2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

Evaluation of GPU-based Seed Generation for Computational Genomics using Burrows-Wheeler transform

Yongchao Liu, Bertil Schmidt
Institut für Informatik
Johannes Gutenberg Universität Mainz
Mainz, Germany
e-mail: {liuy, bertil.schmidt}@uni-mainz.de

Abstract— **Unprecedented production of short reads from the new high-throughput sequencers has posed challenges to align short reads to reference genomes with high sensitivity and high speed. Many CPU-based short read aligners have been developed to address this challenge. Among them, one popular approach is the seed-and-extend heuristic. For this heuristic, the first and foremost step is to generate seeds between the input reads and the reference genome, where hash tables are the most frequently used data structure. However, hash tables are memory-consuming, making it not well-suited to memory-stringent many-core architectures, like GPUs, even though they usually have a nearly constant query time complexity. The Burrows-Wheeler transform (BWT) provides a memory-efficient alternative, which has the drawback of having query time complexity as a function of query length. In this paper, we investigate GPU-based fixed-length seed generation for computational genomics based on the BWT and Ferragina Manzini (FM)-index, where *k*-mers from the reads are searched against a reference genome (indexed using BWT) to find *k*-mer matches (i.e. seeds). In addition to exact matches, mismatches are allowed at any position within a seed, different from spaced seeds that only allow mismatches at predefined positions. By evaluating the relative performance of our GPU version to an equivalent CPU version, we intend to provide some useful guidance for the development of GPU-based seed generators for aligners based on the seed-and-extend paradigm.**

Seed generation; seed-and-extend; CUDA; GPU; Burrows-Wheeler transform

I. INTRODUCTION

The rapid progress of high-throughput sequencing technologies has enabled the production of short reads at an unprecedented scale. This high-throughput production has significantly revolutionized the scale and resolution of many biological applications, such as methylation patterns profiling [1], protein-DNA interactions mapping [2], and differentially expressed genome identification [3]. All these applications require aligning large quantities of short reads to the human genome or the genomes of other species. However, the high throughput of second-generation sequencing technologies comes with shorter read lengths and higher error rates, compared to the conventional Sanger shotgun sequencing. Moreover, a large number of short reads are usually produced to achieve highly redundant coverage of a genome. Thus, conventional sequence aligners, optimized for capillary reads, have exposed their inefficiency, in tackling the flood of short reads, with respect to time efficiency and alignment accuracy [4]. Recently, a few aligners have been developed for short read alignment, which are different from the earlier-generation general aligners like BLAST [5] and BLAT [6]. The earlier-generation aligners are generally designed to find homologous sequences by searching through biological sequence databases, whereas the new-generation short read aligners are generally employed to align short reads, from the species of interest, to the reference genomes of that species or other species. This subtle difference has certain impact on sequence aligners with respect to design methodology and performance [7]. The new-generation aligners are mainly based on two techniques: hash tables and suffix/prefix tries.

Essentially, all hash-table-based aligners follow the seed-and-extend paradigm [12]. The first and foremost step of the seed-and-extend heuristic is to generate seeds between the query sequence and the target sequence, represented as short matches indicating highly similar regions. Subsequently, these seeds are extended and refined to obtain the final alignments using more sophisticated algorithms (e.g. Needleman-Wunsch algorithm [13] or Smith-Waterman algorithm [14]). Several kinds of seeds have been proposed, including fixed-length seeds [5], variable-length seeds [15], maximal unique matches [16], rare exact matches [17] and adaptive seeds [18]. In this paper, we will concentrate our research on the widely used fixed-length seeds. For fixed-length seeds, short seeds can improve sensitivity but are likely to result in longer runtime during the subsequent time-consuming alignment extensions, due to the larger number of seeds found. Long seeds are rarely matched but have the risk of decreasing sensitivity. Fixed-length seeds are represented as *k*-mer matches, where the simplest case is the exact *k*-mer match. Some improvements have been suggested to allow mismatches and gaps in the seeds, including spaced seeds [19] and *q*-gram filters [9]. Spaced seeds allow mismatches to occur only at predefined positions in different templates that are specifically tuned for a reference genome with some sensitivity tolerance. *q*-gram filters allow gaps in the seeds in addition to mismatches. To identify seeds, two approaches can be used: hash tables and the Burrows-Wheeler transform (BWT) [10]. The hash tables are advantageous in terms of fast query time but have a larger memory footprint. The BWT, on the contrary, has a smaller memory footprint but

978-0-7695-4676-6/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPSW.2012.85

has a query time complexity in a function of the seed length. Since the current-generation graphics processing units (GPUs) have limited device memory, we select the memory-efficient BWT approach to generate seeds (i.e. find *k*-mer matches).

Since the advent of compute unified device architecture (CUDA)-enabled GPUs, they have evolved to be a powerful choice, for the high-performance computing community. Their compute power has been demonstrated to reduce the runtime in a range of demanding bioinformatics applications, including sequence database search [20] [21] [22], multiple sequence alignment [23], motif discovery [24] and applications relating to new-generation high-throughput sequencing such as short read error correction [25] and short read alignment [26] [27] [28]. These successes have motivated us to investigate and to evaluate seed generation on CUDA-enabled GPUs for genomics. In this paper, we investigate a GPU-based fixed-length seed generator for genomics based on the BWT and FM-index, where seeds are identified by searching *k*-mer matches of short reads against the reference genome indexed using the BWT. In addition to exact matches, we allow mismatches to occur at any position of a *k*-mer. By evaluating the relative performance of our GPU version to an equivalent CPU version, this paper intends to provide some preliminary guidance to the future development of GPU-based seed generators for aligners using the seed-and-extend heuristic.

II. BACKGROUND

A. Fermi GPU Architecture

A CUDA-enabled GPU is a fully configurable scalable processor (SP) array, organized into a set of streaming multi-processors (SMs) based on two architectures: earlier-generation Tesla architecture [29] and newer-generation Fermi architecture [30]. Since our seed generator is designed and optimized for the Fermi-based GPUs, we will only describe some features of the Fermi architecture.

Each Fermi-based GPU device contains 16 SMs with each SM comprising 32 SPs. 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 at runtime for each CUDA kernel. A CUDA kernel can be configured to use either 48KB of shared memory with 16 KB L1 cache or 16 KB of shared memory with 48 KB L1 cache. The Fermi architecture has a local memory size of 512 KB per thread, which is larger than the Tesla architecture with a local memory size of 16 KB per thread. Furthermore, the Fermi architecture introduces local and global memory caching through a L1 cache of configurable size per SM and a unified L2 cache of size 768 KB per device. This L1/L2 cache hierarchy offers the potential to significantly improve the access performance to the external device memory compared to direct accesses. Programmers can choose to disable the global memory caching in the L1 cache at compile time, but are not able to disable the local memory caching in L1. Thus, for a CUDA kernel, it is of great significance to find out the best combination: 16 KB or 48 KB L1 cache (vice versa for shared memory) with or without global memory caching in L1 and with more or less usage of local memory. CUDA kernels that have more local and global memory accesses might be able to benefit from the 48 KB L1 cache.

B. Suffix Array and Burrows-Wheeler Transform

Given a genome sequence *G*, defined over the alphabet Σ={A, C, G, T}, the suffix array *SA* of *G* stores the starting positions of all suffixes of *G* in lexicographical order. In other words, $SA[i] = j$ means that the i^{th} lexicographically smallest suffix (among all suffixes of *G*) starts at position *j* in *G*. Thus, given a substring *S* of *G*, we can find all of its occurrences within an *SA* interval, i.e. an index range $[I_a, I_b]$, where I_a and I_b represent the indices in *SA* of the lexicographically smallest and largest suffixes of *G* with *S* as the prefix. Figure 1 shows the construction of an example suffix array.

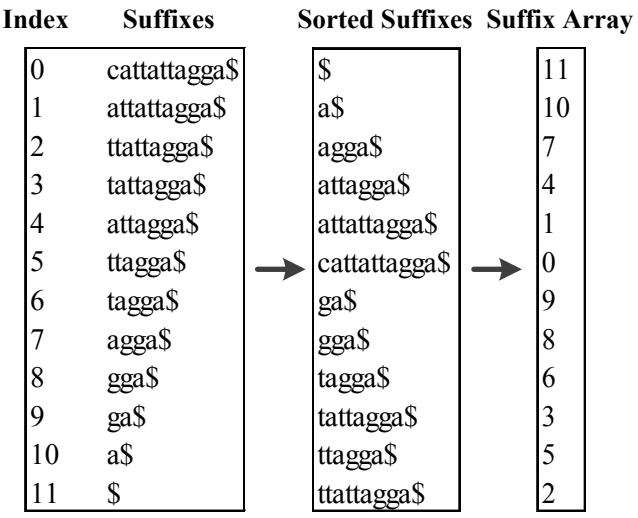

Figure 1. An example suffix array for the sequenc *G*=cattattagga

The forward BWT of *G* can be constructed in the following three steps. We first append a special character $, which is lexicographically smaller than any character in Σ, to the end of *G* to form a new sequence *G*$. Then, we construct a conceptual matrix M_G whose rows are all cyclic rotations of *G*$ (equivalent to all suffixes of *G*) sorted in lexicographical order. In M_G, each column forms a permutation of *G*$. Finally, we take the last column of M_G to form the transformed text *B*, i.e. the forward BWT of *G*. Figure 2 shows the construction of an example BWT. From the construction procedure, we can see that the i^{th} entry in *SA* has a one-to-one correspondence relationship with the i^{th} row of M_G [11].

The matrix M_G 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 forms the basis of pattern search using BWT. Similarly, a reverse BWT of *G* can also be constructed from the reverse orientation (not the reverse complement). The reverse BWT shares the same properties as the forward BWT, and has its own *SA*. For the

convenience of discussion, we define the following denotations:

- $C(\bullet)$: an array of length $|\Sigma|$, where $C(x)$ represents the number of characters in G that are lexicographically smaller than $x \in \Sigma$;
- $Occ(\bullet)$:the occurrence array, where $Occ(x, i)$ represents the number of occurrences of x in $B[0,i]$.

For a genome sequence, we usually construct the BWTs once and then store them on disk for the use of sequence alignment. In this paper, the algorithm devised by Hon et al. [31] is used to construct the BWTs of G, which requires $|G|$ bits of working space. This algorithm has also been used in BWT-SW [32] and BWA [33].

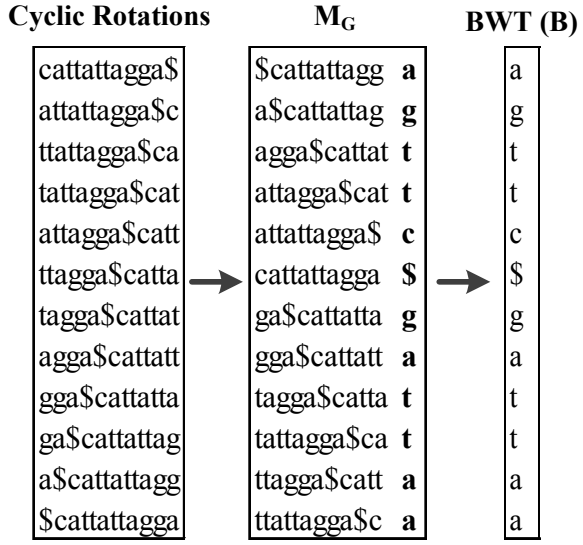

Figure 2. An example Burrows-Wheeler transfrom for the sequence G=cattattagga

C. Backward Search using FM-index

Given a substring S of G, we can find all the occurrences of S using a backward search procedure based on the FM-index [11], which employs the arrays $C(\bullet)$ and $Occ(\bullet)$ to compute the *SA* interval of S. Thus, using the forward BWT, the *SA* interval can be recursively calculated, from the rightmost to the leftmost suffixes of S, as

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

where $I_a(i)$ and $I_b(i)$ represent the starting and end indices of the *SA* interval for the suffix of S starting at position i, and $I_a(|S|)$ and $I_b(|S|)$ are initialized as 0 and $|G|$ respectively. The calculation stops if it encounters $I_a(i+1)>I_b(i+1)$, and the condition $I_a(i)\le I_b(i)$ stands if and only if the suffix of S starting at position i is a substring of G. The total number of the occurrences is calculated as $I_a(0)-I_b(0)+1$ if $I_a(0)\le I_b(0)$, 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 prefixes of S. Figure 3 shows an example of backward search to calculate the *SA* interval.

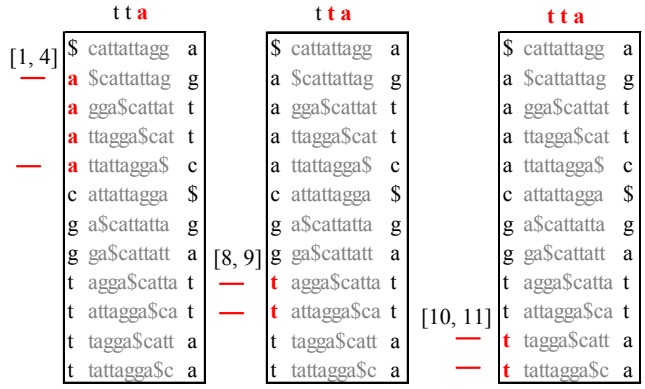

Figure 3. Example of backward search to calculate the *SA* interval for a substring "tta"

III. Methods

A. BWT Memory Reduction

From Equation (1), the *SA* interval calculation only relies on the occurrence array *Occ*, not requiring B. Thus, we can compute all occurrences of S on GPUs after loading *Occ* into the device memory of GPUs. Since the current-generation GPUs have very limited device memory (≤6 GB), it is critical to estimate the memory requirement of *Occ* beforehand. For the genome sequence G, its *Occ* array has $4|G|$ elements. Assume that each element takes $\lceil \log_2 |G| \rceil$ bits, it requires up to $4|G|\lceil \log_2 |G| \rceil$ bits. This memory overhead is quite considerable for large genomes. For example, the human genome (about 3 billion bases) requires about 47 GB to store its *Occ*, far more than the available device memory.

To reduce the memory footprint of *Occ*, we introduce a reduced occurrence array (*ROcc*) that only stores parts of the elements in *Occ* and calculates the others with the help of B at runtime. In this paper, *ROcc* stores the elements whose indices in *Occ* are multiple of q (default=128) to trade off the execution speed and memory space. In this case, the total memory size is the sum of *ROcc* and B. Using 2 bits to represent each character in B, the total memory size can be reduced to $4|G|\lceil \log_2 |G| \rceil / q + 2|G|$ bits (i.e. about 1.1 GB for the human genome). For clarity, we define *bwt* to denote the combination of *ROcc* and B from the forward BWT and define *rbwt* to denote the combination from the reverse BWT.

In this paper, both *bwt* and *rbwt* are employed to generate seeds on the GPU and thus need to be loaded into the GPU device, where the overall memory footprint is about 2.2 GB for the human genome. Considering that we only need to load them once from the host to the device through the high-throughput peripheral component interconnect express

channels, the data transfer time is very small and thus can be neglected in our case.

B. BWT Data Deployment

When performing search, it requires frequent random accesses to the BWT data, without showing good data locality. Considering the large amount of device memory consumed by the BWT data for large genomes, we store the BWT data in cached global memory instead of cached texture memory. This deployment of the BWT data is based on the following two considerations. On one hand, we do not need some other features, such as address calculations and texture filter, which can benefit from texture fetches. On the other hand, L1 cache has a higher bandwidth than texture cache. Therefore, for random accesses to large memory, we can expect a higher performance gain through regular global memory loads cached in L1 than texture fetches. In general, the larger L1 cache size, the better memory access performance. Hence, we configure the GPU with a 48 KB per-SM L1 cache with global memory cached in L1. For the array $C(\bullet)$, it only has four integer elements and is stored in cached constant memory.

C. Locating Occurrences using a Suffix Array

After getting the *SA* interval, we can determine the starting position of each occurrence in *G* by directly looking up *SA*. Loading the entire *SA* into the host memory would require $|G|\lceil \log_2 |G| \rceil$ bits. For the human genome, the memory size is about 12 GB. Although this memory requirement can be met in high-end workstations, it is still a challenge for most commonly available computers. Fortunately, we can reconstruct the entire *SA* from parts of it. Ferragina and Manzini [11] have shown that an unknown value $SA[i]$, can be re-established from a known $SA[j]$ using Equation (2).

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

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

$$\beta(i) = C(B[i]) + Occ(B[i], i) \tag{3}$$

Based on Equations (2) and (3), we construct a reduced suffix array (*RSA*) by simply storing $SA[i]$ whose index i is a multiple of p (default=32), reducing the total memory size of *SA* to $|G|\lceil \log_2 |G| \rceil / p$ bits. Users can trade-off the lookup time and memory space by selecting different p. A smaller p means larger memory and (nearly always) faster lookups. For a suffix array index i that is not a multiple of p, we repeat t iterations using Equation (3) until j is a multiple of p, where $SA[j]$ is equal to $RSA[j/p]$, and then calculate $SA[i]$ as $SA[j]+t$ following Equation (2).

If the starting positions are calculated on the CPU following Equation (2), it does not only require loading *RSAs* into the memory, but also needs to load the corresponding *bwt* or *rbwt*. This results in more memory overhead in the host. In addition, more compute overhead will also be required to calculate j and t values in Equation (2), thus increasing the execution time. Fortunately, we find that the calculation of j and t is independent of *RSAs* and can be directly computed using the corresponding *bwt* (or *rbwt*) for each occurrence. Hence, after gaining the *SA* interval, we calculate the j and t values on GPUs for all indices in the *SA* interval, and then output them instead of the final starting positions. In this way, on the host we only require loading the *RSAs* into the memory, and can get the starting position of each occurrence very quickly by only two operations: one table lookup and one addition.

D. Exact-match and Inexact-match Search

For exact-match search, it can be easily done following the calculations in Equation (1). As for inexact-match search, it can be transformed into exact-match search by introducing substitutions (mismatches), insertions and deletions in the seed and the reference genome. Since we do not allow insertions and deletions, the inexact-match search can be transformed to the exact-match search of all possible sequences that have a Hamming distance of not more than the allowable number of mismatches. Because the exact-match search is straightforward, two separate CUDA kernels are employed to implement the two searches. In addition to the forward strand of *S*, we also consider its reverse complement. For simplicity and clarity, the following discussions only refer to the search from the forward strand, if not specified.

The inexact-match search is equivalent to the traversal of a complete 4-ary tree, where each possible sequence corresponds to a path from the root (the root node is Ø, meaning an empty string) to a leaf and each node along the path corresponds to a base in the sequence that has the same position. Hence, we can find all inexact matches of *S* by traversing the tree using either depth-first search (DFS) or breadth-first search (BFS). In this paper, we use the DFS traversal and implement it using a stack data structure. Since *S* has a maximal allowable number of mismatches, we must confine the number of mismatches not to exceed the limit when searching along a path from top to down. If the number of mismatches, in the sub-path down to the current node, exceeds the limit, we will stop traversing the sub-trees of the current node. In addition, to further reduce the search space, we estimate the lower bound of the number of mismatches that are required to transform the substring, represented by the sub-path down from the current node to the leaf, to have exact matches to the genome. The approach proposed in BWA is used, which estimates the lower bound of the number of mismatches for a string by counting the number of all constituent non-overlapping substrings that do not have exact matches to the genome. We define a vector $DIFFS(\bullet)$ for *S* to store the estimated values, where $DIFFS(i)$ represents the minimal number of mismatches that are

required in the suffix of S starting at position i+1, where $0 \le i < |S|-1$.

To generate fixed-length seeds, we need to identify exact or inexact matches in the reference genome for k-mers from the input short reads. Since we only measure the runtime performance of fixed-length seed generation on the GPU, without loss of generality, we pre-generate k-mers from short reads and then take these k-mers as input for simplicity. For both exact-match and inexact-match searches, one thread is assigned to search a k-mer as well as its reverse complement. Since we usually have a large number of k-mers, it is infeasible to load all k-mers into the device memory on the GPU. In this case, we organize all k-mers into batches and employ multiple passes to complete the whole search. In this way, in each pass, we only need to load a batch of k-mers into the device memory, thus significantly alleviating the device memory pressure. The k-mer batch is stored in texture memory bound to linear memory at start-up time, and then is loaded into shared memory when performing the search since we see slight performance improvement after using shared memory. For a k-mer, it is possible to find a lot of matches in G, making it difficult for us to store and to output all matches because after loading the BWT data, we usually have a little device memory available. To solve this problem, we simply discard all matches of a k-mer if the number of its matches exceeds a specified threshold (128 by default). During the search, we will stop the traversal if a k-mer has already found more than the threshold matches. This approach works but has a risk of discarding some significant matches. Figure 4 shows the pseudocode of the CUDA kernel for the inexact-match search from the forward strand of S.

```
#bwt: the forward BWT of genome G; rbwt: the reverse BWT of G; S: a forward-strand k-mer.

1. estimate the lower bound of the number of mismatches for each suffix of S
   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. 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;
      check the number of mismatches in the full length;
      if exceeding the limit 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
            if the total number of k-mer matches exceeds the threshold then
               return;
            else
               store the hit;
            fi
         else
            push the next base in S into the stack as well as other information;
         fi
      fi
   done
```

Figure 4. Pseudocode of the CUDA kernel for inexact-match search from the forward strand of a k-mer S

IV. Performance Evaluation

A. Experimental Design

We have measured the performance of our GPU-based seed generator by comparing the execution times of the CPU and GPU versions that execute the same core code, where the kernel code for a single CUDA thread is ported onto the CPU with no need of algorithmic changes. 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. A Fermi-based Tesla C2050 GPU is used for the evaluation of the GPU version. This 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). The CPU version has been parallelized using the OpenMP programming model and is compiled with the GNU GCC (version 4.1.2) tool chain. The GPU version is compiled with the CUDA toolkit release 4.0. Both of the two versions are optimized at compile time using the compiling option "-O3".

We have pre-generated five k-mer datasets from short reads simulated from the human genome, named as S11, S15, S20, S24 and S30 respectively. Each dataset consists of one million k-mers and all k-mers in a specific dataset have the same lengths. The k-mer length of a dataset can be inferred from its name, i.e. S11 means a k-mer length of 11 and S15 means a k-mer length of 15 and so on. All the following tests do not allow mismatches in the seeds for the S11 and S15 datasets of small k-mer lengths, allow one mismatch for the S20 and S24 datasets of medium k-mer lengths, and allow two mismatches for the S30 dataset of large k-mer lengths. As mentioned above, two separate CUDA kernels are used for the exact-match and inexact-match search respectively. For all tests, we set a thread block to have 192 threads for the exact-match search and 64 threads for the inexact-match search.

B. Results and Discussion

Firstly, we have evaluated the total runtime (measured in wall time) performance to complete the whole seed generation, where the GPU version calculates the location information (see Equation (2)) of all matches on the GPU. Table 1 gives the runtimes (in seconds) of the CPU and GPU versions as well as the speedups of the GPU version over the CPU version. The experimental results in the table indicate that the GPU version on a single GPU yields significant speedups over the CPU version on either a single CPU core or four CPU cores. Specifically, the GPU version achieves an average speedup of 9.5, with a highest of 13.5, over the CPU version on a single CPU core, and an average speedup of 2.9, with a highest of 3.8, over the CPU version on four CPU cores. Furthermore, the speedups generally increase with the k-mer length increasing, even though a lowered speedup is encountered when using the S24 dataset.

Secondly, we have evaluated the runtime performance for only searching the k-mer matches using the BWT and FM-index, without considering locating occurrences using the suffix array. Table 2 shows the runtimes (in seconds) of the two versions as well as the speedups of the GPU version

over the CPU version. Similar to the results of the whole seed generation (see Table 1), the speedups generally increase with the *k*-mer length increasing except for a lowered speedup when using the S24 dataset. The GPU version achieves a highest speedup of 13.6 over the CPU version on a single CPU core, and a highest speedup of 3.9 over the CPU version on four CPU cores. Readers might be surprised at the speedups for the S11 and S15 datasets, where the GPU version is slower than the CPU version. This can be explained by the short computational time on the GPU, due to the small *k*-mer lengths, and the relatively larger extra overhead, introduced by data preparation and transfer for the CUDA kernel executions. However, after performing the exact-match search using the other three datasets of larger *k*-mer lengths, we found that the GPU version is faster than the CPU version on a single CPU core, with speedups increasing as the *k*-mer lengths increase, but not very much (data not reported).

TABLE I. PERFORMANCE COMPARISON FOR THE WHOLE SEED GENERATION EXECUTION

Dataset	#Mismatch	Time (in seconds)			Speedup	
		1 core	*4 cores*	*1 GPU*	*1 core*	*4 cores*
S11	0	60.7	17.2	10.6	5.7	1.6
S15	0	122.4	36.1	12.4	9.9	2.9
S20	1	179.8	59.1	17.6	10.2	3.4
S24	1	129.1	44.3	15.8	8.2	2.8
S30	2	1094.0	311.2	80.9	13.5	3.8

TABLE II. PERFORMANCE COMPARISON FOR THE *K*-MER MATCHES SEARCH

Dataset	#Mismatch	Time (in seconds)			Speedup	
		1 core	*4 cores*	*1 GPU*	*1 core*	*4 cores*
S11	0	5.8	3.4	6.8	0.9	0.5
S15	0	7.3	4.1	7.2	1.0	0.6
S20	1	139.8	47.6	14.9	9.4	3.2
S24	1	107.6	37.9	13.4	8.0	2.8
S30	2	1067.6	309.5	78.6	13.6	3.9

Finally, we have evaluated the runtime performance of the GPU version by calculating the location information sequentially on the CPU, instead of the GPU. Figure 5 shows the speedups over the CPU version on a single CPU core and four CPU cores. For the S11 and S15 datasets of small *k*-mer lengths, the runtimes of the CPU and GPU versions are almost the same. Referring to the runtimes shown in Tables 1 and 2, we found that for datasets of small *k*-mer lengths, the calculation of location information dominates the whole execution. This tells us that we should select as small *p* for *RSA* as possible, according to the available memory capacity in the host, to speed up the locating of positions. For the other three datasets, the GPU version achieves considerable speedups over the CPU version with increasing speedups towards larger *k*-mer lengths. For the S30 dataset, the GPU version yields a highest speedup of 11.0 (3.1) over the CPU version on a single CPU core (four CPU cores).

V. CONCLUSION

In this paper, we have investigated and evaluated the performance of fixed-length seed generation on GPUs based on the BWT and FM-index. In addition to exact matches, we allow mismatches to occur at any position within a seed. For the exact-match search, we simply follow the backward search procedure using the FM-index and for the inexact-match search, we employ a stack data structure to implement the DFS traversal in order to find all possible matches that have a Hamming distance of not more than the allowable number of mismatches. The major challenges for GPU-based seed generation using the BWT are the frequent accesses to global memory with poor data locality and the divergence of search paths for different *k*-mers. The poor data locality will lead to more misses in the L1/L2 caches for global memory accesses and the divergence of search paths cause the execution paths of the threads in a warp to diverge frequently. Overall, we have achieved some encouraging experimental results through our evaluations, where the GPU version achieves significant speedups over the CPU version. Specifically, the GPU version achieves an average speedup of 9.5 (2.9), with a highest of 13.5 (3.8), over the CPU version on a single CPU core (four CPU cores). Furthermore, the speedups generally increase with the increase of *k*-mer lengths. We hope these results can provide some useful guidance to the development of GPU-oriented seed generators for genomics.

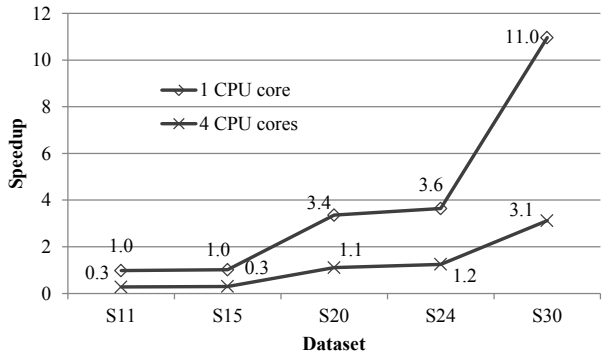

Figure 5. Speedups of the GPU version with occurrence locating sequentially on the CPU over the CPU version on a single CPU core and four CPU cores

ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful comments that helped to improve the manuscript, and we acknowledge the fund "Schwerpunkt für Rechnergestützte Forschungsmethoden in den Naturwissenschaften".

REFERENCES

[1] T.A. Down, V.K. Rakyan, D.J. Turner, P. Flicek, H. Li, E. Kulesha, S. Gräf, N. Johnson, J. Herrero, E.M. Tomazou, N.P. Thorne, L. Bäckdahl, M. Herberth, K.L. Howe, D.K. Jackson, M.M. Miretti, J.C. Marioni, E. Birney, T.J. Hubbard, R. Durbin, S. Tavaré, and S. Beck, A Bayesian deconvolution strategy for immunoprecipitation-based

DNA methylome analysis," Nat Biotechnol, vol. 26, no. 7, 2008, pp. 779-785

[2] D.S. Johnson, A. Mortazavi, R.M. Myers, and B. Wold, "Genome-wide mapping of in vivo protein-DNA interactions," Science, vol. 316, no. 5830, 2007, pp. 1497-1502

[3] J.C. Marioni, C.E. Mason, S.M. Mane, M. Stephens, and Y. Gilad, "RNA-seq: an assessment of technical reproducibility and comparison with gene expression arrays," Genome Res., vol. 18, no. 9, 2008, pp. 1509-1517

[4] H. Li, J. Ruan, and R. Durbin, "Mapping short DNA sequencing reads and calling variants using mapping quality scores,", Genome Res., vol. 18, no. 11, 2008, pp. 1851-1858

[5] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman, "Basic local alignment search tool," J Mol Biol., vol. 215, no. 3, 1990, pp. 403-410

[6] W.J. Kent, "BLAT--the BLAST-like alignment tool," Genome Res. vol. 12, no. 4, 2002, pp. 656-664

[7] P. Flicek, and E. Birney, "Sense from sequence reads: methods for alignment and assembly," Nat Methods., vol. 11, no. Suppl., 2009, pp. S6-S12

[8] N. Homer, B. Merriman, and S.F. Nelson, "BFAST: an alignment tool for large scale genome resequencing," PLoS One, vol. 4, no. 11, 2009, pp. e7767

[9] S.M. Rumble, P. Lacroute, A.V. Dalca, M. Fiume, A. Sidow, and M. Brudno, "SHRiMP: accurate mapping of short color-space reads," PLoS Comput Biol., vol. 5, no. 5, 2009, pp. e1000386

[10] M. Burrows, and D.J. Wheeler, "A block sorting lossless data compression algorithm," Technical Report 124 Palo Alto, CA, Digital Equipment Corporation, 1994

[11] P. Ferragina, and G. Manzini, "Indexing compressed text," Journal of the ACM, vol. 52, no. 4, 2005

[12] H. Li, and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," Brief Bioinform., vol 11, no. 5, 2010, pp. 473-483

[13] S.B. Needleman, and C.D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," J Mol Biol., vol. 48, no. 3, 1970, pp. 443-453

[14] T.F. Smith, and M.S. Waterman, "Identification of common molecular subsequences," J. Mol.Biol., vol. 147, no. 1, 1991, pp. 195-197

[15] M. Csurös, "Performing local similarity searches with variable length seeds," Lect Notes Comput Sci, vol. 3109, 2004, pp. 373–387

[16] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg, "Versatile and open software for comparing large genomes," Genome Biol., vol. 5, no. R12, 2004

[17] E. Ohlebusch, and S. Kurtz, "Space efficient computation of rare maximal exact matches between multiple sequences," J Comput Biol, vol. 15, no. 4, 2008, pp. 357-377

[18] S.M, Kiełbasa, R. Wan, K. Sato, P. Horton, and M.C. Frith, "Adaptive seeds tame genomic sequence comparison," Genome Res., vol. 21, no. 3, 2011, pp. 487-493

[19] B. Ma, J. Tromp, and M. Li, "PatternHunter: faster and more sensitive homology search," Bioinformatics, vol. 18, no. 3, 2002, pp. 440-445

[20] Y. Liu, D.L. Maskell, and B. Schmidt, "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units," BMC Research Notes, vol. 2, no. 73, 2009

[21] W. Liu, B. Schmidt, and W. Müller-Wittig, "CUDA-BLASTP: accelerating BLASTP on CUDA-enabled graphics hardware," IEEE/ACM Trans Comput Biol Bioinform., vol. 8, no. 6, 2011, pp. 1678-1684

[22] P.D. Vouzis, and N.V. Sahinidis, "GPU-BLAST: using graphics processors to accelerate protein sequence alignment," Bioinformatics vol. 27, no. 2, 2010, pp. 182-188

[23] Y. Liu, B. Schmidt, and D.L. Maskell, "MSA-CUDA: multiple sequence alignment on graphics pro-cessing units with CUDA," 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, 2009, pp. 121-128

[24] Y. Liu, B. Schmidt, W. Liu, and D.L. Maskell, "CUDA-MEME: accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units," Pattern Recognition Letters, vol. 31, no. 14, 2009, pp. 2170-2177

[25] Y. Liu, B. Schmidt, and D.L. Maskell, "DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI," BMC Bioinformatics, vol. 12, no. 95, 2011

[26] J. Blom, T. Jakobi, D. Doppmeier, S. Jaenicke, J. Kalinowski, J. Stoye, and A. Goesmann, "Exact and complete short read alignment to microbial genomes using GPU programming," Bioinformatics, vol. 27, no. 10, 2011, pp. 1351-1358

[27] C.M. Liu, T.W. Lam, T. Wong, E. Wu, S.M. Yiu, Z. Li, R. Luo, B. Wang, C. Yu, X. Chu, K. Zhao, and R. Li, "SOAP3: GPU-based compressed indexing and ultra-fast parallel alignment of short reads," 3th Workshop on Massive Data Algorithms, 2011

[28] Y. Liu, B. Schmidt, and D.L. Maskell, "CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform," http://cushaw.sourceforge.net, 2011

[29] E, Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: a unified graphics and computing architecture," IEEE Micro, vol. 28, no. 2, 2008, pp. 39-55

[30] NVIDIA, "NVIDIA's next generation CUDA compute architecture: Fermi," NVIDIA Corporation Whitepaper, 2009

[31] W.K. Hon, T.W. Lam, K. Sadakane, W.K. Sung, and S.M. Yiu, "A space and time efficient algorithm for constructing compressed suffix arrays,", Algorithmica, vol. 48, no. 1, 2007

[32] H. Li, and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform,", Bioinformatics, vol. 25, no. 14, 2009, pp. 1754-1760

[33] T.W. Lam, W.K. Sung, S.L. Tam, C.K. Wong, and S.M. Yiu., "Compressed Indexing and Local Alignment of DNA,", Bioinformatics, vol. 24, no. 6, 2008, pp. 791-797