CUSHAW2-GPU: Empowering Faster Gapped Short-Read Alignment Using GPU Computing

Yongchao Liu and Bertil Schmidt
Johannes Gutenberg Universität Mainz

Editor's notes:
Aligning short-reads to a genome is often essential to many applications of next-generation sequencing (NGS) and has motivated the development of a variety of short-read aligners. This paper presents CUSHAW2-GPU, a parallelized, fast and accurate gapped short-read aligner based on GPU computing. This aligner is designed to accelerate the CUSHAW2 algorithm proposed by the authors earlier, using CUDA enabled GPUs.
—Partha Pratim Pande, Washington State University

Aligning short reads to a genome is often essential to many applications of next-generation sequencing (NGS) and has motivated the development of a variety of short-read aligners. However, many existing aligners are designed and optimized for very short reads (typically $\leq$ 100 base pairs). They usually assume very small deviation between the reads and the genome. Therefore, they typically perform only ungapped alignments, or gapped alignments allowing a very limited number of gaps (typically one gap). Moreover, for the major NGS platforms, such as Illumina, 454, and Ion Torrent, the maximum or average read lengths are steadily increasing beyond 100, e.g., the Illumina MiSeq sequencers can already produce 250-bp paired-end (PE) reads as of writing this paper. However, these longer short reads usually come at the expense of higher sequencing error rates, and thus require gapped aligners supporting more mismatches and gaps. In addition to sequencing errors, they are prone to have more true indels. In this case, many existing aligners will become inefficient in terms of alignment quality, speed, or even both, when handling such longer short reads with larger deviation from the genome.

Several gapped aligners for longer short reads have been developed, including BWA–SW [1], Bowtie2 [2], CUSHAW2 [3], and GEM [4]. All these aligners are designed intrinsically following the seed-and-extend heuristic by indexing the genome using Burrows–Wheeler transform (BWT) and Ferragina–Manzini (FM) index [5]. In this heuristic, a read is aligned by first finding seeds on the genome and then extending the alignment to the rest of the read using dynamic programming. Different aligners may employ different seeding policies. BWA–SW employs variable-length gapped seeds and Bowtie2 extracts fixed-length inexact-match seeds through ungapped alignments. CUSHAW2 identifies all maximal exact match (MEM) seeds, while GEM adopts a filtration-based approximate string matching approach. BWA–SW, Bowtie2, and CUSHAW2 support mapping quality scores, whereas GEM does not. In addition, all four aligners have been parallelized for

Digital Object Identifier 10.1109/MDAT.2013.2284198
Date of publication: 02 October 2013; date of current version: 20 February 2014.

multicore CPUs using multithreading. Although usually fast, they still have difficulties in substantially meeting the requirements of ever-increasing sequence volume on a single multicore CPU. In this regard, it becomes attractive to resort to accelerators such as many-core GPUs. In a computer, we can gain some additional compute power beyond the CPU by plugging one or more GPUs into Peripheral Component Interconnect Express slots. This heterogeneous computing architecture enables users to offload some compute-intensive parts of a task from the CPU to the GPU, or to concurrently conduct multiple tasks on the two types of processing units. Some pioneer research, e.g., CUSHAW [6] and SOAP3 [7], has shown that short-read alignment can benefit from the heterogeneous computing with GPUs. In addition, field-programmable gate array, an alternative accelerator, has already been used to accelerate short-read alignment. Unfortunately, publicly available implementations (see [8]) merely support small genomes.

We present CUSHAW2–GPU, a parallelized, fast, and accurate gapped short-read aligner based on GPU computing. This aligner is designed to accelerate our CUSHAW2 algorithm [3] using CUDA-enabled GPUs. In this aligner, we have investigated an intertask hybrid CPU–GPU parallelism mode to fully exploit the compute power of both CPUs and GPUs, which conducts concurrent alignments of different reads on the two types of processing units. Moreover, a new tile-based Smith–Waterman (SW) [9] alignment backtracking algorithm has been devised using CUDA to facilitate fast alignments. We have compared the performance of CUSHAW2–GPU to BWA–SW, Bowtie2, and GEM by aligning both simulated and real reads to the human genome. On simulated data, our aligner achieves superior recall and precision to the three aligners for both single-end (SE) and PE alignments. On real data, it has demonstrated competitive or better sensitivity. As for speed, our aligner with a Tesla K20c GPU runs up to 2.8×, 4.2×, 1.7×, and 2.0× faster than CUSHAW2, BWA–SW, Bowtie2, and GEM on the 12 cores of a high-end CPU for the SE alignments, and up to 1.6×, 2.7×, 1.3×, and 2.1× faster for the PE alignments, respectively.

GPU architecture

A CUDA-enabled GPU can be conceptualized as a fully configurable array of scalar processors (SPs). These SPs are further organized into a set of streaming multiprocessors (SMs) under three architecture generations: Tesla, Fermi, and Kepler. CUSHAW2–GPU is optimized for the Kepler architecture and, hence, it is fundamental to understand the features of the underlying hardware and the associated parallel programming model.

For Kepler, each SM comprises 192 SP cores sharing a configurable 64-kB on-chip memory. The on-chip memory can be configured at runtime as 48-kB shared memory with 16-kB L1 cache, 32-kB shared memory with 32-kB L1 cache, or 16-kB shared memory with 48-kB L1 cache, for each CUDA kernel. This architecture has a local memory size of 512 kB per thread and has an L1/L2 cache hierarchy with a size-configurable L1 cache per SM and a dedicated unified L2 cache of size up to 1536 kB. However, L1 caching in Kepler is reserved only for local memory accesses such as register spills and stack data. Global memory loads can only be cached in L2 cache and the 48-kB read-only data cache. Same as all previous architectures, threads launched onto a GPU are scheduled in groups of 32 parallel threads, called warps, in single-instruction–multiple-thread fashion.

Intertask hybrid CPU–GPU parallelism

Hybrid CPU–GPU parallelism is the common parallelism model for GPU-based applications. However, all existing GPU-based short-read aligners only focus on intratask parallelism, i.e., dispatching highly parallel and compute-intensive portions of a single task onto GPUs and leaving the rest to CPUs. This intratask model usually leads to a waste of the compute power of multicore CPUs, as the portions executed on CPUs are usually inherently sequential or have limited parallelism. Therefore, we employ an intertask hybrid CPU–GPU parallelism model to fully exploit the compute capability of multicore CPUs, in which both CPUs and GPUs conduct alignments of different reads. In the model, each CPU thread employs streaming SIMD extension (SSE) instructions as in CUSHAW2. For CUSHAW2–GPU, an instance (i.e., a single process) supports only a single GPU and multi-GPU support can be realized by executing multiple instances.

Mapping using CUDA

In general, a GPU thread works in three major stages (see Figure 1): 1) seed generation; 2) top seeds selection; and 3) final alignments generation, for both SE and PE alignments. An additional step in

stage 3) is required for PE alignments to pair reads from the top-scoring alignments. In addition, a CPU thread exactly follows the alignment pipeline as in CUSHAW2.

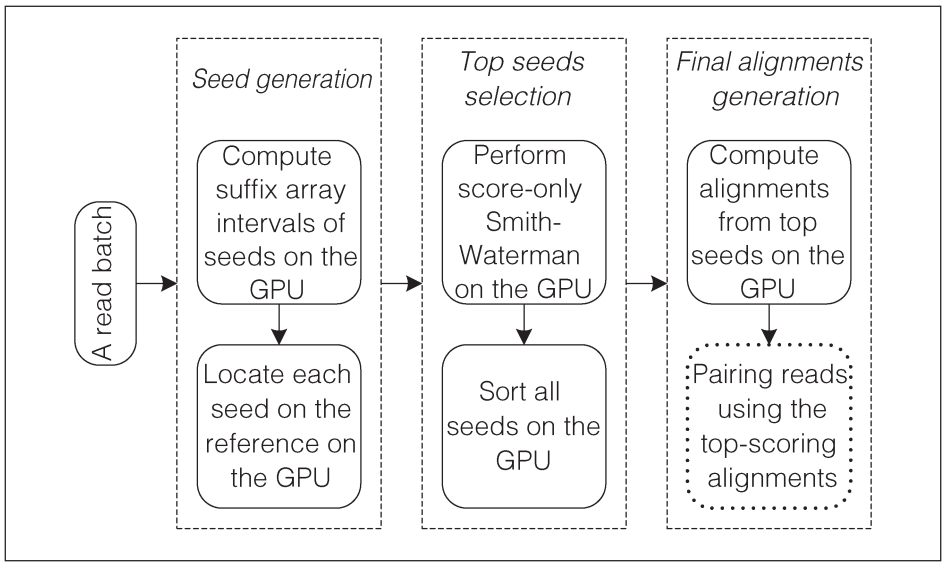

Figure 1. Program outline of the GPU thread in CUSHAW2–GPU.

Seed generation

The MEM seeds between a read S and the genome can be identified through a unidirectional substring search based on the BWT and FM index [3]. This stage consists of two steps: computing suffix array intervals (SAIs) of seeds and locating seeds on the genome. A SAI is an index range of a suffix array, and its computation relies on the reverse BWT and its reduced FM index [3]. When searching MEMs, the BWT and FM-index data are frequently accessed, but do not show good data locality. Due to their large memory footprint for a large genome, they can be stored either in global memory or in texture memory. On Kepler-based GPUs, both texture memory and read-only global memory can be cached by both the dedicated L2 cache and the 48-kB read-only cache. As texture cache is generally optimized for 2-D spatial locality and streaming fetches, we have stored the BWT and FM-index data in texture memory. For the resulting SAIs, we require allocating a writable result buffer in global memory before launching the kernel. However, different reads usually have different number of SAIs. Fortunately, we have found that given a minimum seed length k, the number of seed SAIs of S cannot exceed $|S| - k + 1$ for each strand. This observation enables us to calculate the peak memory size of the buffer for a read batch.

The second step locates each seed occurrence on the genome, which resorts to the BWT and its reduced suffix array (RSA). We have stored the RSA in read-only mapped memory due to its large memory footprint and very few accesses during the computing. This mapped memory is physically located on the host, but can be accessed by CUDA kernels as normal device memory. For a single SAI, it might cover a large number of seeds. Hence, it is usually impossible to locate all seeds of a read batch within a single kernel launch due to the limited available device memory. In this case, we organize all seeds into a set of small batches and then schedule all batches one by one onto the GPU. An intermediate file is used to store all seed locations so as to reduce the host memory overhead.

Top seeds selection

For a read, we extend each of its seeds in both directions on the genome to calculate the range of the corresponding mapping region. Subsequently, we perform the score-only SW algorithm between the read and all mapping regions to rank the seeds in terms of optimal local alignment score.

This stage consists of two steps: score-only SW algorithm computation and seed ranking by sorting. For the SW algorithm computation, we require the genome on the GPU. To save device memory, we have encoded the genome by randomly converting unknown bases to known ones (negligible impact on alignment quality) and representing each base with 2 b. This encoded genome is stored in texture memory. A modified version of the GPU-based SW algorithm in [10] has been used in CUSHAW2–GPU. The differences come from two folds. First, we did not use the query profile. Instead, we calculate the substitution scores by directly comparing base values. Second, the intermediate buffers for the SW algorithm are allocated in local memory, instead of global memory. This is because writable global memory accesses can only be cached by the L2

cache, whereas local memory can be additionally cached by the L1 cache per SM.

After gaining optimal local alignment scores, we sort all seeds in the descendent order of score. This sorting is realized on the GPU by using the sort primitive in the Thrust library (http://thrust.github.com). Similar to stage 1), all seeds are also organized and scheduled in small batches in order to overcome the GPU device memory limitation. In our aligner, we will discard seeds whose scores are less than a minimal score threshold (default = 30) and keep all qualified seeds being stored in an intermediate file for future use.

Tile-based SW alignment backtracking using CUDA

Very limited research was conducted on GPU-based SW alignment backtracking. Liu et al. [11] proposed to use a linear-space global alignment algorithm. However, for local alignment, it requires two additional runs of score-only SW algorithm to gain the global alignment range before tracing back the alignment path. Furthermore, this linear-space algorithm has a quadratic time complexity for alignment backtracking. Blazewicz et al. [12] proposed to directly record alignment moves while performing local alignments. This approach stores alignment moves in four bitwise backtracking matrices and has a linear time complexity for alignment backtracking. Although the backtracking matrices take much GPU device memory, this approach has been shown to work well for short protein sequences.

For our aligner, we have proposed a new tile-based alignment backtracking for the SW algorithm on the GPU (Figure 2 shows the pseudocode), which mainly follows the parallelization model in [10]. For a sequence pair S_1 and S_2, our approach allocates a single backtracking matrix of size $|S_1| \times |S_2|$ with 2 b representing each cell, since each cell depends on its left, upper, and upper left neighbors. Due to the small lengths of short reads, we can afford the memory overhead of the backtracking matrix, which are allocated in local memory in order to benefit from the L1/L2 cache hierarchy. In our implementation, both the alignment and backtracking matrices are partitioned into small tiles of size 4×4. The whole SW algorithm computation is conducted by computing all tiles one by one. This tiled-based computation can significantly improve data access performance because of the following two reasons. First, all alignment moves in a single tile can be represented by a 32-b integer because each cell takes 2 b. This means that only a single write to the backtracking matrix is needed for one tile computation, significantly

```
for (i=0; i<glen; i+=8){  /*reference fragment*/
    initialize related variables;
    load four consecutive bases starting from index i;
    load four consecutive bases staring from index i + 4;
    for (j=0; j<rlen; j+=4){  /*short-read*/
        load the packed 4 bases with indices [j, j+3] to a register variable rB.
        /*the coordinate of tile T1 is (j/4, i/4) and of tile T2 is (j/4, i/4+1)*/
        resetting register variables rA1 and rA2 storing alignment moves for tiles T1 and T2.
        for (k=0; k<4; k++){
            get the (j+k)-th base of the read from rB.
            load the alignment scores of cell (j+k,i-1) from the buffer in local memory.
            compute the k-th column of T1, and save the alignment moves in rA1.
            compute the k-th column of T2, and save the alignment moves in rA2.
            save the alignment scores of cell (j+k, i+7) to the buffer in local memory.
        }
        save rA1 to the alignment backtracking matrix in local memory.
        save rA2 to the alignment backtracking matrix in local memory.
    }
}
```

Figure 2. Pseudocode of the tile-based SW algorithm supporting alignment backtracking.

reducing the number of writes to external device memory. Second, the data access performance to the backtracking matrix can get improved while tracing back the alignment. This is because all alignment moves in a single tile, represented as a 32-b integer, can be realized by only a single data fetch from the backtracking matrix. While tracing back the alignment, if the next cell lies in the same tile with the current cell, we can reuse the current tile value with no need of reloading. Albeit the existence of caches, the alignment backtracking still can benefit from our data reuse, as the current tile value is possibly swapped out of the caches.

Read pairing

Our read pairing mainly follows CUSHAW2. However, we choose to pair reads from their real mapping positions, rather than use the seed-pairing heuristic. For a read, the real mapping positions are calculated from all its top-scoring seeds using the aforementioned tile-based SW implementation. In addition, the read rescuing from aligned mates is carried out on the host, since it does not often happen.

Algorithmic differences between CUSHAW2–GPU and CUSHAW2, and other limitations

CUSHAW2–GPU does not produce completely identical results to CUSHAW2 for the following three reasons. First, for the SE alignment, CUSHAW2–GPU possibly selects a different top-scoring seed for a read if there is more than one seed with the same alignment score. In terms of PE alignment, CUSHAW2–GPU directly performs read paring from real mapping positions, rather than use the seed-paring heuristic as in CUSHAW2. Second, CUSHAW2–GPU uses the full SW algorithm to obtain optimal local alignments, whereas CUSHAW2 uses the banded one. Finally, a read possibly has more than one optimal local alignment to a mapping region, causing the final alignment to be implementation dependent.

Some limitations also exist in CUSHAW2–GPU. First, it assumes that the BWT and FM-index data can be entirely stored in texture memory. Second, ambiguous bases in the genome are randomly converted to known ones in order to save device memory. Finally, the memory footprint of the SW alignment backtracking algorithm scales quadratically with the maximum allowable short-read length (default = 320 and configurable at compile time). For the human genome, we require $\leq$ 4-GB GPU device memory and about 6-GB host resident memory. Although designed and optimized for the Kepler architecture, CUSHAW2–GPU still works on earlier generation Fermi-based GPUs.

Performance evaluation

We have evaluated the performance of CUSHAW2–GPU (v2.1.8-r16) by aligning simulated and real reads to the human genome (hg19). This performance is further compared to CUSHAW2 (v2.1.8), BWA–SW (v0.6.2-r126), Bowtie2 (v2.0.6), and GEM (v1.366). In terms of alignment quality, we have used the recall and precision measures for all simulated data and sensitivity for all real data. Recall (precision) is calculated by dividing the number of correctly aligned reads by the total number of reads (the number of aligned reads), while sensitivity is the proportion of reads that are aligned. For a read, GEM and BWA–SW might report > 1 alignment. In such cases, we only consider the first alignment occurrence of the read in the Sequence Alignment/Map (SAM) output for the two aligners. For all alignment-quality evaluations, CUSHAW2–GPU was executed only on the GPU. For simplicity, we will use "proposed" to represent CUSHAW2–GPU in Tables 2 and 3, and have also highlighted the best values in bold.

All tests are conducted on a hex-core 12-thread Intel Xeon E5-2620 2.0-GHz CPU, a Tesla K20c GPU, and 16-GB RAM. The GPU comprises 13 SMs (2496 cores) with a core frequency of 705.5 MHz and 5-GB device memory. We have turned off the error-correcting code for all GPU-based tests. All aligners have used the default settings for each data set, except for the insert-size related information for PE alignments.

Alignment quality on simulated reads

To assess alignment quality, we have simulated nine Illumina-like PE data sets, from the human genome using *wgsim* in SAMtools v0.1.18 (http://samtools.sourceforge.net), which have different read lengths (100, 150, and 250) and mean base error rates (2%, 4%, and 6%). Each data set comprises one million read pairs with insert-sizes drawn from normal distribution $N(500, 50)$, $N(700, 70)$, and $N(1000, 100)$ for the 100-bp, 150-bp, and 250-bp reads, respectively. For a read, an alignment is

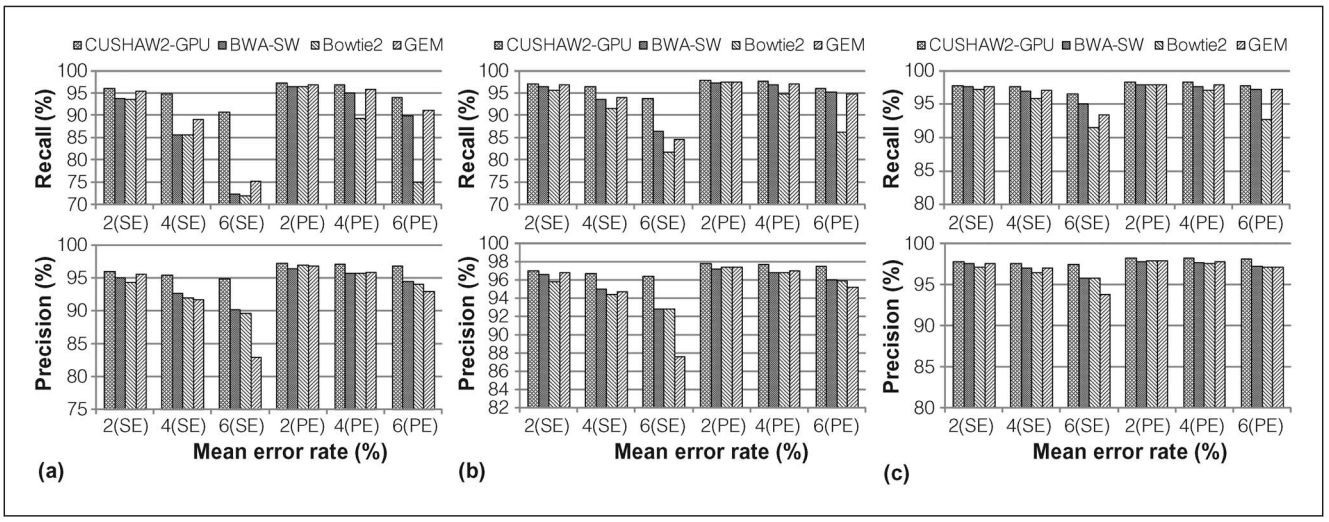

Figure 3. Alignment quality on Illumina-like reads: (a) 100 bp; (b) 150 bp; and (c) 250 bp.

deemed to be correct if the distance between the mapping and the true positions does not exceed 10.

As CUSHAW2–GPU keeps the alignment quality of CUSHAW2 (see Table S1 in the supplementary data), CUSHAW2 was excluded from the following alignment quality assessment. Figure 3 shows the alignment quality comparison between all evaluated aligners. CUSHAW2–GPU yields superior recall and precision to all other aligners for all evaluations. Moreover, it gets more superior as the error rate becomes larger. For data sets of the same read length, each aligner experienced performance drops for increasing error rates. Furthermore, the SE alignments are observed to have greater performance drops than the PE alignments for each aligner. This suggests that the PE information usually enables each aligner to have higher tolerance in sequencing errors. For data sets of the same error rate, each aligner gets its performance improved as the read length grows.

Table 1 Real data set information.

Type	Name	#Reads	Max	Mean	Insert
454	SRX000706	4 161 822	300	299±6	–
Ion Torrent	SRX202788	3 847 839	300	183±33	–
Illumina	SRX064195	5 093 7050	100	100±0	302
Illumina	ERX009608	53 653 010	102	101±1	313

Table 2 Alignment results on real data sets.

Data set	Proposed	BWA-SW	Bowtie2	GEM
		SE		
SRX000706	87.70	**87.92**	89.63	–
SRX202788	96.31	95.86	**97.55**	52.57
SRX064195	**96.75**	96.23	96.67	93.14
ERX009608	96.73	96.16	**96.77**	92.50
		PE		
SRX064195	**96.95**	96.88	96.87	95.17
ERX009608	96.92	96.76	**96.97**	95.07

Alignment quality on real reads

Furthermore, we have assessed all aligners using four real data sets from 454, Ion Torrent, and Illumina sequencers, respectively. All four data sets are publicly available and named after their accession numbers in the National Center for Biotechnology Information (NCBI) Sequence Read Archive (SRA; see Table 1 and the supplementary downloadable multimedia material available along with the article in IEEEXplore as provided by the authors.).

Table 2 shows the alignment results. For SRX000706, GEM failed due to memory allocation failure. BWA–SW performs best and CUSHAW2–GPU is second. For SRX202788, Bowtie2 is superior to all other aligners and CUSHAW2–GPU is the second best. For SRX064195, CUSHAW2–GPU performs best in

terms of SE and PE alignment. Bowtie2 outperforms BWA–SW and GEM for the SE alignment, while BWA–SW is better than Bowtie2 and GEM for the PE alignment. For ERX009608, Bowtie2 outperforms all other aligners and CUSHAW2–GPU is the second best for each case. For each applicable data set, GEM is the worst.

Table 3 Runtimes (in minutes) of all evaluated aligners.

Aligner	SRX000706	SRX202788	SRX064195	ERX009608
SE				
Proposed*	**9.0**	**4.8**	**31.1**	**34.2**
Proposed**	11.5	6.3	45.1	49.7
CUSHAW2	25.3	11.1	55.9	61.4
BWA-SW	37.7	17.8	92.1	96.4
Bowtie2	13.3	8.0	48.2	51.3
GEM	–	5.7	61.8	55.8
PE				
Proposed*	–	–	**35.6**	**39.3**
Proposed**	–	–	57.3	63.9
CUSHAW2	–	–	57.5	63.5
BWA-SW	–	–	95.2	98.7
Bowtie2	–	–	49.4	52.7
GEM	–	–	38.0	45.3

*means hybrid CUSHAW2-GPU with 12 extra threads and **means GPU-only one.

Speed evaluation

We have assessed the wall-clock runtime of each aligner using the data sets in Table 3. CUSHAW2, BWA–SW, Bowtie2, and GEM are run with 12 threads. Besides the GPU-only CUSHAW2–GPU, we have evaluated the hybrid CUSHAW2–GPU with 12 additional CPU threads. For GEM, we have counted in the SAM format conversion time (sometimes taking $> 50\%$ of the overall runtime) for fair comparisons, as every other aligner reports alignments in SAM format.

In terms of SE alignment, the GPU-only instance yields a speedup of 2.2 and 1.8 over CUSHAW2 for SRX000706 and SRX202788, respectively. For SRX064195 and ERX009608, the GPU-only instance runs 1.2× faster than CUSHAW2. These speedups suggest that the GPU-only instance seems to yield greater speedups as the average read length increases. The hybrid instance runs 1.3× faster than the GPU-only one for SRX000706 and SRX202788, and 1.5× faster for the remaining two data sets. On average, the hybrid instance achieves a speedup of 2.2, 3.4, 1.5, and 1.6 (with a maximum of 2.8, 4.2, 1.7, and 2.0) compared to CUSHAW2, BWA–SW, Bowtie2, and GEM, respectively. In terms of PE alignment, the GPU-only instance has a tie with CUSHAW2 for each data set, whereas the hybrid instance runs 1.6× faster than CUSHAW2. Compared to BWA–SW, Bowtie2, and GEM, the hybrid instance runs 2.7×, 1.4×, and 1.1× faster for SRX064195, and 2.5×, 1.3×, and 1.2× faster for ERX009608.

Runtime breakdown and workload distribution

We have further assessed the runtime proportion of each major stage of our GPU-only instance using the two aforementioned real Illumina data sets (see Table 4). In terms of SE and PE alignment, the seed generation and the top seed selection stages dominate the overall runtime for each data set. However, for the PE alignment, the alignment output and read pairing procedures are most time consuming, both of which are always run on CPUs. By comparing to the runtime of the SE alignments, we can roughly infer the runtime proportion taken by the read paring procedure of the PE alignments. In addition, we have evaluated the workload distribution between CPU and GPU threads of our hybrid instance (see Table 4). On average, the single GPU has processed 59.0% reads for the SE alignments and 50.3% reads for the PE alignments.

We have presented CUSHAW2–GPU, a CUDA-compatible gapped short-read aligner based on our CUSHAW2 algorithm. In this aligner, an intertask hybrid CPU–GPU parallelism has been investigated to concurrently align different reads on both multicore CPUs and GPUs. In addition, a new tiled-based SW alignment backtracking algorithm has been devised using CUDA to facilitate fast alignment. CUSHAW2–GPU accepts FASTA, FASTQ, SAM, and Binary sequence Alignment/Map formats as input, and reports alignments in SAM format.

Table 4 Runtime breakdown and workload distribution.

Assessment	ERX009608		SRX064195	
	SE	PE	SE	PE
Runtime breakdown (in %)				
Read loading	4.6	3.6	4.9	3.6
Seed generation	37.5	29.0	25.5	18.5
Top seed selection	38.4	31.1	49.3	36.6
SW alignment backtracking	1.6	2.7	1.7	3.3
Others*	17.9	33.6	18.6	38.0
Workload distribution (in %)				
#reads processed by the CPU threads	42.0	50.2	40.0	49.2
#reads processed by the GPU thread	58.0	49.8	60.0	50.8

*including alignment output for the SE alignment; and both alignment output and read pairing for the PE alignment.

We have assessed the performance of CUSHAW2–GPU and other aligners (CUSHAW2, BWA–SW, Bowtie2, and GEM), using simulated and real reads. On simulated data, CUSHAW2–GPU yields superior recall and precision to BWA–SW, Bowtie2, and GEM for both SE and PE alignments. On real data, CUSHAW2–GPU has demonstrated comparable or better sensitivity. As for speed, our aligner with a Tesla K20c GPU can achieve a speedup of up to 2.8, 4.2, 1.7, and 2.0 in comparison to the multithreaded CUSHAW2, BWA–SW, Bowtie2, and GEM on the 12 cores of a high-end CPU for the SE alignment, and up to 1.6, 2.7, 1.3, and 2.1 for the PE alignment, respectively. CUSHAW2–GPU and all simulated data are available at http://cushaw2.sourceforge.net.

References

[1] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 26, pp. 589–595, 2010.

[2] B. Langmead and S. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, pp. 357–359, 2012.

[3] Y. Liu and B. Schmidt, "Long read alignment based on maximal exact match seeds," *Bioinformatics* vol. 28, pp. i318–i324, 2012.

[4] S. Marco-Sola, M. Sammeth, R. Guigó, and P. Ribeca, "The GEM mapper: Fast, accurate and versatile alignment by filtration," *Nature Methods*, vol. 9, pp. 1885–1888, 2012.

[5] P. Ferragina and G. Manzini, "Indexing compressed text," *J. ACM*, vol. 52, no. 4, pp. 552–581, 2005.

[6] Y. Liu, B. Schmidt, and D. L. Maskell, "CUSHAW: A CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform," *Bioinformatics* vol. 28, pp. 1830–1837, 2012.

[7] C. M. Liu, T. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, and T.-W. Lam, "SOAP3: Ultra-fast GPU-based parallel alignment tool for short reads," *Bioinformatics*, vol. 28, pp. 878–879, 2012.

[8] Y. Chen, B. Schmidt, and D. L. Maskell, "A hybrid short read mapping accelerator," *BMC Bioinformatics*, vol. 14, no. 67, 2013, DOI: 10.1186/1471-2105-14-67.

[9] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Mol. Biol.* vol. 147, pp. 195–197, 1991.

[10] Y. Liu, B. Schmidt, and D. L. Maskell, "CUDASW++2.0: Enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions," *BMC Res. Notes*, vol. 3, no. 93, 2010, DOI: 10.1186/1756-0500-3-93.

[11] Y. Liu, B. Schmidt, and D. L. Maskell, "MSA-CUDA: Multiple sequence alignment on graphics processing units with CUDA," in *Proc. 20th IEEE Int. Conf. Appl.-Specific Syst. Architect. Processors*, 2009, pp. 121–128.

[12] J. Blazewicz, W. Frohmberg, M. Kierzynka, E. Pesch, and P. Wojciechowski, "Protein alignment algorithms with an efficient backtracking routine on multiple GPUs," *BMC Bioinf.*, vol. 12, no. 181, 2011, DOI: 10.1186/1471-2105-12-181.

Yongchao Liu is a Postdoctoral Researcher at the Institute of Computer Science, Johannes Gutenberg University Mainz, Mainz, Germany. His research interests focus on parallel and distributed algorithm design for bioinformatics and heterogeneous computing with accelerators. He has a PhD in computer

engineering from Nanyang Technological University, Singapore (2012).

Bertil Schmidt is a Professor of Computer Science at the Johannes Gutenberg University Mainz, Mainz, Germany. His research interests include parallel algorithms and architectures, bioinformatics, and high-performance computing. He has a PhD in computer science from Loughborough University, Loughborough, Leicestershire, U.K. (1999). He is a Senior Member of the IEEE.

■ Direct questions and comments about this article to Yongchao Liu, Institut für Informatik, Johannes Gutenberg Universität Mainz, 55099 Mainz, Germany; liuy@uni-mainz.de.