Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 1 (2012) 1129–1138

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

Quality-score guided error correction for short-read sequencing data using CUDA

Haixiang Shi[a*], Bertil Schmidt[a], Weiguo Liu[a], and Wolfgang Müller-Wittig[a]

School of Computer Engineering, N40-2-32a Nanyang Ave., Nanyang Technological University, Singapore 639798, {hxshi,asbschmidt,liuweiguo,askwmwittig}@ntu.edu.sg

Abstract

Recently introduced new sequencing technologies can produce massive amounts of short-read data. Detection and correction of sequencing errors in this data is an important but time-consuming pre-processing step for de-novo genome assembly. In this paper, we demonstrate how the quality-score value associated with each base-call can be integrated in a CUDA-based parallel error correction algorithm. We show that quality-score guided error correction can improve the assembly accuracy of several datasets from the NCBI SRA (Short-Read Archive) in terms of N50-values as well as runtime. We further propose a number of improvements of to our previously published CUDA-EC algorithm to improve its runtime by a factor of up to 1.88.

Keywords: DNA sequencing, CUDA, high-through short-read assembly, bioinformatics

1. Introduction

Second generation DNA sequencing technologies [8][9][16] open up a range of new opportunities for genome research. However, the generated datasets are of massive size, which poses a challenge for tools processing this data in terms of scalability. For example, the *Illumina Genome Analyzer IIx (IGA-IIx)* can currently generate an output of up to 25 billion base-pairs (bps) within a single run, which is expected to increase to 100 billion bps by the end of 2010. Further challenges arise from the facts that reads are short (between 35 and 100 bps for the IGA-IIx) and contain sequencing errors (IGA-IIx typically has a per base-pair error-rate of 1%-2%).

Recently introduced de-novo assemblers for *high-throughput short-read* (HTSR) sequencing data can be classified into two categories: overlap graph based and de Bruijn graph based. Edena [6] and Taipan [10] belong to the former, while the latter category includes tools like Euler-SR [2-4], Velvet [17], ALLPATHS [1], and ABYSS [14]. All these assemblers have in common that an exact overlap of length l is required to either build the prefix array or generate a link in the graph. Consequently, they are sensitive to sequencing errors. To deal with errors in reads, many algorithms have to either fix it separately in a preprocessing step before assembly [2-4], or include the

* Corresponding author. Tel.: +65-67906679; fax: +65-67928123
E-mail address: hxshi@ntu.edu.sg.

1877-0509 © 2012 Published by Elsevier Ltd.
doi:10.1016/j.procs.2010.04.125

procedure in the assembly, which introduces the extra overhead in runtime and memory in fragment assembly [6][17]. The former approach has obvious benefits but is time-consuming. For example, the profiling of Euler-SR shown in Table 1 shows that the error correction step can take up to 72% runtime of the overall assembly runtime. It can also be seen that the time spent in the pre-processing goes up with increasing error rate. Therefore, speeding up error correction is of high importance to research in this area.

Table 1. Runtime of error correction for Euler-SR with simulated read dataset consisting of 1.1 million reads of length 35 each (generated from *Saccharomyces cerevisiae chromosome V* with an error rate range between 1% and 3%) and its percentage of the overall runtime.

Error Rate	Runtime (sec.)	Percentage (%)
1%	1527	52%
2%	2506	63%
3%	3324	72%

In this paper we use CUDA to accelerate error correction by one to two orders of magnitude compared to the sequential Euler-SR program. We describe several improvements made to our previously published CUDA-EC software [12][13] to improve runtime as well as error correction accuracy by integrating quality-score into the correction algorithm. Firstly, we use quality-scores to trim the reads in a preprocessing step before constructing the spectrum, which leads to a spectrum with a lower number of false positives. Secondly, we only correct at positions with lower quality score (smaller than a threshold value) in the reads. The proposed parallel quality-score guided error correction method is evaluated in terms of speedup as well as in terms of N50-values and runtimes of the subsequent assembly.

The rest of this paper is organized as follows. In Section 2, we briefly review the spectral alignment approach to error correction. Features of the CUDA parallel programming model are highlighted in Section 3. Section 4 presents the sequential spectrum counting pre-processing method and the parallel error correction algorithm. The performance of our CUDA implementation is evaluated in Section 5. Finally, Section 6 concludes the paper.

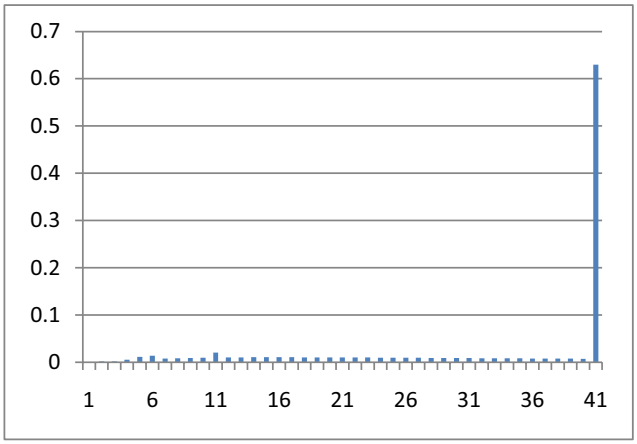
2. Spectral Alignment Problem

Consider a read r_i is of length L over the alphabet $\Sigma = \{A, C, G, T, N\}$, i.e. $r_i \in \Sigma^L$. Let $1 \leq l \leq L$ and let an l-mer (or l-tuple) be any substring of length l of r_i. Given a set of k reads $R = \{r_1,\ldots,r_k\}$ and a reference genome G, the *spectral alignment problem* (SAP) considers every l-mer of each read and compares them to the spectrum $T(G)$. The spectrum $T(G)$ consists of all l-mers of the reference genome sequence G, where the reads originate from. If a read is error-free, all its l-mers will have a corresponding exact match in $T(G)$. If a read has a single error (i.e. mutation) at position j, the corresponding $min\{l, j, L–j\}$ overlapping l-mers have (in most cases) a low number of corresponding exact matches in $T(G)$. However, by mutating position j back to the correct base-pair, all l-mers have a corresponding match.

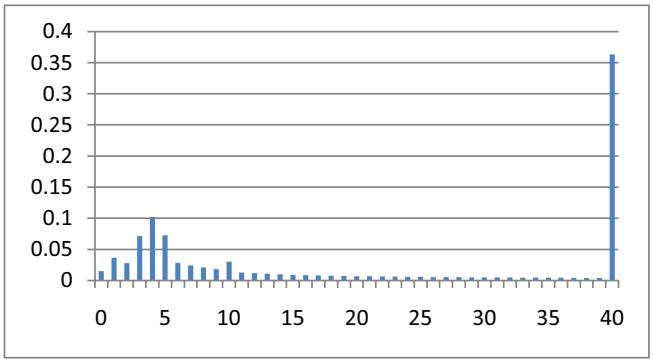
We use the following terminology for the definition of the SAP. An l-mer of a read is called *solid*, if it has an exact match in a given l-mer spectrum T and *weak* otherwise. A read r_i is called a T-string if all its l-mers have an exact match in the spectrum T. The SAP can now be defined as follows:

Definition (SAP): Given a read r_i and an l-mer spectrum T. Find a T-string r_i^* in the set of all T-strings that minimizes the distance function $d(r_i, r_i^*)$.

Depending on the error model of the utilized sequencing technology the distance function $d(\cdot,\cdot)$ can either be edit distance or hamming distance. Since we focus on Illumina sequencing technology which sequencing errors are almost completely mutations [5], the latter approach is chosen in this paper.



(a)



(b)

Fig. 1. Histogram of base quality values for base calls for the SRR006332 Acinetobacter sp. ADP1 dataset. (a) Percentage of correct base calls with a certain quality-score (b) Percentage of erroneous base calls with a certain quality-score.

In a de-novo genome assembly project the reference genome G is generally not known beforehand. Therefore, the spectrum $T(G)$ of all l-mers of G needs be approximated from the available read data. This is usually done by introducing the additional parameter m (*multiplicity*). The ideal spectrum $T(G)$ is then replaced by the approximated spectrum $T(R,m)$, where $T(R,m)$ consists of all l-mers that occur at least m times in R.

It should be mentioned that the use of an approximate spectrum is not always ideal, since

1. Some l-tuples that are in $T(G)$ might not necessarily be in $T(R,m)$ due to low coverage or sequencing errors (false negatives).
2. Some l-tuples that are in $T(R,m)$ might not necessarily be in $T(G)$ due to the same error occurring several times (false positives).

In this paper we try to reduce the number of false positives and false negatives by using the quality-score associated to each base-pair. The quality-score of a base-pair produced by an Illumina sequencer is an integer between 0 and 40. In theory, a quality-score of 40 corresponds to an expected probably of 0.01% of this base-pair being erroneous while a quality-score of 0 corresponds to an expected error probability of 50%. We use quality scores to perform quality trimming of reads, i.e., we will trim out the parts with lower quality score (smaller than a given threshold) and keep only high-quality parts to build a high-quality spectrum.

In order to judge the suitability of this approach, it is useful to first assess the correspondence between sequencing errors and low-quality score in practice. Therefore, we have mapped each read of an Illumina dataset

(SRA: SRR006332 *Acinetobacter sp. ADP1*) to its reference genome using RMAP [15]. The number of correct and wrong base calls were then determined and counted with respect to the respective quality-score values. The resulting histogram is shown in Fig.1. From this figure we can see that

- Correct base calls are less likely to appear at low quality scores (<6).
- The errors have high probability to occur at low quality scores in the range 0-10, but there is also a surge at quality value of 40.

3. CUDA Programming Model and Tesla Architecture

CUDA (Compute Unified Device Architecture) is an extension of C/C++ to write scalable multi-threaded programs for CUDA-enabled GPUs (Nickolls et al. 2008). CUDA programs can be executed on GPUs with NVIDIA's Tesla unified computing architecture (Lindholm et al. 2008). Examples of CUDA-enabled GPUs include the GeForce 8/9/200, Tesla 800/1000, and Quadro FX 3000/4000/5000 series.

CUDA programs contain a sequential part, called a kernel. The kernel is written in conventional scalar C-code. It represents the operations to be performed by a single thread and is invoked as a set of concurrently executing threads. These threads are organized in a hierarchy consisting of so-called thread blocks and grids A thread block is a set of concurrent threads and a grid is a set of independent thread blocks. Each thread has an associated unique ID (threadIdx,blockIdx) $\in \{0,\ldots,\text{dimBlock}-1\} \times \{0,\ldots,\text{dimGrid}-1\}$. This pair indicates the ID within its thread block (threadIdx) and the ID of the thread block within the grid (blockIdx). Similar to MPI processes, CUDA provides each thread access to its unique ID through corresponding variables. The total size of a grid (dimGrid) and a thread block (dimBlock) is explicitly specified in the kernel function-call:

kernel<<<dimGrid, dimBlock>>> (parameter list);

The hierarchical organization into blocks and grids has implications for thread communication and synchronization. Threads within a thread block can communicate through a per-block shared memory (PBSM) and may synchronize using barriers. However, threads located in different blocks cannot communicate or synchronize directly. In order to write efficient CUDA applications it is important to understand the different types of memory spaces in more detail.

- Readable and writable global memory is relatively large (typically around 1GB), but has high latency, low bandwidth, and is not cached.
- Readable and writable per-thread local memory is of limited size (16KB per thread) and is not cached. Access to local memory is as expensive as access to global memory.
- Read-only constant memory is of limited size (totally 64KB) and is cached. Reading from constant memory can be as fast as reading from a register.
- Read-only texture memory is large and is cached. Texture memory can be read from kernels using texture fetching device functions. Reading from texture memory is generally faster than reading from global or local memory.
- Readable and writable PBSM is fast on-chip memory of limited size (16KB per block). Shared memory can only be accessed by all threads in a thread block.
- Readable and writable per-thread registers is the fastest memory but is very limited.
- The Tesla architecture supports CUDA applications using a scalable processor array. The array consists of a number of streaming multiprocessors (SM). Each SM contains eight scalar streaming processor (SP) cores, which share a PBSM of size 16KB All threads of a thread block are executed concurrently on a single SM. The SM executes threads in small groups of 32, called warps, in single-instruction multiple-thread (SIMT) fashion. Thus, parallel performance is generally penalized by data-dependent conditional branches and improved if all threads in a warp follow the same execution path.

4. Parallel Error Correction

4.1. Overview

The quality-score guided error correction algorithm with CUDA consists of the following steps:

1. *Quality trimming and pre-computation of spectrum list*. Using the input parameters *l*, *m* and *h*, the reads are trimmed so that all the bases in the trimmed reads have a quality-score of at least *h*. A spectrum list consisting of all *l*-mers occurring at least *m* times in the trimmed read dataset is then compiled.
2. *Hashing the spectrum list into a Bloom filter*. Before transferring the spectrum to the GPU, all *l*-mers in the spectrum list are hashed into a Bloom filter. The Bloom filter is then bound to the texture memory on the GPU. Each item in the spectrum list is added to the Bloom filter using the following hash functions.

```
//Step 1: get hash value
 hash = hash_function_list[j](tempTuple,TUPLE_SIZE) % (table_size * char_size);
//Step 2: add to the bloom filter
 hash_table[hash / char_size] |= bit_mask[hash % char_size];
```

3. *Bind the Bloom filter to 1D texture on the GPU*:

```
// allocate data on device
unsigned char *dev_hash;
cudaMalloc( (void**)&dev_hash, table_size );
// copy memory to device
cudaMemcpy( dev_hash, hash_table, table_size, cudaMemcpyHostToDevice );
// bind texture
cudaBindTexture(0, tex, dev_hash );
```

4. *Allocate memory on the GPU and transfer all reads from CPU to GPU*: Reads are allocated in a one-dimensional character array and copied to device memory.
5. *Parallel error correction on GPU*: By applying different levels of parallelization we use multiple stages to perform data filtration. Firstly, we filter out the error-free reads through a procedure called "detect errors". In this stage, each thread works on a different character of a given read in parallel. Because reads are mapped into a one-dimensional array of characters, *L* threads will be used on each read where *L* is the read-length. After filtering out the error-free reads, the actual error correction procedure is performed on the remaining erroneous reads (i.e., voting and fixing/trimming). In the voting procedure, each CUDA thread works on each *l*-tuple to vote the read. Hence, *L*–*l*+1threads run in parallel on the same read. Finally, in the fixing procedure each thread works on each read in parallel to perform the actual error correction. Overall, the produced output is divided into four categories error-free reads (obtained in Stage 1), fixed reads, trimmed reads, and discarded reads (in Stage 3).
6. *Transfer from GPU to CPU and write results to files*. The output is transferred from GPU to CPU and written to two separate output files, one for fixed/trimmed and one for discarded reads.

4.2. Quality Trimming

Inputs are a set of reads R = {r1,…,rk} of length L each, a set of quality-score strings Q = {q1,…,qk} of length L each (where qi[j] (with $0 \le qi[j] \le 40$) is the quality-score associated with the base-pair ri[j] for all $1 \le i \le k$ and $0 \le j \le L-1$), the tuple-length l, and the quality score threshold h. The quality trimming procedure produces the longest substring of length at least l of each read ri such that all associated quality-scores with that substring are at least h. If no such substring exits, the corresponding read will not be considered for the subsequent spectrum computation.

The output of the quality trimming procedure is a file in "RAW" format, where each line of the file is a read fragment with all associated quality scores being at least h and of length at least l.

4.3. Spectrum construction using a counting Bloom filter

After quality trimming, the high quality part of reads will be used to build the spectrum The spectrum $T(R,m)$ consists of the set of all *l*–mers which have a multiplicity of at least *m*. We represent the spectrum by a counting Bloom filter data structure [13], denoted as $B(T(R,m))$, which is subsequently transferred to the CUDA texture memory in order to be used for parallel error correction. The computation of $B(T(R,m))$ is done sequentially on the host CPU. The counting bloom filter extends the bucket used in a conventional Bloom filter from a single bit to an *n*-bits counter. The value at a filter position can be increased to count the number of times that the key has appeared. The size of the counter is determined by the parameter *m*. For a multiplicity *m*, we only need $n = \lceil \log_2 m \rceil$ bits.

For the counting of substrings in the reads, we increase the number at the positions where the *l*-mer appears. For each *l*-mer, the algorithm queries the Bloom filters. Once a positive query is found for some *k*, the value at those positions is increased, if all the minimal value at all those positions is larger than the *m*, the tuple is kept and saved as output.

4.4. Parallel Error Correction

4.4.1. Error Detection

We use the CUDA texture memory to hold the Bloom filter structure since we need a fast read-only memory to perform string querying. Reads are stored in global memory due to the large size of the read-array. All reads are concatenated to each other forming a one-dimensional character array, which is mapped into the CUDA global memory. The first stage of parallel error correction aims to distinguish the error-free and the erroneous reads. We use each thread to run on each character position of the read array to check the *l*-tuple starting at this position whether it is in the spectrum as shown in Figure 2. If affirmative, then the position is recorded as 1, and as 0 otherwise. The resulting binary one-dimensional array is then transferred back to CPU. A read is classified as error-free, if all the corresponding positions of the read in the resulting array are 1.

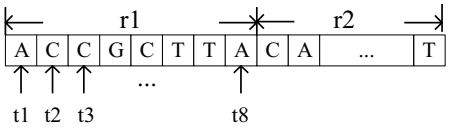


Fig. 2. CUDA kernel thread layout for error detection.

4.4.2. Voting

The error-free reads are filtered out after the first stage. The remaining erroneous reads are left to perform voting and error correction. As the voting procedure is relatively time consuming, this filtration of input read data is beneficial to the overall runtime. The CUDA implementation of the voting procedure described in our previous work [12] used one thread per read. However, we found that it is more efficient to use one thread per l-tuple instead as shown in Figure 4. Thus, there are up to $L - l + 1$ threads working on each read, compared to the one-read-per-thread approach used in [12]. Overall, the new approach allows a more fine-grained parallelization, which results in a more efficient utilization of shared memory (see the Results section for a speedup comparison).

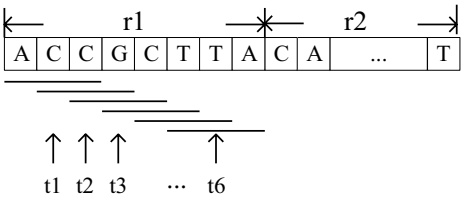


Fig. 3. CUDA kernel thread layout for voting

4.4.3. Fixing and Trimming

In this stage, we use each thread working on a whole read. The fixing algorithm is the same as in [12] with the difference that we only make the correction when the quality score at that position is below the threshold h.

We trim the read and keep the high quality parts if the first l-mers are all above the threshold. If not, the read is discarded.

5. Results

Firstly, we have compared the total number of false positives (FP) and false negatives (FN) of the spectrum produced by the quality-score guided trimming approach with the traditional approach. Table 2 shows the comparison for different multiplicity values for the Dataset A in Table 3. It can be seen that the sum of FP and FN of the produced spectrum generally smaller using the quality guided approach.

Table 2. Comparison of the sum of the false positives and false negatives in the spectrum constructed without quality trimming and with quality trimming.

Multiplicity *m*	Sum of FP and FN w/o Quality-trimming	Sum of FP and FN with Quality-trimming
2	943112	508850
3	381137	208808
4	196885	107788
5	118018	69060
6	82954	57034
7	70938	59846

Furthermore, we have evaluated the impact of our parallel quality-score guided error correction approach to the assembly of six real datasets downloaded from the NCBI SRA (shown in Table 3). The accession IDs are listed in column two. The size of the dataset ranges from 1.7M to 20M reads. The read length varies from 36 to 76 bps and corresponding coverage varies from 68 to 346.

Table 3. Summary of the datasets used for performance evaluation.

ID	SRA	Genome Length	Read Length	Coverage	Size
A	SRR006331	0.9 M	36	68	1.7 M
B	SRR016146	2.8 M	51	80	4.4 M
C	SRR016390	2.8 M	51	188	10.3 M
D	SRR011186	4.4 M	76	183	10.6 M
E	SRR016399	2.8 M	76	345	12.7 M
F	SRR026446	4.4 M	76	346	20

We have evaluated the performance of our new CUDA-EC program in terms of speed and assembly quality as follows:

Speed: we have measured the runtime of these datasets on an NVIDIA Tesla C1060 with CUDA version 2.3 and CUDA driver version 2.3. The card is connected to an Intel i7 2.67GHz CPU with 8GB RAM running Linux Fedora 10. The performance of our CUDA implementation is compared to the single-threaded C++ Euler-SR [2] implementation running on the same CPU in Table 4. Euler-SR is compiled with the –O3 option on an Intel i7 2.67GHz CPU with 8GB RAM. The runtime of CUDA-EC is divided into two parts: the sequential part running on the CPU (Counting) and the parallelized part running on the GPU (EC). The speedup of the sequential part is up 558 (Dataset E) and up to 84 for the parallelized part (Dataset A). The total speedup is up to 288. Table 5 further compares the speedup to the previous CUDA-EC version published in [12]. Compared to the previous version, the new CUDA-EC version contains (besides the integration of quality scores) counting Bloom filter data structure, multiple stage data filtration and different levels of thread parallelization, which results in a speedup of up to 1.88

for error correction in dataset A.

Assembly quality: Assembly performance is evaluated in terms of N50 values and the maximal contig length using the error-corrected and original datasets (A-F). Edena is chosen as the assembly tool which is based on an OLC (overlap layout consensus) approach. We have chosen an OLC-based assembler over a deBrujin graph based assembler like Euler-SR or Velvet because of two reasons:

1. The OLC approach can benefit more from error correction than the deBrujin graph based approaches (Velvet and Euler-SR).
2. The deBrujin graph based approaches (Velvet and Euler-SR) show scalability problem for larger datasets (e.g. E-F).

Table 6 shows the results produced by Edena for each dataset. In the table, the first line with suffix "O" stands for the original dataset as input. The adjacent line below with suffix "P" means the processed dataset with our error correction algorithm is used as input to Edena. The results are evaluated in terms of N50-values, maximal contig length (Max), and the runtime needed to complete the assembly. From Table 6, we can see that our error correction can help to improve the N50-values of up to 72% (Dataset C). Furthermore, the time needed to assemble the processed data is up to 10 times less than for the original data. This is due to fact that the produced overlap graph has significantly less branches after error correction.

Table 4. Runtime (in seconds) and speedup comparison between CUDA-EC and Euler-SR.

ID	Euler-SR Counting (Runtime)	Euler-SR EC (Runtime)	CUDA-EC Counting (Runtime)	CUDA-EC EC (Runtime)	Speedup Counting	Speedup EC	Speedup Total
A	61	671	53	8	1.15	83.88	12.000
B	338	1524	210	59	1.61	25.83	6.922
C	746	7016	210	160	3.55	43.85	20.978
D	11993	24735	438	505	27.38	48.98	38.948
E	351652	19129	630	655	558.18	29.20	288.546

Table 5. Runtime (in seconds) and speedup comparison between CUDA-EC (denoted as CUDA-EC2) and the old version of CUDA-EC (denoted as CUDA-EC1)

ID	CUDA-EC1 Counting (Runtime)	CUDA-EC1 EC (Runtime)	CUDA-EC2 Counting (Runtime)	CUDA-EC2 EC (Runtime)	Speedup Counting	Speedup EC	Speedup Total
A	55	15	53	8	1.03	1.88	1.15
B	220	106	210	59	1.05	1.80	1.21
C	393	242	210	160	1.87	1.51	1.72
D	701	770	438	505	1.60	1.52	1.56
E	1012	1077	630	655	1.61	1.64	1.63

Table 6. Comparsion of assembly results with Edena for the datasets A-F using the original and error corrected read datasets as inputs. All tests

were carried in strict assembly mode. The reported results are for the optimal parameters for each dataset.

Dataset ID		N50	Max	Min Overlap	Cutoff	Runtime (sec.)	N50	Max	Runtime speedup
A	AO	2389	11920	20	25	912			
	AP	2981	12777	20	27	90	1.248	1.072	10.133
B	BO	30226	104797	31	31	503			
	BP	32555	122771	31	31	335	1.077	1.172	1.501
C	CO	16375	61474	31	31	937			
	CP	28240	96476	31	31	691	1.725	1.569	1.356
D	DO	8591	43292	44	44	2645			
	DP	8620	43328	40	45	1730	1.003	1.001	1.529
E	EO	34063	93154	60	60	1518			
	EP	37632	137453	60	60	1018	1.105	1.476	1.491
F	FO	2404	16169	40	40	10785			
	FP	2928	16191	40	43	6240	1.218	1.001	1.728

6. Conclusion and Future Work

The emergence of new HTSR sequencing technologies establishes the need for new tools and algorithms that can process massive amounts of short reads in reasonable time. In this paper, we have addressed this challenge by writing scalable CUDA error correction software, which is an important but time-consuming pre-processing step for many de-novo assembly tools. In order to derive an efficient CUDA implementation we have used a space-efficient counting Bloom filter for hashing, multi-stage data filtration and different levels of thread parallelization. Our performance evaluation shows speedups between one and two orders of magnitude for various datasets.

A weakness of the SAP-based error correction method is that the l-mer spectrum of the reference genome $T(G)$ is only approximated by $T(R,m)$; i.e. the set of all l-mers with multiplicity $\geq m$ in the input read data set. We have addressed this weakness by incorporating base-call quality-scores to the spectrum construction in order to improve this approximation. The effectiveness of this incorporation has resulted in clearly improved N50-values for several real datasets from the NCBI SRA.

The speedup of the current CUDA implementation is often reduced by the sequential pre-computation of the Bloom filter on the CPU. Therefore, another part of our future work is to investigate more efficient methods for the sequential pre-processing stage. Furthermore, it would be interesting to compare the SAP-based error correction approach to other approaches, such as the recently introduced SHREC method [11], which uses a suffix tree of all input reads to identify and correct sequencing errors.

References

1. Butler J, et al. ALLPATHS: de novo assembly of whole-genome shotgun microreads. Genome Research 18(5), 810-820. 2008
2. Chaisson, M.J., Pevzner, P.A.. A short read fragment assembly of bacterial genomes. Genome Research 18(2), 324-330, 2008.
3. Chaisson, M.J., Brinza, D., and Pevzner, P.A.. De novo Fragment Assembly with Short Mate-Paired Reads: Does the Read Length Matter? Genome Research 19(2), 336-346, 2009.
4. Chaisson, M.J., Tang, H., Pevzner, P.A.. Fragment assembly with short reads. Bioinformatics, 20(13) 2067-2074, 2004.
5. Dohm, J.C., Lottaz, C., Borodina, T., and Himmelbauer, H. 2008. Substantial biases in ultra-short read data sets from high-throughput DNA sequencing. Nucleic Acid Research 36(16), e105.
6. Hernandez, D., et al.. De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. Genome Research 18(5), 802-809, 2008.

7. L. Fan, P. Cao, J. Almeida, and A.Z. Broder, Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol, IEEE/ACM Transaction on Network. Vol. 8, No. 3, June 2000.
8. Mardis, E.R. The impact of next generation sequencing on genetics. Trends in Genetics 24(3), 133-141, 2008.
9. Pop, M., and Salzberg, S.L. 2Bioinformatics challenges of new sequencing technology. Trends in Genetics 24(3), 142-149, 2008.
10. Schmidt, B., Sinha, R., Beresford-Smith, B., Puglisi, S., A Fast Hybrid Short Read Fragment Assembly Algorithm. Bioinformatics 25(17) 2279-2280, 2009
11. Schröder, J., Schröder, H., Puglisi, S., Sinha, R., and Schmidt, B. 2009. SHREC: A short-read error correction method. Bioinformatics 25(17) 2157-2163.
12. Shi, H., Schmidt, B., Liu, W., Müller-Wittig, W.: Accelerating error correction in high-throughput short-read DNA sequencing data with CUDA. IPDPS 1-8, 2009.
13. Shi, H., Schmidt, B., Liu, W., Mueller-Wittig, W.: A Parallel Algorithm for Error Correction in High-Throughput Short-Read Data on CUDA-enabled Graphics Hardware, Journal of Computational Biology, to appear.
14. Simpson, J.T., Wong, K., Jackman, S.D., ABySS: A parallel assembler for short read sequence data. Genome Research 19, 1117-1123, 2009
15. Smith, A.D., Xuan, Z., Zhang, M.: Using quality scores and longer reads improves accuracy of Solexa read mapping, BMC Bioinformatics 2008, 9:128.
16. Strausberg, R.L., Levy, S., Rogers, Y.H. Emerging DNA sequencing technologies for human genomic medicine. Drug Discovery Today 13(13/14), 569-577, 2008.
17. Zerbino, D.R., and Birney, E. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. Genome Research 18(5), 821-829, 2008.