load of the GPU concurrently.

1 Introduction

A genome or DNA can be regarded as a long sequence of characters chosen from the alphabet $\{A, C, G, T\}$. For example, the human genome has 3 billion characters and *Allium cepa L.* (Onion) 16 billion. The human genome has attracted the most attention from scientists as it provides the source of studying human genetic diseases. The first human genome was constructed in 2003. It is known that genomes of individuals (even from the same species) are not exactly the same, though very similar. Their differences are often the clues of genetic diseases. The famous 1000 Genomes Project¹ is to reveal and compare the genomic sequences of thousands of individuals from different ethnic groups in order to identify the genetic differences among the groups.

The tasks of constructing the genome of an individual (the process is usually referred as *sequencing*) and comparing genomes of different individuals are not trivial at all. Based on the

*SOAP3 is available for download at <http://soap.genomics.org.cn/soap3.html>.

¹<http://www.1000genomes.org>.

current sequencing technologies (e.g. Illumina Genome Analyzer, Roche 454, and Applied Biosystems SOLiD), in the first (and the most core) step, a sequencing machine can rapidly generate and read out a huge number of substrings of length about 100 at random positions of a given genome. Each substring is called a read or short read. Note that information about the ordering and the positions of these reads in the genome is lost in the process. If a genome of another individual of the same species exists (called the *reference*), one can align (map) these reads to the reference to get a rough sketch of the genome. Since there are sequencing errors in the reads and also variations between individual genomes, the alignment should allow small errors. Based on the alignment results, one can start to reconstruct the genome (called *resequencing* [2, 3, 7, 23]) and identify the differences between the genomes (e.g. [6, 8, 15]). One or two years ago, high-throughput sequencing machines could only generate reads of length 35 to 75, and it was common to consider alignment that allowed up to two mismatches (in the case of human genome, two mismatches would already allow at least 90% of reads to be mapped). But with the longer reads nowadays, we need alignment that can allow up to three or even four mismatches.

Massive data volume: To resequence a human genome with sufficient accuracy, we usually need to cover every character at least 20 times on average, and this translates to about at least 600 millions of reads of length 100. The advancement of sequencing technologies has made it feasible to generate this volume of data in a fast and low-cost manner. A single sequencing machine like Illumina HiSeq 2000 can generate about 200 million reads of length 100 in 10 days. Large genome centers usually have tens to over a hundred of such machines. And 600 million reads can be generated easily in 10 days using three sequencing machines. On one hand, current sequencing technologies have made it possible to generate sufficient reads in a cost-effective manner for many other exciting high-throughput biological applications (e.g. mapping DNA-protein interactions [10], whole-transcriptome sequencing [9], and whole-genome expression profiling [22]). Yet current sequencing technologies also increase the stress on the speed of the core alignment step tremendously. Note that the alignment process is only the first step and to get the final result, there are a few other subsequent computational steps which are also time consuming to be carried out. Thus, an extremely fast alignment tool is necessary.

Existing alignment tools: The literature contains quite a number of software for aligning short reads onto a reference genome. The most popular ones are Maq [16], SOAP2 [19], ZOOM [20], Bowtie [14], and BWA [17]. A very recent comparison of their performance can be found in [4], which reports that SOAP2 is faster than the others. In the most common application where the reference is the human genome, SOAP2 can align one million reads of length 100 with up to two mismatches in 330 seconds (see Table 1 in Section 4). The current version of SOAP2 does not support alignments with three mismatches; nevertheless a straightforward projection would show that more than 2,000 seconds are needed (theoretically speaking, the time complexity is exponential rather than linear to the number of errors). That means, to resequence a human genome as described above, it requires more than 13 days to just complete the alignment step. This is indeed longer than the time needed to generate the reads by the sequencing machines. It is conceivable that the indexing and alignment algorithms of SOAP2 could be further optimized and the time for three mismatches could be reduced to hundreds of seconds. Yet to further reduce the time to tens of seconds seemed impossible before the work of this paper.

Our contributions: The core engine of SOAP2 relies on a compressed full-text indexing data structure called Burrows-Wheeler Transform or BWT (see [1] for a theoretical survey of BWT in the context of pattern matching). To obtain a drastic improvement in speed, we leverage the huge number of multiprocessors in a graphics processing unit (GPU) and develop a GPU version of BWT. The new alignment engine called SOAP3 achieves a speed up of 10, 22, and 40 times over SOAP2 when aligning with one, two, and three mismatches, respectively (see Table 1 in Section 5). In our experiments, aligning one million reads of length 100 onto the human genome with three

mismatches can be done in only 46 seconds (and 15 seconds for 2 mismatches). In other words, 7 hours are needed to align the 600 million reads in the process of resequencing a human genome. More interestingly, SOAP3 can also take advantage of the host CPU to share the load of the GPU on those relatively unstructured alignments, achieving a further speed up. Our experimental results show that this enhanced version of SOAP3 can reduce the time for 3-mismatch alignment to 15 seconds per million reads, and even 4-mismatch alignment can be done in about 40 seconds (see Section 6).

The novelty of SOAP3 stems from the following two aspects. Unlikely many other GPU applications which are usually computation intensive, BWT is a sophisticated compressed indexing data structure, and pattern searching using BWT requires a huge number of random memory access. This causes a lot of difficulties for GPU as the conflicts in memory access among different threads would impair the overall performance drastically. We solve this problem by redesigning part of the data structure of the BWT engine to decrease the number of memory access as much as possible. We also improve the memory access time by coalescing memory accesses. The other difficulty is that the GPU works in a single-instruction multiple-thread (SIMT) mode. Thus, the processors in the same unit (called streaming multiprocessor, SM) must execute the same instruction. Too many diverging branches in the execution path would force some of the processors to idle. However, how many diverging branches a pattern would introduce cannot be determined easily until runtime. Based on our pattern matching application, we derive a useful parameter to determine in runtime whether a pattern would introduce too many diverging branches (referred as *hard* patterns). We stop the execution of those hard patterns, then group them and re-do the matching for them in another round to decrease the idle time of the processors. With these techniques, we are able to achieve a significant saving of the processing time.

Organization of the paper: The rest of the paper is organized as follows. Section 2 contains the problem definition and preliminaries for BWT. The issues of implementing SOAP2 in the GPU will be highlighted in Section 3. The core ideas of SOAP3 are presented in Section 4. Experimental results are discussed in Section 5. Section 6 concludes the paper.

We remark that when allowing up to 3 mismatches in aligning reads of length 100, SOAP3 would only fail to align 5 to 10% of reads to the reference genome due to too many sequencing errors and variations in the genomes. For those reads, we need a more involved alignment (e.g., allowing more errors as well as insert/delete using the Needleman-Wunsch dynamic programming), but this would be very time consuming. Very recently, [4] has provided a GPU-based tool to perform this kind of dynamic-programming alignment, (the speed is slightly slower than the old software SOAP2 for short reference), which complements SOAP3 in the perfect way.

2 Problem definition and preliminaries

The alignment problem is formally defined as follows. Let $T[1..n]$ be a reference sequence (e.g., the human genome). Given a large number (millions to billions) of short patterns (reads) P , we want to locate all substrings of T that is at most at a Hamming distance of δ from each P . Each such substring is said to be an *occurrence* of P . We assume that characters in T and each P are chosen from the alphabet $\Sigma = \{A, C, G, T\}$, each pattern is of length in the range 75–120, and $\delta \leq 3$.

In the following, we give a brief overview on two indexing data structures, namely, suffix arrays and Burrows-Wheeler Transform (BWT) [5].

2.1 Suffix array

Given a text $T[1..n]$, we define the suffix array of T , denoted $SA[1..n]$, as $SA[i] = j$ if the suffix $T[j..n]$ is lexicographically the i -th smallest suffix among all suffices of T (and we say that the

rank of the suffix $T[j..n]$ is i). That is, SA stores the starting positions of all suffixes of T in lexicographical order.

Suppose that a given pattern P appears in T . We define the *the SA range* of P with respect to T as $[s, e]$ such that s and e are respectively the rank of the lexicographically-smallest and largest suffix of T that contains P as a prefix. To find all occurrences of a pattern P in T , we can first compute the SA range of P and retrieve the occurrences of P from the suffix array directly one by one.

Approximate matching and multiple SA ranges: To solve our alignment problem, for a pattern P , one can enumerate all possible strings S with at most a hamming distance of δ from P and perform an exact match of S on T resulting in multiple non-empty SA ranges which give all occurrences of P within a hamming distance of δ .

The suffix array of a text with n characters requires $n \log n$ bits of memory in addition to the text. For example, to store the suffix array of a human genome, it requires 12G memory. Instead of using suffix arrays, we consider the compressed indexing data structure Burrows-Wheeler Transform (BWT).

2.2 Burrows-Wheeler Transform (BWT)

Given a text $T[1..n]$, the BWT data structure, denoted $BWT[1..n]$, is defined as $BWT[i] = T[j-1]$ where $j = SA[i]$ for $SA[i] \neq 1$, otherwise, set $BWT[i] = \$$, where $\$$ is a special character not in Σ and assumed to be lexicographically smaller than all other characters. That is, $BWT[i]$ stores the character immediately before the i -th smallest suffix. Note that BWT requires only the same amount of memory as for storing the text. Using BWT and some auxiliary functions, we can compute efficiently the SA range of a given pattern in a backward manner (searching the pattern from right to left) known as *backward search*. The idea is as follows.

For any character x , let $Count(x)$ be the total number of characters in the text T that are smaller than x . And let $Precede(i, c)$ be the number of character c that occurs in $BWT[1..i-1]$. With these two functions, given the SA range $[s, e]$ of a pattern P , computing the SA range $[s', e']$ for the pattern cP for any character c can be computed based on the following lemma [11, 13].

Lemma 1. *Let P be a pattern and $[s, e]$ be its SA range. Let c be any character, the SA range of cP can be computed as $[Count(c) + Precede(s, c) + 1, Count(c) + Precede(e + 1, c)]$.*

The efficiency of backward search depends largely on how fast $Count$ and $Precede$ can be computed. We can precompute and store up the values of $Count$ and $Precede$ allowing fast retrieval of the values. For $Count$, we can store the values in an array of $|\Sigma|$ entries to allow constant time retrieval. For $Precede$, auxiliary data structures are used. There are several methods proposed, including rank-and-select data structures [21] and sampling (e.g. SOAP2's implementation, to be described later). Given any $P[1..m]$, the SA range of $P[m..m]$ is simply $[Count(P[m]) + 1, Count(c)]$, where c is the character just lexicographically larger than $P[m]$. Then, based on Lemma 1, we can compute the SA range of a pattern using BWT easily.

Based on [12], for efficient approximate matching, we also need to support *forward search*, i.e., searching the pattern from left to right and switching from backward search to forward search or vice versa so that we can start the matching from the middle of a pattern, then extend to either direction. The key to accelerating these operations is to support the simultaneous retrieval of $Precede(i, c)$ for every character c efficiently, given a fixed i .

3 Issues of implementing SOAP2 in GPU

3.1 GPU and CUDA

In a GPU, there are several streaming multiprocessors (called SMs). In each SM, there are dozens of processors. Each SM operates in a single-instruction multiple-thread (SIMT) mode, i.e. all processors in a SM must execute the same instruction, but possibly on different data. Thus, if the data in different processors in the same SM needs to be processed in different execution paths (diverging branches), some processors in the SM will sit idle. A common trick in GPU implementation is to eliminate as many diverging branches as possible from the algorithm.

Multiple threads are executed in a GPU as follows. Threads are grouped into *blocks*, and blocks are organized into *grids*. The GPU schedules each block onto a specific SM. The maximum number of threads that can be run on a SM in parallel is called a *warp* (typically 32). A block can contain multiple warps. We use NVIDIA's GPU card and CUDA² in our implementation. The code to be executed in GPU is contained in special C functions called *kernels*. When the host (CPU) passes execution to the GPU, multiple copies of the called kernel is executed in parallel by multiple GPU threads.

There are several types of memory in a GPU. We only make use of the *global memory* which is the largest in size (up to gigabytes) and also used for the CPU and the GPU to share data. Note that the access time of global memory is affected by the memory access pattern. The design of the GPU does not favor random access of memory as conflicts in memory access of different SMs may degrade the overall performance due to memory contention. Most of the other applications in the GPU are computation intensive and each thread will access a different region of the global memory. However, for SOAP3, the application is memory-access intensive and the access pattern is a bit random. It thus increases the difficulty of achieving significant speed up by porting the CPU version to the GPU.

3.2 The issues

The memory access problem: Since BWT-based pattern searching involves a lot of random access to the index and the auxiliary data structures (for the retrieval of *Precede* values), this is not desirable in the GPU. In particular, the sampling technique used in SOAP2 for the auxiliary data structures require a lot of memory accesses.

Let us first briefly describe the sampling approach. Storing all values of *Precede* allows constant time retrieval, but requires $|\Sigma||T|$ words of space, which is approximately 48 GB for the human genome. The space requirement can be reduced by sampling, at the cost of a slightly increased retrieval time. Let $k > 1$ be a sampling ratio. For each character c in the alphabet, we store $Precede(K, c)$ for each K that is a multiple of k . To compute $Precede(i, c)$ for any given i and c , we first look up the sampled array for $Precede(i', c)$ where i' is the multiple of k nearest to i . If $i' < i$, we count the number of character c in $BWT[i'..i - 1]$ and add it to $Precede(i', c)$; otherwise, if $i' > i$, we count the number of character c in $BWT[i..i' - 1]$ and subtract it from $Precede(i', c)$. The goal is to choose a sampling ratio k small enough so that counting characters in *BWT* can be done efficiently, while large enough so that the array can fit into memory.

SOAP2 uses two-level sampled arrays. The first-level array is obtained by sampling every 65536th entry of the *Precede* function for each character. These samples partition the *Precede* function into segments of size 65536. For each of these segments, a second-level array is constructed by sampling at every 256th entry. The difference between any second-level sample and its nearest first-level sample is at most 65536, hence each second-level sample can be represented using only $\log_2 65536 = 16$ bits. For the human genome, the auxiliary arrays require less than 0.1 GB.

²CUDA is the computing engine developed by the company NVIDIA which specializes in GPU cards.

Retrieving a *Precede* value takes two 32-bit memory accesses to the auxiliary arrays and one 256-bit memory access to *BWT*, which may be further split into multiple accesses if the system does not natively support 256-bit memory access. Since GPU global memory accesses are rather expensive, a direct implementation of this sampling technique is not desirable.

The branching problem: The branching problem is a well-known problem in GPU which is sometimes unavoidable. Even worse, how diverging the branches can be may depend on the data to be processed. It may, as in our application, only be known during runtime execution. The branches introduced by a pattern may force many processors in the same SM to idle. How to reduce the impact of this problem is not trivial. In the next section, we will describe how we monitor this situation in a runtime manner based on a parameter due to our application and how to tackle the problems of having these difficult cases of data.

4 Core ideas in SOAP3

4.1 Reducing memory accesses

To reduce the memory accesses for retrieving *Precede*, we redesign the auxiliary data structure to use a simple one-level sampling instead of two-level sample. By choosing a suitable sampling ratio, we achieve good balance between the space requirement and the number of memory accesses.

In SOAP3’s implementation of BWT, the auxiliary array is built by sampling every 128th entry of the *Precede* function for each character. The space required is approximately 0.375 GB for the human genome. With the new design, retrieving a *Precede* value only takes one 32-bit memory access to the auxiliary array and one 128-bit memory access to *BWT*, which is half of that in the SOAP2 implementation. The sampling ratio of 128 is specifically chosen to restrict memory accesses to 128-bit, which is the maximum width of memory read/write operations supported by a single GPU instruction. The 128-bit data is read as a vector of two 64-bit words, in order to exploit the GPU’s native support of 64-bit bitwise operations, including *popcount*. *Population count* (popcount) is a procedure for counting the number of 1’s in a bit string. For any given sequence of 32 characters, represented as a 64-bit bit string, popcount can be applied to count the occurrences of a certain character, after a few bitwise operations. On the latest GPUs (as well as many CPUs), popcount is implemented as a single instruction with very few instruction cycles.

Recall that to support efficient approximate matching, we need to support the retrieval of $Precede(i, c)$ for all character c efficiently. The new auxiliary array also speeds up this operation. Instead of accessing the auxiliary array four times, we only need to access it once. This is done by grouping *Precede* values with the same position i together into a contiguous 128-bit segment, which can then be retrieved by a single 128-bit memory read instruction. Hence forward search requires as many memory accesses as backward search.

To provide a rough idea how much speed up we gain, we consider the time for executing backward search and forward search which are the critical memory operations in the application. The SOAP3 version takes 6 ns and 6.1 ns for backward and forward search per character respectively³. This is a great improvement compared to 24.3 ns and 26.4 ns for backward and forward search respectively in SOAP2.

4.2 Coalescing memory accesses

The access time of the global memory is very sensitive to memory access patterns. Memory accesses that are *coalesced* within a warp are generally faster than those that are not. For memory accesses to be coalesced within a warp, they need to access nearby locations in the memory (e.g., within

³Based on the setting of using 8192 blocks and 128 threads per block.

a 128-byte segment), with no two threads in the warp accessing the same memory location. An example situation would be thread 1 accessing the first word in a 128-byte segment, thread 2 accessing the second word, and so on.

The reads (patterns to be aligned) are loaded into the global memory and accessed frequently during alignment. In SOAP3, the reads are stored in the global memory in a specific way to allow coalesced memory accesses. Precisely, the reads are partitioned into groups of 32 (the warp size). For each group, the reads are arranged as follows. Let $w_{i,j}$ denote the j -th word of the i -th read in the group ($1 \leq i \leq 32$). We store the words in the global memory in this order: $w_{1,1}, w_{2,1}, \dots, w_{32,1}, w_{1,2}, w_{2,2}, \dots, w_{32,2}, w_{1,3}, w_{2,3}, \dots$. When the threads in a warp simultaneously access, say, the first words of the reads (i.e., $w_{1,1}, \dots, w_{32,1}$), the memory locations accessed form a contiguous 128-byte segment. These 32 memory accesses are coalesced and done in a single memory transaction. This way a high memory throughput can be achieved.

4.3 Reducing branching effect

We reduce the branching effect by the idea of early termination for patterns that introduce a lot of branches. As mentioned earlier, GPU threads in a warp (precisely, a half-warp) must execute the same instruction. Stalling will occur if threads in the same warp take different execution paths. Therefore divergent branches should be avoided.

From now on, we refer to the CPU as the host and the GPU as the device. In approximate matching, a pattern can generate more than one SA range during alignment. The processing time for patterns with more SA ranges tends to be longer. These “hard” patterns, if mixed with other patterns with few SA ranges, may have a negative impact on the elapsed time of the warp. It is therefore beneficial to separate hard patterns and other patterns into different warps. However, it is not obvious how to identify hard patterns until they are being aligned, so we take a different approach. We estimate the hardness of a pattern by the number of SA ranges it generates. To prevent hard patterns from dominating the elapsed time of the warp, we terminate the execution of any pattern as soon as it generates more than a threshold τ_1 of SA ranges. These hard patterns are then collected and processed again by the device. In the second round of alignment, more space is allocated to each pattern since they are known to generate many SA ranges. If some patterns still generate more than a threshold $\tau_2 > \tau_1$ of SA ranges in the second round, they are terminated and later passed into the device for the third round of alignment. Alternatively, these “very hard” patterns can be handled by the host. Experiments showed that early termination of hard patterns improves the performance significantly. For instance, using early termination with thresholds $\tau_1 = 8$ and $\tau_2 = 2048$, aligning one million length-100 reads with at most 1 mismatch takes 3.8657 seconds, whereas without early termination it takes 9.7895 seconds.

4.4 Splitting kernels

Experiments showed that separating a big kernel into two or more kernels could reduce the total elapsed time. Our conjecture is that big kernels cause thread synchronization and scheduling issues, thus affecting the instruction throughput. The alignment algorithm benefits from this observation. Take 2-mismatch alignment as an example. Given a pattern, we divide it into three parts from left to right. There are four cases depending on the positions of the two mismatches. (A) Both mismatches occur in the first two parts. (B) Both mismatches occur in the last part. (C) One mismatch occurs in the second part, the other in the last part. (D) One mismatch occurs in the first part, the other in the last part. The pattern is independently processed by four kernels, each taking care of one of the cases. In our experiments, the total elapsed time turns out to be shorter than that when handling all four cases in one kernel.

4.5 The overall framework

The BWT index, along with the redesigned auxiliary arrays, is loaded into both the host and device memory. In the case of human genome, this occupies a total of 2.25 GB, which fits well into any GPU with at least 3 GB of video memory. On the other hand, the suffix array, which occupies 12 GB, is only loaded into the host memory. For systems with little host memory, a sampled suffix array [12] can be used instead (for example, a 4-sampled SA only requires 3 GB).

The device is mainly responsible for computing SA ranges for the patterns. The computed SA ranges are then sent back to the host for retrieval of the positions using the suffix array. Retrieving positions on device side is avoided since a single SA range can be translated to thousands of positions, and sending these positions back to the host would be a waste of resources.

5 Experimental results

SOAP3 was implemented using CUDA and C++. Testing was performed on a 2.8 GHz 4-core machine with 16 GB main memory (but only one core was used). In the machine, a GPU card (model: NVIDIA Tesla C2070) was installed with 6 GB global memory, 7 SMs and 48 processors in each SM. In the experiment, we selected one million length-100 human reads for evaluation, which was a real dataset provided by Beijing Genome Institute at Shenzhen (BGI). To understand the effectiveness of SOAP3, the program was used to search all the reads along the whole human genome (build 36 version 3) downloaded from NCBI, and compared with the running time of SOAP2, which was shown to be the fastest among the existing alignment tools [4]. SOAP2 was also executed on the same machine and only one core was used. The number of mismatches allowed for alignment was from 0 to 3, and all the valid alignments were reported in the experiment. Table 1 shows the summary on the comparison of the time required by SOAP3 and SOAP2. As shown in the table, SOAP3 requires total 1.85 seconds to perform exact-match alignment for one million length-100 reads while SOAP2 requires 17.02 seconds. For 1-mismatch and 2-mismatch alignments, SOAP3 requires 3.87 and 14.92 seconds, while SOAP2 requires 42.04 and 329.58 seconds respectively. SOAP2 does not support alignments with 3 mismatches but the projected time is over 2000 seconds, while SOAP3 requires 45.82 seconds. The speed up of SOAP3 compared with SOAP2 ranges from 9 to 40.

Type	SOAP3			SOAP2	Speed-up ratio
	Loading of reads	Alignment + output	Total		
Exact	1.42	0.97	2.39	17.02	7.12
1-mismatch	1.41	2.46	3.87	42.04	10.88
2-mismatch	1.41	13.51	14.92	329.58	22.10
3-mismatch	1.43	44.39	45.82	-	-

Table 1: Time required (in seconds) for aligning one million reads (of length 100) using SOAP3 and SOAP2. SOAP2 does not support alignments with 3 or more mismatches. The projected time required for SOAP2 to align one million length-100 reads is over 2,000 seconds.

To further examine the performance of SOAP3 on different lengths of reads, we also repeated the experiment using two other sets of reads of length 75 and 120. Each set also consisted of one million human reads. The length-75 reads were obtained from NCBI short read archive⁴ and the length-120 reads were simulated using wgsim (part of SamTools [18]) with default parameters

⁴The length-75 reads were downloaded from NCBI Short Read Archive #SRR013647

except that the length is set to 120. Table 2 shows the summary on the time required by SOAP3 and SOAP2 for length-75 and length-120 reads. As shown in the table, the speed up of SOAP3 for aligning one million length-75 human reads along the whole human genome compared with SOAP2 is around 10 to 15, while the speed up for length-120 reads is 5 to 28. Note that the drop in the speed up for exact alignment of length-120 reads is because the time required for loading the reads (which is 1.64 seconds) becomes more dominant and cannot be further reduced, although the time required for alignment and answer reporting (which is only 0.44 seconds) has been reduced a lot compared to SOAP2. The above experiment results show that by making use of the GPU, the alignment engine of SOAP2 can be greatly enhanced and the new engine SOAP3 can perform more efficiently.

Type	Length-75 reads			Length-120 reads		
	SOAP3	SOAP2	Speed-up ratio	SOAP3	SOAP2	Speed-up ratio
Exact	2.56	26.89	10.51	2.07	11.49	5.55
1-mismatch	4.83	64.50	13.35	3.52	26.37	7.48
2-mismatch	27.60	436.13	15.80	10.54	298.18	28.29
3-mismatch	118.68	-	-	25.91	-	-

Table 2: Time required (in seconds) for SOAP3 and SOAP2 to align one million length-75 human reads and one million length-120 human reads along whole human genome.

On the other hand, when the read length increases, we need to consider alignment with up to 4 mismatches. We have also enhanced SOAP3 to support 4 mismatches; our experiment shows SOAP3 takes 95.85 seconds to align one million length-120 reads with up to four mismatches.

6 Further improvement of SOAP3: Concurrent CPU and GPU processing

In the current implementation, the CPU idles when the GPU is running a kernel. Since each kernel call takes up to a few seconds, it would be a waste of the CPU if we simply let the CPU idle. One way to better utilize the CPU is to let it share some of the load of the GPU concurrently. According to the CUDA specifications, CPU and GPU threads can be executed concurrently as long as there is no memory operation between the host and the device. Therefore, we can have a pipeline workflow as follows. While the CPU is handling hard patterns and positions output, the GPU can work on the alignment of the next batch of patterns.

Care must be taken when deciding the workload of the CPU and GPU to avoid stalling among the threads. It is difficult to balance their workload as GPU tends to overwhelmingly outperforms CPU, even if we allow the latter is using a quad-core or hexa-core processor. Nevertheless, the CPU is more suitable to handle those exceptional cases where the reads have a high number of hits and the alignment process is rather unstructured. Technically speaking, we make use of the thresholds τ_1 and τ_2 to balance the workload. For instance, using the thresholds $\tau_1 = 4$ and $\tau_2 = 256$, the average time for GPU to finish its two rounds of 3-mismatch alignment on one million length-100 reads is 36.46 seconds, while the average time for CPU to handle the hard reads is 15.31 seconds. On the other hand, setting $\tau_1 = 4$ and $\tau_2 = 64$, the time figures have become 28.48 seconds and 26.00 seconds. It is obvious that the latter is more desirable for the pipelining.

We have tested this pipeline version of SOAP3 using a quad-core CPU. Three CPU threads are working concurrently with the GPU. We use a data set with 25.1 million reads of length 100 to perform the experiment. As shown in Table 3, the alignment time of SOAP3 has improved

drastically. For the case of three mismatches, the average alignment time per million reads is about 15 seconds, (recall that the non-pipeline version as shown in Table 1 is more than 44 seconds). More importantly, even when up to 4 mismatches are required, the average alignment time is no more than 40 seconds per million reads.

Type	SOAP3 with pipelining		
	Total time for 25.1M reads		Average time per million reads
	Loading of reads	Alignment + Output	Alignment + Output
Exact	43.63	15.24	0.61
1-mismatch		48.77	1.94
2-mismatch		235.53	9.38
3-mismatch		360.56	14.36
4-mismatch		980.31	39.05

Table 3: Time required (in seconds) by SOAP3 with pipelining to find all alignments of 25.1 million length-100 human reads with respect to the human genome.

7 Future work

Our current implementation only uses the global memory which is the slowest among all available memory in a GPU card. How to make use of other types of faster memory to speed up the alignment process or other similar memory intensive applications in GPU is still an open issue.

References

- [1] Donald Adjeroh, Tim Bell, and Amar Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 2008.
- [2] David R. Bentley, Shankar Balasubramanian, and Harold P. Swerdlow et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59, 2008.
- [3] D.R. Bentley. Whole-genome re-sequencing. *Curr. Opin. Genet. Dev.*, 16:545–552, 2006.
- [4] Jochen Blom, Tobias Jakobi, Daniel Doppmeier, Jorn Kalinowski, Jens Stoye, and Alexander Goesmann. Exact and complete short read alignment to microbial genomes using gpu programming. *Bioinformatics*, 2011.
- [5] M. Burrow and D.J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, California, 1994.
- [6] Ken Chen, John W Wallis, Michael D McLellan, David E Larson, Joelle M Kalicki, Craig S Pohl, Sean D McGrath, Michael C Wendl, Qunyuan Zhang, Devin P Locke, Xiaoqi Shi, Robert S Fulton, Timothy J Ley, Richard K Wilson, Li Ding, and Elaine R Mardis. Break-dancer: an algorithm for high-resolution mapping of genomic structural variation. *Nature Methods*, 6:677–681, 2009.
- [7] L.W. Hillier, Gabor Marth, Aaron R. Quinlan, and David Dooling et al. Whole-genome sequencing and variant discovery in *C. elegans*. *Nature Methods*, 5:183–188, 2008.

- [8] F. Hormozdiari, C. Alkan, E.E. Eichler, and S.C. Sahinalp. Combinatorial algorithms for structural variation detection in high-throughput sequenced genomes. *Genome Research*, 19(7):1270–1278, 2009.
- [9] T. Jarvie and T. Harkins. Transcriptome sequencing with the Genome Sequencer FLX system. *Nature Methods*, 5, 2008.
- [10] D.S. Johnson, A. Mortazavi, R.M. Myers, and B. Wold. Genome-wide mapping of in vivo protein-DNA interactions. *Science*, 316(5830):1497–1502, 2007.
- [11] Ming-Yang Kao, editor. *Encyclopedia of Algorithms*. Springer, 2008.
- [12] T.W. Lam, Ruiqiang Li, Alan Tam, Simon Wong, Edward Wu, and S.M. Yiu. High throughput short read alignment via bi-directional BWT. In *Proceedings of IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2009)*, pages 31–36, 2009.
- [13] T.W. Lam, W.K. Sung, S.L. Tam, C.K. Wong, and S.M. Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, 2008.
- [14] B. Langmead, C. Trapnell, M. Pop, and S.L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(R25), 2009.
- [15] S. Lee, F. Hormozdiari, C. Alkan, and M. Brudno. Modil: detecting small indels from clone-end sequencing with mixtures of distributions. *Nature Methods*, 6(7):473–474, 2009.
- [16] H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18:1851–1858, 2008.
- [17] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.
- [18] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Gonçalo R. Abecasis, and Richard Durbin. The sequence alignment/map format and SAM-tools. *Bioinformatics*, 25(16):2078–2079, 2009.
- [19] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.
- [20] H. Lin, Z. Zhang, M.Q. Zhang, B. Ma, and M. Li. ZOOM! Zillions of oligos mapped. *Bioinformatics*, 24(21):2431–2437, 2008.
- [21] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.
- [22] G. Robertson, M. Hirst, M. Bainbridge, M. Bilenky, Y. Zhao, T. Zeng, G. Euskirchen, B. Bernier, R. Varhol, A. Delaney, N. Thiessen, O.L. Griffith, A. He, M. Marra, M. Snyder, and S. Jones. Genome-wide profiles of STAT1 DNA association using chromatin immunoprecipitation and massively parallel sequencing. *Nature Methods*, 4:651–657, 2007.
- [23] Jun Wang, Wei Wang, and Ruiqiang Li et al. The diploid genome sequence of an Asian individual. *Nature*, 456(7218):60–65, 2008.