Accelerating Error Correction in High-Throughput Short-Read DNA Sequencing Data with CUDA

Haixiang Shi, Bertil Schmidt, Weiguo Liu, and Wolfgang Müller-Wittig

School of Computer Engineering, Nanyang Technological University, Singapore 639798, {hxshi,asbschmidt,liuweiguo,askwmwittig}@ntu.edu.sg

Abstract

Emerging DNA sequencing technologies open up exciting new opportunities for genome sequencing by generating read data with a massive throughput. However, produced reads are significantly shorter and more error-prone compared to the traditional Sanger shotgun sequencing method. This poses challenges for de-novo DNA fragment assembly algorithms in terms of both accuracy (to deal with short, error-prone reads) and scalability (to deal with very large input data sets). In this paper we present a scalable parallel algorithm for correcting sequencing errors in high-throughput short-read data. It is based on spectral alignment and uses the CUDA programming model. Our computational experiments on a GTX 280 GPU show runtime savings between 10 and 19 times (for different error-rates using simulated datasets as well as real Solexa/Illumina datasets).

1. Introduction

Recently, a number of second-generation DNA sequencing technologies has been introduced. Compared to the traditional Sanger shotgun technique these new technologies can generate a massive amount of read data at lower cost [11, 14, 18]. Examples of already available second-generation technologies are sequencers from 454 Life Sciences/Roche, Solexa/Illumina, and Applied Biosiciences/SOLiD. However, the length of produced reads is significantly shorter compared to the Sanger method. For example, the Illumina Genome Analyzer can generate 1.5 billion base-pairs (bps) of sequence data in a single-end 60-hours run with a read length of 36.

Established methods and tools for DNA fragment assembly have been designed and optimized for Sanger shotgun sequencing (i.e. read lengths of around 500bps and 6-to-10-fold coverage) and generally do not scale well for high-coverage short read data. Thus, there is a need for scalable fragment assembly tools that can deal with high-throughput short-read (*HTSR*) data. Consequently, several such tools have been introduced recently. SSAKE [20] and SHARCGS [5] are based on a simple *k*-mer extension approach. However, this approach is inaccurate for assembling repeat regions. In order to resolve repeats the graph-based approach to assembly has been introduced in [13], which was then implemented in the Euler assembly package. Euler-SR [3] is a recent extension of Euler to assemble HTSR data. Graph-based assembly approaches are highly sensitive to sequencing errors and therefore require error correction as a necessary pre-processing step. Unfortunately, this step is highly time-consuming for large amounts of data used in second-generation sequencing projects.

In this paper we present a parallel algorithm for correcting sequencing errors in HTSR data using the CUDA programming model. It is based on the spectral alignment problem and uses a Bloom filter data structure in order to take advantage of the CUDA memory hierarchy. We show how this leads to significant runtime savings between 10 and 19 times on low cost CUDA-compatible graphics hardware.

The rest of this paper is organized as follows. In Section 2, we introduce the spectral alignment approach to sequential error correction. Features of the CUDA programming model are highlighted in Section 3. Section 4 presents the parallel error correction algorithm. The performance of our CUDA implementation is evaluated in terms of runtime and correction accuracy in Section 5. Finally, Section 6 concludes the paper.

2. Error Correction using Spectral Alignment

Error correction is a preprocessing step for many DNA fragment assembly tools. A common and accurate approach is to compute multiple alignments of

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

shotgun reads and then detect and correct errors in certain columns of these alignments [1, 19]. Unfortunately, calculating multiple alignments is too time-consuming for HTSR data. The approach used in this paper is based on spectral alignment [4, 13], which uses the following definitions and concepts.

Given are a set of k reads $R = \{r_1,\ldots,r_k\}$ with $|r_i|=L$ and $r_i \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}^L$ for all $1 \le i \le k$ and two integers: multiplicity m ($m>1$) and length l ($l < L$).

Definition 1 (Solid and weak): An l-tuple (i.e. a DNA string of length l) is called *solid* with respect to R and m if it is a substring of at least m reads in R and *weak* otherwise.

Definition 2 (Spectrum): The spectrum of R with respect to m and l, denoted as $T_{m.l}(R)$, is the set of all solid l-tuples with respect to R and m.

Definition 3 (*T*-string): A DNA string s is called a $T_{m.l}(R)$-*string* if every l-tuple in s is an element of $T_{m.l}(R)$.

The spectral alignment problem (*SAP*) to error correction can now be defined as follows:

Definition 4 (SAP): Given a DNA string s and spectrum $T_{m.l}(R)$. Find a $T_{m.l}(R)$-string s^* in the set of all $T_{m.l}(R)$-strings that minimizes the distance function $d(s,s^*)$.

Depending on the error model of the utilized sequencing technology the distance function $d(\cdot,\cdot)$ can either be *edit distance* (suitable for 454 Life Sciences/Roche) or *hamming distance* (suitable for Solexa/Illumina). Euler-SR contains two corresponding heuristics to approximate the SAP. One is based on edit distance and uses the dynamic programming approach described in [4]. The other is based on hamming distance and uses the iterative approach presented in [13]. We focus on Solexa/Illumina technology and therefore the latter approach is chosen in this paper.

The main idea of the iterative approach is as follows. Let us assume there is an error (mutation) at position j in read r_i. Then, this mutation creates $\min\{l, j, L-j\}$ erroneous l-tuples that point to the same sequencing error. The iterative SAP heuristic associates the correction of r_i at position j to the transformation of $\min\{l, j, L-j\}$ weak l-tuples into solid l-tuples. The heuristic searches for such corrections using a voting procedure, which is described in detail in Section 4. It should also be mentioned that it is not always the case that an error leads exactly to $\min\{l, j, L-j\}$ weak-to-solid transformations. This can be caused by three factors:

1. Several errors are located within distance l.
2. Correct l-tuples are considered weak.
3. Erroneous l-tuples are considered solid.

The first case can be included in the heuristic by considering up to Δ ($\Delta \ge 1$) corrections within a weak l-tuple. However, this also increases the time complexity of the heuristic. The second and third case can be minimized by an optimal choice of the parameters m and l. This choice depends on a, the average number of reads covering an l-tuple of the sequenced genome ($a = N\cdot(L-l)/G$, where G is the genome length)).

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 [12]. CUDA programs can be executed on GPUs with NVIDIA's Tesla unified computing architecture [8]. Examples of CUDA-enabled GPUs include the GeForce 8, 9, and 200 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*) $\in \{0,\ldots,dimBlock-1\} \times \{0,\ldots,dimGrid-1\}$. 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. Besides the PBSM, there are four other types of memory: per-thread private local memory, global memory for data shared by all threads, texture memory and constant memory. Texture memory and constant memory can be regarded as fast read-only caches.

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 processors, 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.

Previous work on using CUDA for Bioinformatics focused on sequence alignment [10, 16] and molecular dynamics simulations [9]. MUMerGPU [16] processes HTSR data for re-sequencing applications; i.e. produced reads are aligned to a known reference genome sequence using a suffix tree. The approach presented in this paper is targeted towards de-novo sequencing where the reference genome sequence is unknown.

4. Parallel Error Correction with CUDA

The computational domain of the SAP approach to error correction consists of a set of reads R that need to independently access the spectrum $T_{m,l}(R)$. Hence, we use a kernel to represent the sequential processing necessary for the correction of an individual read r_i. The kernel is then invoked using a thread for each $r_i \in R$.

Algorithm 1: Single-mutation voting algorithm used in the CUDA kernel.

Input: Read $r_i[0\ldots L-1]$ and spectrum $T_{m,l}(R)$.
Output: Voting matrix $V_i[][]$ of size $L\times 4$

```
for $j := 0\ldots L-1$
    for $c \in$ {A, C, G, T} $V_i[j][c] := 0$;
for $j := 0\ldots L-(l+1)$
    if ($r_i[j\ldots j+l-1] \notin T_{m,l}(R)$)
        for $p := 0\ldots l-1$
            for $c \in$ {A, C, G, T}\{$r_i[j+p]$}
                if ($r_i[j\ldots j+p-1]\cdot c\cdot r_i[j+p+1\ldots j+l-1] \in T_{m,l}(R)$) $V_i[j+p][c]$++;
                // · denotes string concatenation
```

Our CUDA kernel consists of two phases. The first phase is the single-mutation voting algorithm outlined in Algorithm 1. It identifies all l-tuples of the given read that are not in the spectrum set. Each single-point mutation of these l-tuples is then tested for membership in the spectrum. If successful a corresponding counter in the voting matrix is incremented. Errors can be fixed based on high values in the voting matrix. In certain cases; e.g. when two errors are close two each other, the single-mutation voting algorithm cannot correct the errors. However, it is still possible to identify the read as erroneous and to trim it at certain positions or discard it. Algorithm 2 outlines the fixing/trimming/discarding procedure used as the second phase of our CUDA kernel.

Algorithm 2: Single-mutation fixing/trimming/ discarding procedure used in the CUDA kernel.

Input: Read $r_i[0\ldots L-1]$ and Voting matrix $V_i[][]$ of size $(L-(l+1))\times 4$.
Output: Fixed, trimmed, discarded or unchanged read

```
$maxV := \max\{V_i[j][c] \mid 0\le j\le L-(l+1);\ c\in \{A,C,G,T\}\}$;
$(maxj,maxc) := \operatorname{argmax}_{0\le j\le L-(l+1)\,;\ c\in\{A,\,C,\,G,\,T\}}\{V_i[j][c]\}$;
if ($maxV = 0$) return $r_i$;   // input read is error-free
$r^c_i := r_i[0\ldots j+maxj-1]\cdot maxc\cdot r_i[maxj+1\ldots L-1]$;
            // · denotes string concatenation
corrected_flag := true; trimmed_flag:=false;
for $j := 0\ldots L-(l+1)$
    if ($r^c_i[j\ldots j+l-1] \notin T_{m,l}(R)$) corrected_flag := false;
    else trimmed_flag:=true:
if (corrected_flag) return $r^c_i$; // return *corrected* read
else if (trimmed_flag)  return $r^c_i[k_1\ldots k_2]$;
    // return *trimmed* read, where $r^c_i[k_1\ldots k_2]$ is the
    //longest substring of $r^c_i$ in which all $l$-tuples
    //belong to $T_{m,l}(R)$
else return ε;  // return *discarded* read as empty string
```

The time complexity of the kernel is dominated by the first phase. The most important operation that determines the runtime of the voting algorithm is the spectrum membership test. The overall amount of membership queries by a single thread is $(L-l)\cdot(1+p\cdot 3\cdot l)$, where p is the number of l-tuples of the read that do not belong to the spectrum. *Hashing* can perform membership tests to a hashing table in constant time. An efficient way to store a frequently accessed hash table in CUDA is to use the read-only texture memory. However, the hash table can become prohibitively large for the amount of l-tuples generated by in HTSR sequencing projects. Therefore, we have decided to use probabilistic hashing based on the space-efficient bloom filter data structure to store the l-tuples of the spectrum.

A Bloom filter represents a set of given keys in a bit-vector. Insertion and querying of keys are supported using several independent hash functions. Bloom filters gain their space-efficiency by allowing a false positive answer to membership queries. Space savings often outweigh this drawback in applications where a small false positive rate can be tolerated, particularly when space resources are at a premium. Both criteria are met for the CUDA error correction algorithm. In the following, we briefly review the definition, programming, querying and false positive probability of Bloom filters.

Definition: A *Bloom filter* is defined by a bit-vector of length m, denoted as $BF[1..m]$. A family of k hash functions h_i: $K \rightarrow A$, $1 \leq i \leq k$, is associated to the Bloom filter, where K is the key space and $A = \{1,\ldots,m\}$ is the address space.

Programming: For a given set I of n keys, $I = \{x_1,\ldots,x_n\}$, $I \subseteq K$, the Bloom filter is *programmed* as follows. The bit vector is initialized with zeros; i.e. $BF[i] := 0$ for all $1 \leq i \leq m$. For each key $x_j \in I$, the k hash values $h_i(x_j)$, $1 \leq i \leq k$, are computed. Subsequently, the bit-vector bits addressed by these k values are set to one; i.e. $BF[h_i(x_j)] := 1$ for all $1 \leq i \leq k$. Note that, if one of these values addressed a bit which is already set to one, that bit is not changed.

Querying: For a given key $x \in K$, the Bloom filter is *queried* for membership in I in a similar way. The k hash values $h_i(x)$, $1 \leq i \leq k$, are computed. If at least one of the k bits $BF[h_i(x)]$, $1 \leq i \leq k$, is zero, then $x \notin I$. Otherwise, x is said to be a member of I with a certain probability. If all k bits are found to be one but $x \notin I$, x is said to be a false positive.

False Positive Probability: The presence of false positives arises from the fact that the k bits in the bit-vector can be set to one by any of the n keys. Note that a Bloom filter can produce false positive but never false negative answers to queries. The false positive probability (denoted as *FPP*) of a Bloom filter is given by [2].

$$FPP = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{\frac{kn}{m}}\right)^k$$

Obviously, *FPP* decreases as the bit-vector size m increases, and increases as the number of inserted keys n increases. In our Bloom filter implementation using one-dimensional linear texture memory we have chosen $k = 8$ and $m = 64 \times n$, which leads to FPP = 3.63e-08.

The memory requirement for storing the voting matrix $V_i[][]$ in Algorithm 1 can be reduced to $l \times 4$ Bytes by using cyclic indexing. The amount of memory required for storing the voting matrix and the read is therefore $4 \times l + L$ bytes per thread. This leads to a requirement of $4 \times 20 + 35 = 115$ bytes for typical parameters used for Solexa/Illumina HTSR data (l=20, L=35). Therefore, PBSM could be used to store this data. However, this would limit the number of threads per block to 128. Furthermore, it would negatively affect the maximum number of thread blocks per multiprocessor (*MTBPM*). In general, there is a trade-off between PBSM usage and MTBPM: increased PBSM usage decreases MTBPM. Lower MTBPM results in a lower warp occupancy and efficiency. The CUDA occupancy calculator tool recommends a usage of less than 4KB PBSM for our implementation. Therefore, we have decided not to store the voting matrix in PBSM but in local memory.

Furthermore, if the number of reads exceeds the total number of threads used in the kernel our implementation allows processing several reads per thread. Overall, the steps of our CUDA implementation are as follows.

1. *Pre-computation on the CPU*: Program the Bloom filter bit-vector by hashing each l-tuple in the spectrum.
2. *Data transfer from CPU to GPU*: Transfer bloom filter bit-vector and read data to the allocated texture and global memory on the GPU.
3. *Execute CUDA kernel.*
4. *Data transfer from GPU to CPU.* Transfer the set of corrected/trimmed reads to the CPU and save them to a file.

5. Performance Evaluation

We have evaluated the performance of our CUDA implementation using several simulated Solexa/Illumina-style datasets as well as a real Solexa/Illumina dataset. The simulated datasets have been produced by generating random reads with a given error-rate from a reference genome sequence. In order to test scalability, we have selected yeast chromosomes (S.cer5, S.cer7) and bacterial genomes (H.inf, E.col) as reference sequences of various lengths (ranging from 0.58Mbp to 4.71Mbp). Three datasets have been created for each reference genome sequence using per-base error-rates of 1%, 2%, and 3%. Each dataset uses a constant read length of L=35 and has a coverage of C=70. Thus, the size of simulated input datasets varies from 1.1M to 9.4M reads. The real dataset consists of 3.9M unambiguous reads of length 35 each and was downloaded from http://www.genomic.ch/edena.php. It has been obtained experimentally by Hernandez et al. [7] using the Illumina Genome Analyzer for sequencing the Staphylococcus Aureus strain MW2 (*H.Aci*). We have estimated the error rate of this dataset as 1% by aligning each read to the reference genome using RMAP [17]. The 12 datasets used for our performance evaluation are summarized in Table 1.

Table 1: Datasets used for performance evaluation.

ID	Reference genome (GenBank ID)	Genome Length	Error-rate	Coverage	Read length	Number of reads
A1	S.cer 5 (NC_001137)	0.58M	1%	70	35	1.1M
A2			2%			
A3			3%			
B1	S.cer 7 (NC_001139)	1.1M	1%			2.2M
B2			2%			
B3			3%			
C1	H.inf (NC_007146)	1.9M	1%			3.8M
C2			2%			
C3			3%			
D1	E.col (NC_000913)	4.7M	1%			9.4M
D2			2%			
E	S.aureus(NC_003923.1)	2.8M	1%	43		3.5M

Table 2: Runtime comparison (in seconds) between parallel CUDA and sequential Euler-SR error correction.

Dataset	A1	A2	A3	B1	B2	B3	C1	C2	C3	D1	D2	E
Euler	118	187	257	239	389	535	455	750	1026	1253	1979	510
CUDA	11	14	15	20	25	28	44	56	65	108	139	40
Speedup	10.7	13.6	17.1	11.9	15.6	19.1	10.3	13.4	15.8	11.6	14.2	12.7

We have measured the runtime of these 12 datasets on an NVIDIA GeForce GTX 280 with CUDA version 2.0 and CUDA driver version 177.73. The GTX 280 comprises 30 SMs, each containing 8 SPs running at 1.3GHz. The total size of the device memory is 1GB. The card is connected to an AMD Opteron dual-core 2.2GHz CPU with 2GB RAM running Linux Fedora 8 by the PCIe 2.0 bus. The performance of our CUDA implementation is compared to a single-threaded C++ code running on the same CPU. It is taken from the error correction for Solexa/Illumina data in Euler-SR (using the version from Aug., 14th, 2008), which is available at http://euler-assembler.ucsd.edu. The code is a serial implementation of the single-mutation error correction algorithm presented in Section 4. It stores the spectrum in a sorted vector and then calls the efficient STL function std::lower_bound() for membership queries. CUDA does generally not support STL functions, which was one of the reasons to use a Bloom filter data structure instead. Table 2 shows the runtime comparison between the sequential and the CUDA implementation for all datasets using the default parameters l=20 and m=6. The CUDA timings include pre-computation time on the CPU, CUDA kernel time, and CPU-GPU data transfer time. CUDA kernels are executed using 256 thread-blocks and 256 threads per block. The Euler-SR code is compiled using GNU GCC 4.1.2 with the Full Optimization (-O3) enabled.

The speedup values in Table 2 indicate the following trends:

- The speedup increases for higher error rates.

The trend can be explained as follows. The single-mutation voting algorithm contains the data-dependent conditional branch "if ($r_i[j\ldots j+l-1] \notin T_{m,l}(R)$)" (see Algorithm 1) that is executed for each l-tuple in the given read. This leads to inefficiencies in the CUDA implementation due to the SIMT execution model (see Section 3). Threads, for which this statement is true, execute another $3\cdot l$ membership tests. Threads, for which this statement is false, need to wait for these threads within the same warp to finish these tests. The number of error-free reads is decreasing for higher error rates, e.g. for L=35, 70%, 49%, and 34% of reads are error-free for error-rates of 1%, 2%, and 3%, respectively. Thus, the number of waiting threads per warp is decreasing for higher error-rates, which in-turn improves the efficiency of the CUDA implementation.

Table 3: Definitions of TP, FP, FN, and TN for the read identification classification test.

		Read condition	
		erroneous	*error-free*
Algorithm outcome	*Fixed, trimmed, or discarded*	TP	FP
	Unchanged	FN	TN

Table 4: Performance of the read identification classification test measured in sensitivity and specificity for each dataset. The number of reads in the four sets TP, FP, TN, and FN is also shown.

	TP	*FP*	*FN*	*TN*	*Sensitivity*	*Specificity*
A1	339,716	12	696	813,320	99.800%	99.999%
A2	578,346	24	2,337	573,056	99.600%	99.996%
A3	746,559	10	4,179	403,052	99.443%	99.998%
B1	643,112	25	1,343	1,537,422	99.791%	99.998%
B2	1,093,320	34	4,449	1,084,129	99.594%	99.997%
B3	1,411,258	32	8,200	762,526	99.422%	99.996%
C1	1,128,563	26	2,360	2,698,039	99.791%	99.999%
C2	1,920,114	24	7,868	1,901,030	99.591%	99.999%
C3	2,476,651	36	14,612	1,337,898	99.413%	99.997%
D1	2,734,051	122	5,963	6,539,234	99.782%	99.998%
D2	4,651,747	196	19,460	4,608,147	99.583%	99.996%
E	951,063	90	3,814	2,474,218	99.600%	99.996%

Table 5: Reads in TP that have not been discarded (i.e. they are either corrected or trimmed). The percentage of reads in the corrected/trimmed dataset that are accurately corrected/trimmed is also given.

Dataset	A1	A2	A3	B1	B2	B3	C1	C2	C3	D1	D2	E
Corrected/ Trimmed (in %)	95.4	92.2	88.8	94.9	92.1	88.8	95.0	92.2	88.9	94.9	92.2	79.9
Accuracy (in %)	99.5	99.1	99.1	99.5	99.1	98.7	99.5	99.1	98.7	99.5	99.1	99.4

We have further analyzed the accuracy of our CUDA implementation in terms of

- *Identification*; i.e. identifying reads as erroneous or error-free.
- *Correction*; i.e. correcting reads, which have been identified as erroneous.

The identification of erroneous reads can be defined as a binary classification test. The corresponding definitions of true positive (*TP*), false positive (*FP*), true negative (*TN*), and false negative (*FN*) are given in Table 3. Sensitivity and specificity measures are then defined as:

$$\text{sensitivity} = \frac{TP}{TP + FN}; \quad \text{specificty} = \frac{TN}{TN + FP}.$$

Table 4 shows the specificity and sensitivity measures for the 12 tested datasets. It can be seen that that the algorithm identifies erroneous reads with very high accuracy.

Our implementation outputs four datasets: unchanged reads, corrected reads, trimmed reads, and discarded reads. The union of unchanged, corrected and trimmed reads is usually taken as an input dataset to a subsequent de-novo DNA fragment assembly algorithm. Therefore, we have further analyzed the reads that have been classified as *TP*. The amount of corrected/trimmed reads relative to the number of discarded reads is shown Table 5. Furthermore, Table 5 shows the accuracy of the actual correction/trimming operation; i.e. if correction/trimming is done at the correct read positions.

Table 5 shows that the percentage of corrected/trimmed reads decreases compared to the discarded reads for higher error-rates. This can be explained by the larger number of erroneous reads with more than one error for higher error-rates. These reads cannot be corrected with the single-mutation voting algorithm; and will therefore be either trimmed or discarded. A further observation is that the percentage of corrected/trimed reads is lower for the real dataset (i.e. dataset “E”) than for the simulated datasets with a similar error-rate (1%). The likely reason for this is that errors in real reads are not as evenly distributed as in our simulated reads. A recent paper [6] has reported that error rates in Solexa/Illumina reads range from 0.3% at the beginning of a reads to 3.8% at the end of reads.

6. Conclusion and Future Work

Emerging HTSR sequencing technologies presents a major bioinformatics challenge. In particular, Bioinformatics tools that can process massive amounts of short reads are required. A promising approach to address this challenge is to write scalable software for modern many-core architectures such as GPUs. In this paper we have presented the design, implementation and testing of a parallel short read error correction method using the CUDA programming model. In order to derive an efficient CUDA implementation we have used a space-efficient Bloom filter for hashing and taken advantage of the CUDA memory hierarchy. Our performance evaluation on a commodity GPU shows speedups of up to 19 times for datasets of various sizes and error rates.

HTSR technologies such as Solexa/Illumina are capable of generating much larger datasets than the ones used in this paper. Therefore, our future work includes testing the scalability of the presented algorithm on even bigger datasets and for different read/tuple-lengths. The produced software will also be made publicly software available. Furthermore, we are planning to improve the correction performance of real datasets by running a double-mutation or triple-mutation voting algorithm on the trimmed and discarded read datasets

References

[1] Batzoglou, S., et al.: “ARACHNE: A whole-genome shotgun assembler”, *Genome Research*, **12**, 177-189, 2002.

[2] Bloom, B.: “Space/Time Trade-Offs in Hash Coding with Allowable Errors,” *Commun. ACM*, 13 (7) , 422-426, 1970.

[3] Chaisson, M.J., Pevzner, P.A.: “A short read fragment assembly of bacterial genomes”. *Genome Research*, 18, 324-330, 2008.

[4] Chaisson, M.J., Tang, H., Pevzner, P.A.: “Fragment assembly with short reads”. *Bioinformatics*, **20**, 2067-2074, 2004.

[5] Dohm, J.C., Lottaz, C., Borodina, T., Himmelbauer, H.: “SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic assembly”, *Genome Research*, 17, 1697-1706, 2007

[6] Dohm, J.C., Lottaz, C., Borodina, T., Himmelbauer, H.: “Substantial biases in ultra-short read data sets from high-throughput DNA sequencing”, *Nucleic Acid Research*, 36 (16):e105, 2008.

[7] Hernandez, D., et al.: “De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer”. *Genome Research.* 18:802-809, 2008.

[8] Lindholm, E., Nickolls, J., Oberman, S., Montrym: “NVIDIA Tesla: A unified graphics and computing architecture”. IEEE Micro, 28 (2), 40-52, 2008.

[9] Liu, W., Schmidt, B., Voss, G., Mueller-Wittig W.: "Accelerating Molecular Dynamics simulations using Graphics Processing Units with CUDA", *Computer Physics Commun-ications*, 179 (9), 634-641, 2008

[10] Manavski, S.A., Valle, G.: "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment", *BMC Bioinformatics* 9(S2) 2008.

[11] Mardis, E.R.: “The impact of next generation sequencing on genetics”, *Trends in Genetics*, 24 (3), 133-141, 2008.

[12] Nickolls, J., Buck, I., Garland, M., Skadron, K.: “Scalable parallel programming with CUDA”. *ACM Queue*, 6 (2), 39-55, 2008.

[13] Pevzner, P.A., Tang, H., Waterman M.S.: “An Eulerian path approach to DNA fragment assembly”. *Proc. Natl. Acad, Sci.*, 98, 9748-9753, 2001

[14] Pop, M., Salzberg, S.L.: “Bioinformatics challe-nges of new sequencing technology” *Trends in Genetics*, 24 (3), 142-149, 2008.

[15] Sanger, Nicklen, S. and Coulson, A.R.: “DNA sequencing with chain-terminating inhibitors”. *Proc. Natl. Acad. Sci.*, 74, 5463-5467, 1977.

[16] Schatz, M.C., Trapnell, C., Delcher, A.L. Varshney, A.: “High-throughput sequence align-ment using graphics processing units” *BMC Bioinformatics* 8 (474) 2007.

[17] Smith, A.D., Xuan, Z., Zhang, M.Q.: Using quailty scores and longer reads improves accuracy of Solexa read mapping. *BMC Bioinformatics*, 9, 128, 2008

[18] 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

[19] Tammi, M.T., Arner, E., Kindlund, E., Andersson B.: "Correcting errors for shotgun sequencing" *Nucleic Acid Research*, 31, 4663-4672, 2003.

[20] Warren, R.L., Sutton, G.G., Jones, S.J., Holt, R.A.: "Assembling millions of short DNA sequences using SSAKE". *Bioinformatics*, 23 (4), 500-501, 2007.

Acknowledgments

The work was supported by the A*Star BMRC research grant No. 04/1/22/19/375, an AcRF Tier-1 grant and an NVIDIA Professor partner grant.