Masher: Mapping Long(er) Reads with Hash-based Genome Indexing on GPUs

Anas Abu-Doleh[†‡] Erik Saule[†] Kamer Kaya[†] Ümit V. Çatalyürek[†‡]
[†]Dept. of Biomedical Informatics
[‡]Dept. of Electrical and Computer Engineering
The Ohio State University
abudoleh.1@osu.edu, {esaule,kamer,umit}@bmi.osu.edu

ABSTRACT

Fast and robust algorithms and aligners have been developed to help the researchers in the analysis of genomic data whose size has been dramatically increased in the last decade due to the technological advancements in DNA sequencing. It was not only the size, but the characteristics of the data have been changed. One of the current concern is that the length of the reads is increasing. Although existing algorithms can still be used to process this fresh data, considering its size and changing structure, new and more efficient approaches are required. In this work, we address the problem of accurate sequence alignment on GPUs and propose a new tool, Masher, which processes long (and short) reads efficiently and accurately. The algorithm employs a novel indexing technique that produces an index for the 3,137Mbp `hg19` with a memory footprint small enough to be stored in a restricted-memory device such as a GPU. The results show that Masher is faster than state-of-the-art tools and obtains a good accuracy/sensitivity on sequencing data with various characteristics.

Categories and Subject Descriptors

J.3 [**Life and Medical Sciences**]: Biology and genetics; D.1.3 [**Software**]: Programming Techniques—*Parallel Programming*

General Terms

Algorithms, Experimentation, Performance

Keywords

Sequence alignment, parallel programming, GPUs, indexing

BCB'13, September 22 - 25 2013, Washington, DC, USA

ACM 978-1-4503-2434-2/13/09...$15.00.
http://dx.doi.org/10.1145/2506583.2506641

1. INTRODUCTION

There is a wide interest in efficient and accurate mapping of the genomic sequences which are generated by next generation sequencing (NGS) devices. Due to the improvements in the NGS technology, the length of the reads obtained from these machines is continuously increasing: for example, Roche 454 machines produce 400-500bp reads today. As a result, the focus moved from mapping reads with less than 100 bases to mapping longer reads [15].

Alignment is a computationally expensive process and various tools and techniques have been developed to find high quality alignments while decreasing the alignment time: For example, Bowtie2 uses a Burrows-Wheeler Transform (BWT) index [3] and dynamic programming [10, 11]. GASSST uses filtering in order to eliminate some candidate alignments [25]. SeqAlto employs a compact hash indexing technique and the Needleman-Wunsch algorithm [24]. SHRiMP2 indexes the genome with spaced seeds and applies Smith-Waterman (SW) [29] for the candidate locations [6, 26]. CUSHAW2 uses the BWT to reduce the search space and achieve high alignment quality [18]. BFAST uses multiple independent indexes and SW with gaps to support small indel detection [8]. These tools were designed to map short reads and they perform that task adequately. However, their performance degrades when the reads get longer.

Manycore architectures such as Graphical Processing Units (GPU) can handle a herculean task in parallel and in a short time which make them desirable for sequence mapping. In theory, since there exist a huge amount of reads, the task of mapping is *pleasingly parallel*. That is, one can process a single read by a single GPU thread (or a group of threads) and easily obtain a good speedup on the alignment time since a GPU can execute thousands of threads concurrently. However, in practice, there are two main limitations: first, although GPUs (and similar accelerator architectures such as Intel Xeon Phi) provide a significant computational power, they have less memory than a CPU-based architecture. And second, to obtain the best performance, the computations within a *warp* (a set of threads being executed concurrently by the same processor) must be as homogeneous as possible because, the threads in the same warp are serialized if they are executing different instructions.

Recently, many of the popular alignment tools such as Bowtie2 and CUSHAW2 employ BWT to generate an index and align the reads. A BWT-based index such as the *FM-index* [7] is proved to be very efficient in practice. In fact, its memory footprint can be 10–20x smaller than other popular alternatives such as the suffix array. Its small size

allows to fit it in memory restricted architectures such as GPUs. However, its small size does not make it a *one size fits fit all* solution for all the architectures. When the reads do not exactly match to the reference, BWT-based solutions use techniques like *backtracking* and *branching* to handle inexact matchings, which cause serialized execution in GPUs. Hence, especially with the increasing read length, the use of BWT-based solutions for GPUs arguably causes major performance issues.

In this work, we introduce and experimentally evaluate a GPU-based mapping tool Masher which uses a hash-based scheme to align long (as well as short) reads to a reference genome. The tool uses a novel lossy indexing technique which is memory efficient and allow interesting time-memory tradeoff to map the reads to the reference genome accurately and efficiently.

We evaluate Masher with the 3,137Mb human genome `hg19` whose index fits to GPU. The results show that the proposed algorithm is an order of magnitude faster than some of the state of the art tools on read whose length is greater than 500 (and similar or better performance on smaller reads). Despite using a lossy index, it obtains a good accuracy and sensitivity.

The rest of the paper is organized as follows: Section 2 summarizes some relevant mapping literature. The details of the proposed tool Masher are given in Section 3. Section 4 presents the experimental results, and Section 5 concludes the paper.

2. BACKGROUND

Although many short-read alignment tools have been developed in the past, there are only three main approaches they use: The first is based on hashing, the second is based on the suffix trees or the BWT-based FM-index, and the last approach is based on merge sorting which is very rare [20, 21]. For a survey and the details of these approaches, see [15].

All suffix-tree- and BWT-based approaches use the reference genome to build the index. On the other hand, the hash-based tools either use the reference or reads to build the hash index. This choice depends on the size and structure of the input and a design rationale. The GPU-based tool Masher proposed in this work uses the reference for index construction. Here, we first start with a short description of the GPU architecture and then, we briefly discuss the existing alignment tools which support a form of parallelism.

2.1 GPU architecture

A graphical processing unit (GPU) is a highly parallel device built to speed up the execution of the massively computational applications. The single instruction-multiple threads (SIMT) architecture of the GPU is based on that all threads in the same core execute the same instruction in parallel. In a GPU, a *thread* defines the finest computational granularity throughout the execution. Each thread has a relatively large number of registers and some thread local (off-chip) memory in case registers are not enough. The threads are grouped within *blocks*, and all the threads in a single block can access the same shared memory. The maximum number of threads a block can contain is limited. A *grid* of blocks is responsible form whole kernel execution. Thus, during the computation, each thread has a unique ID in a block, and each block has a unique ID within a grid. The structure is given in Figure 1.

A GPU is composed of streaming multiprocessors (SM in NVIDIA Fermi or SMX in NVIDIA Kepler) and each block is assigned to an SM within the SM's execution capacity. Each SM and SMX has 32 and 192 CUDA cores, respectively. A group of threads that physically run in parallel is called a warp. The number of such threads, i.e., the warp size, may change in the future, but currently it is 32 for cutting-edge GPU architectures. Each thread in a warp usually operates on a different data but they execute the same instruction in parallel. Hence, to obtain the best performance, the computations within a warp must be as homogeneous as possible because, the threads in the same warp are serialized if they are executing different instructions. When the kernel has '**if**'s and '**else**'s, i.e., when the computation is branched (also called *divergent* in GPU computing), the threads are serialized, concurrency decreases and performance degrades.

As shown in Figure 1, in a GPU there are different memory types with different access scope and speed. Hence, taking the hierarchical memory structure into consideration can be very important to improve efficiency. These memory types are grouped into on-chip and off-chip: Registers, shared memory, and cached constant memory are on-chip memory where the access time is very fast. On other hand, global memory, local memory, uncached constant memory, and texture memory are off-chip memory where the access time is slower by 100x times than on-chip memory.

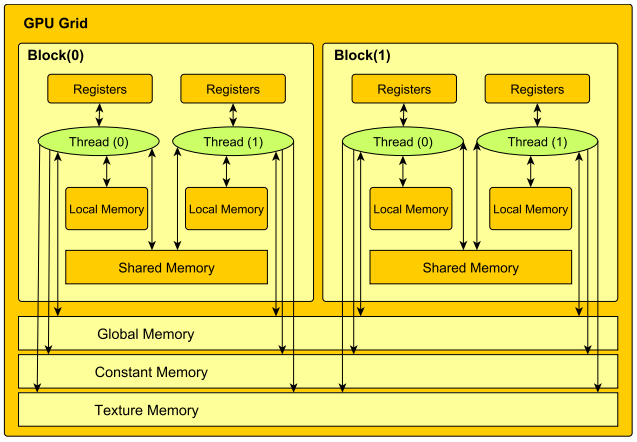


Figure 1: GPU architecture and CUDA memory model.

2.2 Hash-based methods

To cope with the massive sequencing data, several hash-based tools were enhanced and parallelized, and even redesigned. For example, SHRiMP2 [6], whose predecessor SHRiMP [26] was using the reads to build hash-based index, employs the reference for index construction to achieve a better parallelism. This redesigning made SHRiMP2 faster than BFAST [8], which also uses the genome to build the hash index [6].

GNUMAP is a popular tool which uses a probabilistic approach to align short reads. Although, the initial version does not have a concurrency support [5], it has been extended to employ thread-level parallelism [4]. GNUMAP uses the genome to build the index and the threads are assigned to different reads. The major drawback of this

approach is the large memory footprint when the genome is large which is the case for human genomes. Another hash-based alignment tool RMAP indexes the short reads instead of the genome [28]. Aji et al. proposed the tool GPU-RMAP which executes RMAP's algorithm on a GPU using CUDA [1].

In addition to shared memory parallelism, to cope with the memory barrier and to obtain a faster tool, many researchers used distributed memory architectures. For example, GNUMAP has the option to distribute the reference genome among the nodes in a network. Since each node indexes a part of the genome, the memory requirement per node is reduced significantly. Other tools, such as Novoalign and rNA also have versions that support both distributed and shared memory parallelism, namely, NovoalignMPI and mrNA. A similar sequential tool FANGS [23] evolved to pFANGS [22] where a pure distributed-memory approach was found to be more scalable. There exist similar hash-based tools which rely on genome partitioning over a distributed memory to deal with the size of the genome. Various indexing and data distribution strategies for MPI based parallelization of short-read alignment algorithms were investigated by [2].

GPUs have much less memory than a CPU-based system or a distributed memory computer. Hence, they may not be able to store a conventional hash-based genome index. The techniques we propose in this work uses a lossy hash-based genome index. It is engineered in a way that significantly reduces its size and makes it small enough to fit into memory restricted architectures such as GPUs. Still, the alignments produced retain a high quality.

2.3 Suffix-tree and BWT-based methods

Instead of a hash-based index, a suffix tree [30] can be constructed from the reference genome. MUMmer, which is designed for the exact alignment problem, is the first tool that uses a suffix tree for sequence alignment [9]. Schatz et al. parallelized MUMmer by using CUDA and developed MUMmerGPU for the exact alignment on the GPUs [27]. To cope with the memory challenge, MUMmerGPU uses several smaller but overlapping suffix trees instead of one big tree. Although the proposed tree layout is desirable for short queries, when the queries get longer, there is a dramatic reduction on the performance due to more cache access time and the divergence of thread loads.

Another index structure closely related to the suffix tree is the FM-index [7], which is constructed with the Burrows-Wheeler transform [3]. There exist several popular BWT-based alignment tools for short and long reads such as Bowtie and Bowtie2 [10, 11], BWA [13, 14], and SOAP3 [16]. A BWT index is small: 3GB memory is sufficient to store the index created from a human genome, whereas even a suffix array for the same genome consumes more than 12GB memory. Since all the BWT-based algorithms create the index from the genome, while aligning the reads, a massive parallelism is possible. In fact, when Bowtie was first proposed, it was already more than 30 times faster than most of the non-BWT tools [11].

The reduction on the index size induced by the BWT-based algorithms enables using GPUs for alignment. Furthermore, the speedups obtained by the GPU-based parallelization can be spectacular especially when the reads (almost) exactly match to the reference genome. In this case, the instruction patterns of two different threads processing two different reads are similar throughout the alignment. That is even the data are different, the computation patterns resemble each other. However, when the number of mismatches increases, the computation branches, i.e., deviates from the generic scheme, and a significant amount of backtracking [11] is needed to traverse the search space for accuracy. On a GPU, this branching can easy result in the threads of the same warp being sequentialized and wait for each other in different branches. Hence, considering the variety and different characteristics of the HPC architectures available to the researchers, a BWT-based index is not a one size fit all solution for efficiency. Yet, it is still a very powerful and useful technique for sequence mapping.

In our experiments, we used three popular and fast aligners to evaluate and compare Masher's performance: a CPU-based tool, Bowtie2, and two GPU-based tools, SOAP3-dp [19] and CUSHAW2-GPU. We experimented with Bowtie2 both in sensitive and fast mode. We chose to use SOAP3-dp instead of SOAP3 since it is reported to improve SOAP3 in terms of both speed and sensitivity by using dynamic programming on a GPU, a technique we also employ in Masher.

3. MASHER

The workflow of Masher is shown in Figure 2. After obtaining the genome, Masher first constructs the index to be used for the alignment. Then it seeds each read (and also its reversed and complemented form) and use the index to locate the seeds in the genome. The seed locations are then converted to base locations for the corresponding read. By using a merge operation, the base locations which are close to each other are merged, and the candidate locations are sorted w.r.t. their quality, i.e., the number of base locations they correspond. Starting with a batch which contains the promising locations, a local alignment is performed to score each candidate location until one with a score above the minimum desired score is found. Masher processes multiple reads on GPU in batches and in parallel. Here, we describe each step in detail.

3.1 Index construction

Masher uses a novel indexing scheme. Let L and ℓ be the read and seed lengths, respectively. In the current version of Masher, ℓ is set to 15bp. Each base is represented by 2 bits (i.e., A = 00, C = 01, G = 10, and T = 11)[1], so each seed can be represented by a *value* encoded as a 30-bit integer (in memory, a 32 bit integer is used). Let $|x| = \lceil \log_2 x \rceil$ be the number of bits required to represent x and let N be the length of the reference genome.

A regular index uses two arrays: an array locs which stores the locations of each seed in sorted order and requires $N|N|$ bits of memory. And another array ptrs which stores the index of the first location of each seed in locs and requires $4^\ell|N|$ bits of memory. For `hg19`, the genome we used in the experiments has $N = 3,137 \times 10^9$ nucleotides and we need 11.7GB and 4GB of memory to store locs and ptrs, respectively. This typically does not fit in the memory of a GPU.

In order to minimize the size of the index, as Figure 3 shows, Masher leaves a space, the *indexing step*, between

[1]We replaced each N in the genome with A.

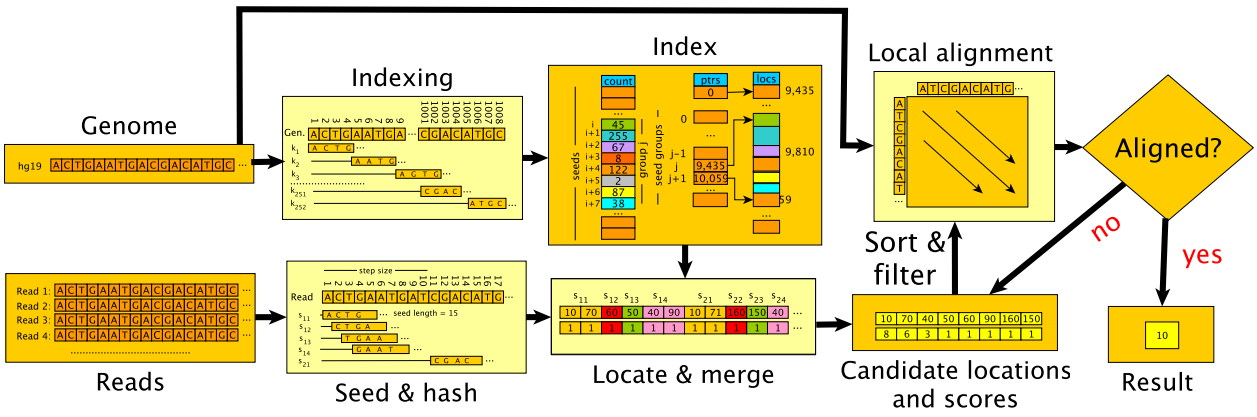


Figure 2: The workflow of Masher.

the seeds whose size is denoted by Δ_G. That is assuming the first location is 0, only the seeds that start from the locations divisible by Δ_G are indexed. Hence for a genome of size Nbp, only $\lceil N/\Delta_G \rceil$ locations are taken in the account. locs now requires $\lceil N/\Delta_G \rceil |N|$ bits of memory and ptrs takes $4^\ell |\lceil N/\Delta_G \rceil|$ bits. Masher uses $\Delta_G = 4$, and the total memory requirement is around 2.9GB for locs and 4GB for ptrs which is still larger than the available memory in many cutting edge GPUs. A similar technique has been used to generate a sparse k-mer graph for de novo genome assembly [31].

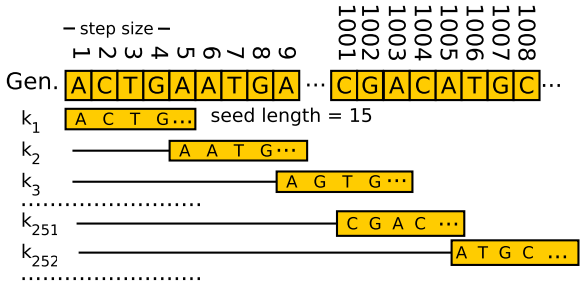


Figure 3: Seeding for index construction in Masher: the seed length ℓ and the indexing step size Δ_G are set to 15 and 4, respectively.

To further reduce the memory footprint, Masher also employs a time-memory tradeoff and uses a two-level access structure as shown in Figure 4: the seed values are grouped by δ in their natural (lexicographic) order, and the ptrs array is constructed not for the individual seeds but only for the seed groups. That is, given a seed s, one can compute its group id by $\lfloor s/\delta \rfloor$ and access the group's first entry in locs whose location is stored in ptrs. This makes ptrs' size $\lceil 4^\ell/\delta \rceil |\lceil N/\Delta_G \rceil|$. Masher uses $\delta = 8$ and ptrs now takes 0.5GB in memory for hg19.

To access seed s's actual first entry in locs, Masher uses another array counts of size 4^ℓ, which stores the number of entries in locs for each possible seed. Hence, with $\mathcal{O}(\delta)$ addition, the position of the first location where the value of s appears is reached. With this scheme, the memory footprint of counts is $4^\ell |\lceil N/\Delta_G \rceil|$ since there are $\lceil N/\Delta_G \rceil$ possible locations. However, we observed that for many seeds, the number of locations a seed appears is much smaller. To keep the index in GPU memory, we store at most 255 locations. Hence, each entry in counts can be stored by using one byte of memory, and the whole array can be stored with 4^ℓ bytes. When a seed appears in more than 255 locations, Masher randomly (uniformly) chooses 255 of them. counts take 1GB of memory. With these parameters and techniques, the genome index can be stored in modern GPUs like C2070 or K20/K20x by using only 4.4GB. For GPUs with smaller memory, one can choose larger values of Δ_G and δ at the expense of increased execution time. Masher currently uses a sequential implementation of the index construction but it can be easily parallelized.

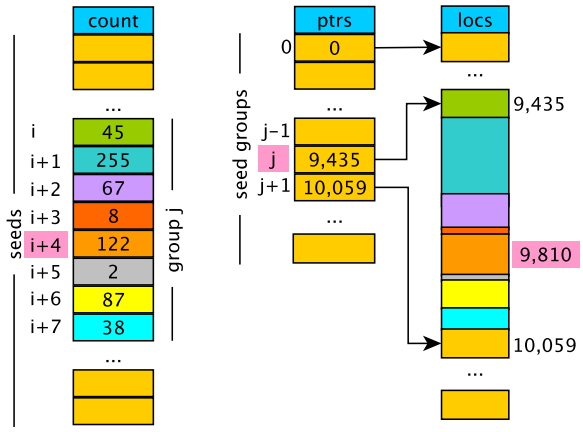


Figure 4: The index structure used in Masher containing three arrays locs, ptrs, and counts. Let j be an integer and $i = \delta \times j$. If seed $i+4$ is accessed, Masher first computes the group id by $j \leftarrow \lfloor (i+4)/\delta \rfloor$. Group j's first entry in locs (actual location entry for seed i) is found via ptrs. By using the counts array, Masher jumps within the group entries and finds the first location entry for seed $i+4$.

3.2 Finding candidate locations

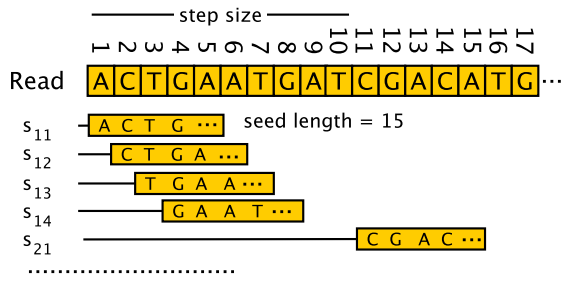


Figure 5: Generation of seeds from a read: in the example, the seed length ℓ and the read step size Δ_R are set to 15 and 10, respectively.

Given an Lbp read, Masher generates multiple seeds of length ℓ. The seed generation process is described in Figure 5. Assuming the first location is 0, the aligner uses only the seeds starting from the read locations

$$\{i : i \bmod \Delta_R < \Delta_G, i < L - \ell\}$$

where Δ_R is the *read step size*. Starting from location 0, Δ_G consecutive seeds form a group, and the distance between the first seed of a group and that of next one is Δ_R. And the second condition $i < L - \ell$ is required to avoid seeds shorter than ℓbp. Hence, for each read, Masher uses approximately $s = \Delta_G\lceil(L-\ell)/\Delta_R\rceil$ seeds.

Consider a read which exactly matches with a region in the reference. Since the seeds in a group are generated from consecutive locations and there are Δ_G of them, at least one seed in each group must have been used during the index construction phase. Note that the index is generated from the seeds starting from every other Δ_Gth reference location. For each location of a seed in the locs array, a *base location*, the one for the first nucleotide of the read in case of a match, is obtained by using the position of the seed in the read. Hence, when there is an exact match between a read and the reference, the base location corresponds to at least $\lfloor(L-\ell)/\Delta_R\rfloor$ seed locations in the locs array. And when Δ_R is small, this number increases.

After the seeds are generated, Masher finds the locations where they appear in the genome as previously described in Figure 4. This is performed in two steps: first, the counts array is visited for each seed, the total number of seed locations is obtained, and the memory which will be used to store these locations is allocated in the GPU. Second, the ptrs and counts arrays are used to access the first entry in locs of each seed. By using the positions of the seeds on the read, the values in the index are converted to base locations and brought to the memory. For efficiency purposes, the seeds which appear in more than ω reference locations are ignored in both steps where ω is an integer parameter smaller than or equal to 255. In the current implementation of Masher, we use $\omega = 60$. That is the seeds which have more than 60 locations are ignored. And for the remaining reads, which are not aligned at the end of local alignment, the whole alignment process is repeated with $\omega = 255$. An example snapshot of the memory is given in Figure 6(a). In the example, the base locations for two seed groups are given where $\Delta_G = 4$.

Finding and storing the candidate base locations is executed in parallel on the GPU with two kernels, each responsible from one of the steps described above. Masher processes the reads in batches where the batches contain 16,384 reads. This number is chosen by considering the memory Masher requires with $L = 1000$. For each kernel, a read is processed by a single GPU block where the block size, i.e., the number of threads in a block is set to s. Thus, a seed is processed by a single thread.

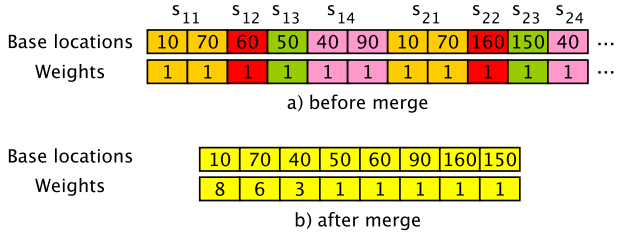


Figure 6: The memory snapshot of the candidate locations for a read. (a) The locations are grouped with respect to their seeds and each group is already sorted (as they are stored in the index). Initially, the weights are set to 1. (b) The merged location array is constructed and the weights are combined. The arrays are sorted w.r.t. the weights. Only a prefix of the location array in (a) is given.

As Figure 6 shows, each candidate location entry is initially weighted with 1. A location can appear in the memory more than once. In fact, if the read is matched with the reference genome the corresponding base location appears many times. To distinguish the good and bad candidate base locations, the locations are merged and their weights are combined by an algorithm similar to *mergesort*. For each seed, the base locations are already sorted (since the exact seed locations are stored in-order in the index). Starting from these sorted location sets, Masher performs merge operations on pairs of sets. These operations are executed in parallel by processing a read with a GPU block with $s/2$ threads, i.e., the maximum number of pairs. Hence, each merge operation is performed by a single thread. The merge operation of Masher is slightly different than the one used by mergesort: if the values of two locations are the same (or *close* to each other) Masher adds the weight of the second to the weight of the first. And the next entry in the second set is visited. In the current implementation, the locations which has at most 28 bases between them are considered *close*. After processing the last pair, the final set is sent to CPU and sorted with respect to seeds' weights as shown in Figure 6(b).

3.3 GPU-accelerated local alignment

After obtaining the candidate locations and sorting them with respect to their weights for all the reads in parallel, Masher chooses a batch of promising read/location pairs, (e.g., all the ones with the best 5 weights), and executes a local alignment for each pair. For each read, the best location (w.r.t. a scoring scheme) is chosen from the current batch and reported. The remaining reads for which a valid alignment cannot be found in the current batch are processed in the next batch(es).

Masher uses a parametrized variant of the well-known

Smith-Waterman algorithm which is based on dynamic programming and employs a cost matrix [29]. We used the scoring scheme of [17] to calculate the values in the matrix and support affinity gap scoring. Given a parameter k, the modified SW algorithm uses a banded search space, and only the matrix cells (i, j) where $|i - j| \leq k$ are visited and scored. For efficiency purposes, Masher does two passes on the read/location pairs and sets k to 4 and 16 in the first and the second pass, respectively. Hence, in the first pass, only the alignments with a few number of errors are looked for. And if a read is not aligned in the first pass, alignments with more number of errors are considered in the second one.

The local alignment phase is run on the GPU where a GPU block performs multiple SWs in parallel and each SW is performed also in parallel by using k threads. At each step of the algorithm, the k matrix cells each computed by a different thread form a line which is parallel to the anti-diagonal of the dynamic programming matrix. During this phase, these intermediate cell scores are kept in shared memory for efficiency. Thanks to the synchronicity of GPUs, only k dynamic programming scores need to be kept in shared memory for a given SW computation. Therefore, using a small k (first 4, then 16) really helps fitting the data of many SW computations in the shared memory which is a very scarce resource. This allows to run a large amount of SWs concurrently on the GPU. In addition to shared memory, Masher uses the global memory to store the predecessor of each matrix cell, which is less critical since in this phase, it is only written but not read.

4. EXPERIMENTAL RESULTS

All the experiments run on a machine equipped with an Intel core i7-960 CPU clocked at 3.2 Ghz. There are 4 Hyper-Threading cores (8 threads in total) and 24GB of DDR3 memory. The machine is equipped with an NVIDIA Tesla K20c GPU featuring 13 Streaming Multiprocessors (SM), 192 cores per SM clocked at 700 MHz (for a total of 2496 CUDA cores), and 4.8GB of global memory clocked at 2.6 GHz. ECC is enabled. On the software side, the machine runs CentOS 5.8 with Linux 2.6.18. The code was compiled using CUDA 5.0 and gcc 4.2.4.

The human genome we used in the experiments is `hg19` which has $N = 3.137 \times 10^9$ bases. We used the wgsim simulator [12] to generate the reads with $L \in \{100, 300, 500, 1000\}$ and error rate $\epsilon \in \{2\%, 4\%, 6\%, 8\%\}$. For the simulator, the fraction of indels is 15% and the probability of an indel extension is 30%. For each of these configurations, we generated 100K single-end reads and evaluate the performance of Masher by comparing it with Bowtie2 (sensitive and fast modes), SOAP3-dp, and CUSHAW2-GPU. For Bowtie2 and Bowtie2-fast, we used shared memory parallelism with 8 threads. For the other tools/modes, we used Tesla K20. The amount of the memory on K20, which is a cutting edge GPU, was not sufficient for CUSHAW2-GPU to align reads longer than 320bp. Hence, we do not present its results for the configurations with $L \in \{500, 1000\}$. We experimented with two modes of Masher: the first one, Masher, uses $\Delta_R = 0.7\sqrt{L}$. The second one, Masher-fast, uses a larger read step size $\Delta_R = \sqrt{L}$. It also uses smaller candidate base location batches during the local alignment phase to reduce the number of SWs performed.

We used three metrics for comparison: the first metric, *sensitivity*, is the percentage of the aligned reads. The second metric, *accuracy*, is the ratio of the number of correctly aligned reads to the number of aligned reads. Hence, the product of sensitivity and accuracy gives the percentage of the correctly aligned reads. We assume that a read is correctly aligned if the left most position is within 50bp of the simulated location. The third metric is the execution time: to be fair with all the tools, we only measured the alignment time and excluded all other operations such as I/O. The time spent for index construction is also not included.

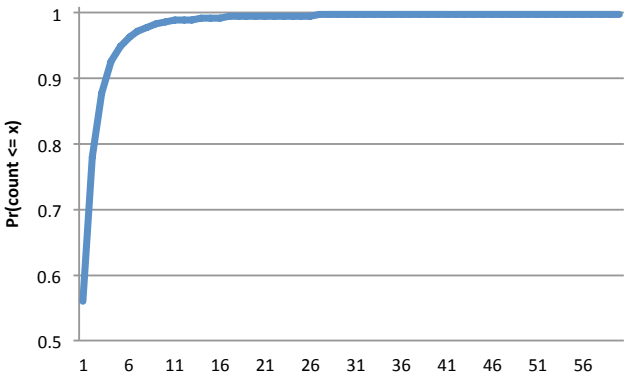


Figure 7: The cumulative distribution function for the number of appearances of a seed in the reference. To draw the plot, only the seeds used during the indexing phase, i.e., the ones which appear at least one location divisible by Δ_G of the reference genome, are taken into the account. A point (x, y) on the plot shows that a random seed (in the used seed set) exists in at most x locations of the genome with y probability.

We tried to be as consistent as possible while setting the parameters for each tools. For index construction, their default parameters are used. While computing the overall alignment score, a match is awarded with 2 points. On the other hand, each mismatch, gap, and gap extension are penalized with -1, -2, and, -1 points, respectively. The same values are used for Masher, Bowtie2, and SOAP3-db. For CUSHAW2-GPU, we use the default configuration. The lower bound for a valid alignment score is set to

$$score_{LB} = L * (2 - 0.1 \times (1 + \epsilon/0.02)).$$

This score is used for Bowtie2 and SOAP3-db in addition to Masher. We keep the default value for CUSHAW2-GPU since its results were much better when used in the default configuration.

We first analyze how our restriction and limitation on the number of maximum seed locations affect Masher. The limitation arises from GPU's memory bottleneck: to reduce the size of the index, the maximum number of locations indexed/stored per seed is set to 255. And the restriction arises from our efficiency concerns: Masher ignores the seeds with more than $\omega = 60$ locations in the first alignment pass. Note that in the second pass, for the remaining, unaligned reads, ω is set to 255, hence all the locations in the index are used. As Figure 7 shows, the percentage of the seeds which appear at more than 60 reference locations is insignificant. In fact, even with $\omega = 20$, Masher may not have an accuracy problem in its first pass. As explained in the previous sec-

tion, in many of our GPU kernels, each seed is processed by a single thread and the amount of work a thread performs is proportional to the number of locations stored in the index for that seed. As explained in Section 2.1, the threads in a warp wait for each other even if they remain idle. Hence, ignoring the seeds with high location counts can be promising to reduce the load imbalance and improve the parallel performance obtained by a GPU.

Figure 8 shows the two metrics, accuracy and sensitivity, and their product for each read length/error rate configuration. As expected, the values of these metrics drop when the error rate increases. And they (usually) increase with the read length. Considering the percentage of the accurately aligned reads (the third part of each table), Masher, Bowtie2, and CUSHAW2-GPU perform better for $L = 100$ and $L = 300$ (we present the maximum value and the ones in its 2% neighborhood as bold). For longer reads, Masher and Bowtie2 continues to perform well. SOAP3-dp also performs good for small error rates. However, when the error rate increases, its sensitivity , the percentage of the reads it aligns, drops significantly. When the read length increases, there is not much difference between the normal and *-fast* versions of Masher and Bowtie2. It may be expected since the good and bad candidate locations can be distinguished easier when L is large. For example, for long reads, Masher uses more seeds and the maximum weight a candidate base location can have also increases. Hence, the weight spectrum will be larger. This will make the things easier for Masher that starts from the *good end* of this spectrum while performing local alignments. Overall, the quality of Masher's alignment is on-par or better than the other tested tools.

Figure 9 shows the alignment times (in seconds) of the tools in log scale for 100K reads and each read length/error rate configuration. Our first observation is similar to the one we have for the previous comparison for accuracy and sensitivity: when the reads get long enough, e.g., $L = 1000$, there is not much difference between the normal and fast modes of Masher. On the other hand, for short reads, e.g., $L = 100$ or 300, the fast modes are indeed faster. Another similar observation is, Masher's normal mode is affected by the error rate more than its fast mode (see for example the trend for $L = 300$ and $L = 500$). That is when ϵ increases both Masher's and Masher-fast's alignment time also increase. However, the increase is less for the fast mode.

As Figure 9 shows, for $L = 100$, the performance of the tools are close to each other. Still, Masher-fast and Bowtie2-fast are the fastest tools. For this setting, SOAP3-dp is faster than Bowtie2. And when $L = 300$ it is even faster than Masher and Bowtie2-fast. However, it is still 1.7 to 2.2 times slower than Masher-fast. When L increases, the difference between Masher and other tools becomes more clear. For example, when $L = 1000$, Masher and Masher-fast are around 22–23 times better than Bowtie2-fast which is the fastest non Masher-based tool in this setting. Hence, we can argue that for long(er) reads, Masher performs much faster than the other state-of-the-art tools we tested.

Figure 10(a) shows the changes on the alignment time of Masher when L and ϵ increases. As expected, the alignment time increases with the error rate. And it also increases with the read length except for a few cases. As we mentioned before, when L gets larger for a fixed ϵ, it can be easier to distinguish the good and bad candidates. As the figure shows, when L increases from 500 to 1000, Masher's alignment time reduces from 32 seconds to 22 seconds for $\epsilon = 0.08$. This reduction indeed is correlated with the reduction on the number of local alignments performed. For the configuration ($L = 500, \epsilon = 0.08$), Masher performs around $3,000$K SW operations whereas this number is only 330K for ($L = 1000, \epsilon = 0.08$). Although it is less visible, for $\epsilon = 0.04$, the alignment time also reduces from 24 to 20 seconds. Again, the reason is the reduction on the number of SW operations; this time from $1,800$K to 368K. These numbers show that the number of local alignments performed is the main bottleneck when we want to achieve a higher a throughput. Masher can be tuned to reduce the number of SWs without significantly reducing the accuracy. In fact, this is what we aimed while setting the parameters for the fast mode whose alignment time plot is shown in Figure 10(b). As expected, using smaller batches of the candidate base locations prevents the fast mode to perform a large amount of local alignments. As a result, the alignment time of Masher-fast increase with L. To be precise, for $\epsilon = 0.08$, the number of SWs performed is between 300K and 400K for both $L = 500$ and $L = 1000$.

5. CONCLUSION AND FUTURE WORK

In this paper, we introduced Masher, a fast and accurate short/long read mapper, which uses a novel alignment algorithm and memory efficient indexing scheme to reduce the size of a human genome index and to make it fit to the memory of a GPU. The results show that Masher produces accurate alignments. Its speed is competitive with the tested state-of-the-art tools for reads of length less than 300 and an order of magnitude faster when the reads are longer than 500.

In the close future, we aim to make the software publicly available. In addition, we want to improve Masher's performance further by using GPU-specific optimizations and with a better CPU/GPU pipelining. We also want to add new features such as a support for paired-end sequences or *fastq* format.

6. ACKNOWLEDGMENTS

This work was partially supported by the NHI/NCI grant R01CA141090 and NSF grant CNS-0643969. We also thank to NVIDIA for providing us the K20 card used in the experiments.

7. REFERENCES

[1] A. M. Aji, Z. Liqing, and W. Feng. GPU-RMAP: accelerating short-read mapping on graphics processors. In *Proc. CSE 2010*, pages 168–175, Dec. 2010.

[2] D. Bozdağ, C. C. Barbacioru, and U. V. Çatalyürek. Parallel short sequence mapping for high throughput genome sequencing. In *Proc. IPDPS 2009*. IEEE Computer Society, May 2009.

[3] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Systems Research Center, 1994.

[4] N. L. Clement, M. J. Clement, Q. Snell, and W. E. Johnson. Parallel mapping approaches for GNUMAP. In *Proc. IPDPSW 2011*, pages 435–443. IEEE Computer Society, May 2011.

Error rate	Masher	Masher -fast	Bowtie2	Bowtie2 -fast	SOAP3 -dp	CUSHAW2 -GPU
Sensitivity (Percentage)						
2%	99.23	97.55	98.80	98.00	98.50	99.90
4%	99.44	96.81	98.00	94.63	92.50	99.90
6%	99.36	94.50	96.00	88.80	81.70	98.80
8%	98.87	89.83	93.15	80.60	67.70	96.20
Accuracy (Percentage)						
2%	95.01	95.49	95.20	95.00	96.20	95.20
4%	93.82	94.44	94.00	93.78	95.50	94.30
6%	92.42	93.07	92.60	91.70	94.50	93.20
8%	90.84	91.49	91.10	89.47	93.00	91.90
Sensitivity × Accuracy (Percentage)						
2%	**94.28**	**93.15**	**94.06**	**93.10**	**94.76**	**95.10**
4%	**93.29**	91.43	92.12	88.74	88.34	**94.21**
6%	**91.83**	87.95	88.90	81.43	77.21	**92.08**
8%	**89.81**	82.19	84.86	72.11	62.96	**88.41**

(a) Read length = 100

Error rate	Masher	Masher -fast	Bowtie2	Bowtie2 -fast	SOAP3 -dp	CUSHAW2 -GPU
Sensitivity (Percentage)						
2%	99.92	99.78	99.90	99.70	99.80	100.00
4%	99.84	99.55	99.80	99.24	99.40	100.00
6%	99.75	99.17	99.60	97.70	97.40	99.90
8%	99.54	98.22	99.20	93.90	91.10	99.70
Accuracy (Percentage)						
2%	97.19	97.62	97.90	97.70	98.40	97.90
4%	96.83	97.25	97.60	97.50	99.30	97.40
6%	96.33	96.88	97.50	97.00	97.60	96.40
8%	95.58	96.37	97.10	96.30	97.00	94.80
Sensitivity × Accuracy (Percentage)						
2%	**97.11**	**97.41**	**97.80**	**97.41**	**98.20**	**97.90**
4%	96.68	**96.81**	**97.40**	**96.76**	**98.70**	**97.40**
6%	**96.09**	**96.08**	**97.11**	94.77	95.06	**96.30**
8%	**95.14**	**94.65**	**96.32**	90.43	88.37	**94.52**

(b) Read length = 300

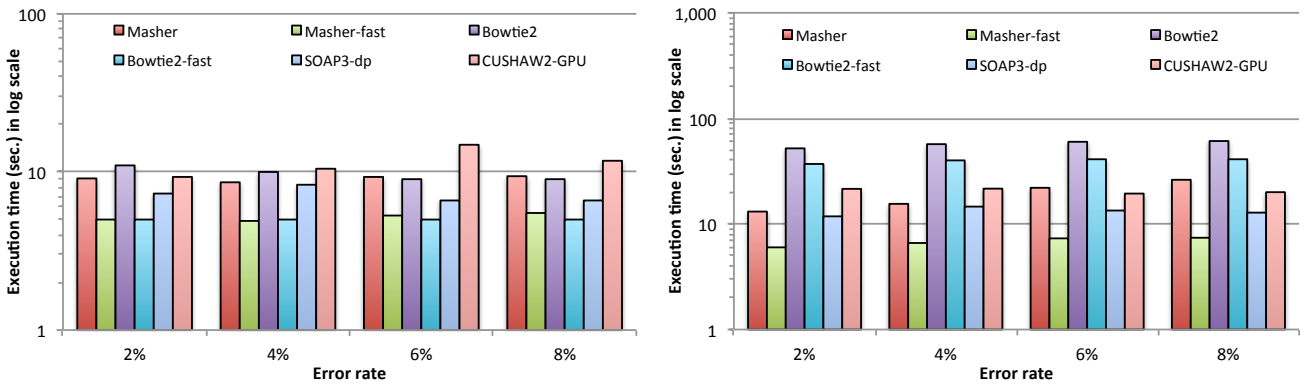
Error rate	Masher	Masher -fast	Bowtie2	Bowtie2 -fast	SOAP3 -dp
Sensitivity (Percentage)					
2%	99.89	99.89	99.90	99.90	99.20
4%	99.84	99.78	99.90	99.80	94.30
6%	99.74	99.51	99.90	99.34	75.30
8%	99.62	98.93	99.90	97.70	48.60
Accuracy (Percentage)					
2%	97.69	97.78	98.20	98.10	98.80
4%	97.20	97.19	98.00	97.80	98.50
6%	96.83	96.83	97.80	97.60	98.30
8%	96.25	96.15	97.40	97.00	98.00
Sensitivity × Accuracy (Percentage)					
2%	**97.58**	**97.67**	**98.10**	**98.00**	**98.01**
4%	**97.04**	**96.98**	**97.90**	**97.60**	92.89
6%	**96.58**	**96.36**	**97.70**	**96.96**	74.02
8%	**95.88**	95.12	**97.30**	94.77	47.63

(c) Read length = 500

Error rate	Masher	Masher -fast	Bowtie2	Bowtie2 -fast	SOAP3 -dp
Sensitivity (Percentage)					
2%	100.00	99.80	99.99	99.90	99.30
4%	100.00	99.73	99.90	99.90	98.70
6%	100.00	99.53	99.90	99.80	91.40
8%	100.00	98.93	99.80	99.50	68.90
Accuracy (Percentage)					
2%	98.50	98.25	98.50	98.50	98.90
4%	98.28	97.78	98.30	98.10	98.50
6%	97.86	97.24	97.50	97.30	97.80
8%	97.41	96.66	96.43	96.10	96.00
Sensitivity × Accuracy (Percentage)					
2%	**98.50**	**98.05**	**98.49**	**98.40**	**98.21**
4%	**98.28**	**97.52**	**98.20**	**98.00**	**97.22**
6%	**97.86**	**96.78**	**97.40**	**97.11**	89.39
8%	**97.41**	**95.63**	**96.24**	**95.62**	66.14

(d) Read length = 1000

Figure 8: Evaluation of the result quality of the tools used in the experiments for various (L, ϵ) configurations. For each configuration, 100K reads are used. *Sensitivity* is the percentage of the reads aligned by the tool within the 100K reads. *Accuracy* is the percentage of the correctly mapped reads within the aligned reads. Hence, the product of accuracy and sensitivity is equal to the percentage of the correctly aligned reads within whole read set. For each configuration, the maximum value of the product and the ones in its 2% neighborhood are shown in bold. By using the 4.8GB memory in K20, CUSHAW2-GPU can only process at most 320bp. Hence, it is not included in tables (c) and (d) for read lengths 500bp and 1000bp, respectively.



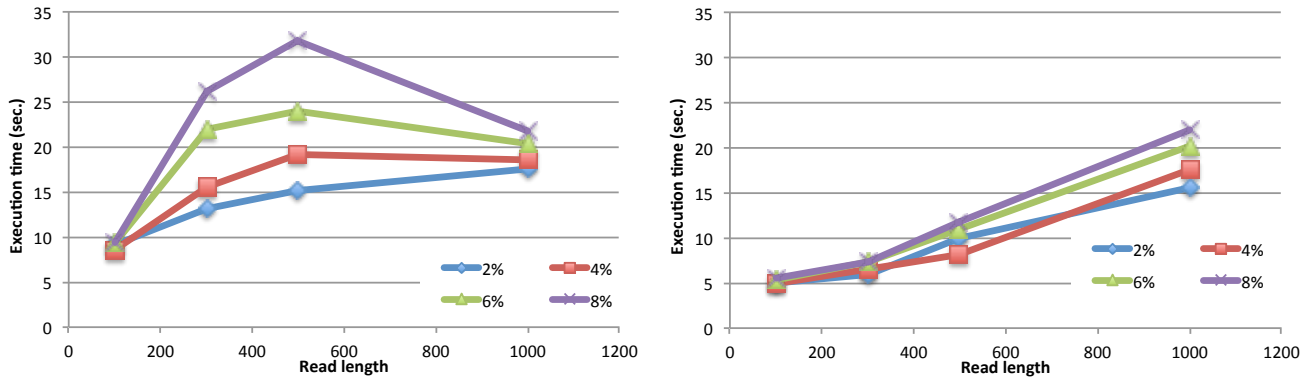
(a) Read length = 100bp

(b) Read length = 300bp

(c) Read length = 500bp

(d) Read length = 1000bp

Figure 9: Log-scaled execution times (in seconds) of the aligners used in the experiments for various (L, ϵ) configurations. For each configuration, 100K reads are used. By using the 4.8GB memory in K20, CUSHAW2-GPU can align the reads with length at most 320. Hence, it is not included in the last two figures for read length 500 and 1000.



(a) Masher

(b) Masher-fast

Figure 10: Alignment time of Masher w.r.t. various (L, ϵ) configurations.

[5] N. L. Clement, Q. Snell, M. J. Clement, P. C. Hollenhorst, J. Purwar, B. J. Graves, B. R. Cairns, and W. E. Johnson. The GNUMAP algorithm: unbiased probabilistic mapping of oligonucleotides from next-generation sequencing. *Bioinformatics*, 26(1):38–45, 2010.

[6] M. David, M. Dzamba, D. Lister, L. Ilie, and M. Brudno. SHRiMP2: sensitive yet practical short read mapping. *Bioinformatics*, 2011.

[7] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS 2000*, pages 390–. IEEE Computer Society, 2000.

[8] N. Homer, B. Merriman, and S. F. Nelson. BFAST: an alignment tool for large scale genome resequencing. *PLoS ONE*, 4(11):e7767, Nov. 2009.

[9] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(2):R12, 2004.

[10] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie2. *Nature Methods*, 9:357–359, 2012.

[11] 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(3):R25+, 2009.

[12] H. Li. wgsim. Accessed, May 2013.

[13] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, July 2009.

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

[15] H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, Sept. 2010.

[16] 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. Lil. SOAP3: GPU-based compressed indexing and ultra-fast parallel alignment of short reads. In *Proc. MASSIVE 2011*, June 2011.

[17] Y. Liu, D. L. Maskell, and B. Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73+, 2009.

[18] Y. Liu and B. Schmidt. Long read alignment based on maximal exact match seeds. *Bioinformatics*, 28(18):i318–i324, 2012.

[19] R. Luo, T. Wong, J. Zhu, C.-M. Liu, E. Wu, L.-K. Lee, H. Lin, W. Zhu, D. W. Cheung, H.-F. Ting, S.-M. Yiu, C. Yu, Y. Li, R. Li, and T.-W. Lam. SOAP3-dp: fast, accurate and sensitive GPU-based short read aligner. Technical Report 1302.5507v2, arXiv, Feb. 2013.

[20] N. Malhis, Y. S. N. Butterfield, M. Ester, and S. J. M. Jones. Slider-maximum use of probability information for alignment of short sequence reads and snp detection. *Bioinformatics*, 25(1):6–13, 2009.

[21] N. Malhis and S. J. M. Jones. High quality snp calling using illumina data at shallow coverage. *Bioinformatics*, 26(8):1029–1035, 2010.

[22] S. Misra, R. Narayanan, W.-K. Liao, A. N. Choudhary, and S. Lin. pFANGS: Parallel high speed sequence mapping for next generation 454-roche sequencing reads. In *Proc. IPDPSW 2010*. IEEE Computer Society, 2010.

[23] S. Misra, R. Narayanan, S. Lin, and A. N. Choudhary. FANGS: high speed sequence mapping for next generation sequencers. In *Proc. SAC 2010*, pages 1539–1546, 2010.

[24] J. C. Mu, H. Jiang, A. Kiani, M. Mohiyuddin, N. Bani Asadi, and W. H. Wong. Fast and accurate read alignment for resequencing. *Bioinformatics*, 28(18):2366–2373, Sept. 2012.

[25] G. Rizk and D. Lavenier. Gassst: global alignment short sequence search tool. *Bioinformatics*, 26(20):2534–2540, 2010.

[26] 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 Biology*, 5(5):e1000386+, May 2009.

[27] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, page 474, 2007.

[28] A. D. Smith, Z. Xuan, and M. Q. Zhang. Using quality scores and longer reads improves accuracy of solexa read mapping. *BMC Bioinformatics*, 9(1):128+, Feb. 2008.

[29] T. Smith and M. Waterman. Identification of common molecular subsequences. *J. Molecular Biology*, 147:195–197, 1981.

[30] P. Weiner. Linear pattern matching algorithms. In *Proc. SWAT 1973*, pages 1–11, Oct. 1973.

[31] C. Ye, Z. Ma, C. Cannon, M. Pop, and D. Yu. Exploiting sparseness in de novo genome assembly. *BMC Bioinformatics*, 13(Suppl 6):S1, 2012.