JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 17, Number 4, 2010
© Mary Ann Liebert, Inc.
Pp. 603–615
DOI: 10.1089/cmb.2009.0062

A Parallel Algorithm for Error Correction in High-Throughput Short-Read Data on CUDA-Enabled Graphics Hardware

HAIXIANG SHI, BERTIL SCHMIDT, WEIGUO LIU, and WOLFGANG MÜLLER-WITTIG

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 article, we present a scalable parallel algorithm for correcting sequencing errors in high-throughput short-read data so that error-free reads can be available before DNA fragment assembly, which is of high importance to many graph-based short-read assembly tools. The algorithm is based on spectral alignment and uses the Compute Unified Device Architecture (CUDA) programming model. To gain efficiency we are taking advantage of the CUDA texture memory using a space-efficient Bloom filter data structure for spectrum membership queries. We have tested the runtime and accuracy of our algorithm using real and simulated Illumina data for different read lengths, error rates, input sizes, and algorithmic parameters. Using a CUDA-enabled mass-produced GPU (available for less than US$400 at any local computer outlet), this results in speedups of 12–84 times for the parallelized error correction, and speedups of 3–63 times for both sequential preprocessing and parallelized error correction compared to the publicly available Euler-SR program. Our implementation is freely available for download from http://cuda-ec.sourceforge.net.

Key words: algorithms, dynamic programming, sequence analysis, suffix trees.

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 (Mardis, 2008; Pop and Salzberg, 2008; Strausberg et al., 2008). 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

School of Computer Engineering, Nanyang Technological University, Singapore.

compared to the Sanger method. For example, the Illumina Genome Analyzer can generate up to 100 million (unpaired or paired) reads in a single run (lasting 2–8 days depending on the performance parameters used) with a read length of 35–75 and a per-base error rate of 1–2%.

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 (Warren et al., 2007) and SHARCGS (Dohm et al., 2007) 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 Pevzner et al. (2001), which was then implemented in the Euler assembly package. Euler-SR (Chaisson et al., 2008, 2009) is a recent extension of Euler to assemble HTSR data. Other recently published graph-based HTSR *de novo* assembly tools include ALLPATHS (Butler et al., 2008), Taipau (Schmidt et al., 2009), Velvet (Zerbino and Birney, 2008), and Edena (Hernandez et al., 2008). Graph-based assembly approaches generally use an exact overlap of length *k* to generate a link in the graph and are therefore highly sensitive to sequencing errors. Hence, correcting as many base-pair errors as possible in the input read data before graph construction can significantly improve assembly quality. Unfortunately, this pre-processing step is highly time-consuming for large amounts of data used in second-generation sequencing projects.

In this article, we present a parallel error correction algorithm in HTSR data using the CUDA parallel programming model. The error correction algorithm 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 further introduce a sequential preprocessing method for spectrum counting, which also uses a set of Bloom filters to calculate substrings from reads with multiplicity above a given threshold. The performance of our algorithm is rigorously tested in terms of correction sensitivity, specificity, and accuracy as well as runtime using real and simulated Illumina read datasets. Speedups are compared to the sequential error correction implementation of Euler-SR (Chaisson et al., 2008, 2009), which is also based on spectral alignment (so is the sequential error correction implementation in ALLPATHS (Butler et al., 2008)). The achieved runtime saving on a GeForce 280 GTX are up to 84 times for the parallelized error correction and up to 63 times for error correction, including sequential spectrum counting.

The rest of this article is organized as follows. In Section 2, we introduce the spectral alignment approach to error correction in short-read data. 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 terms of runtime and correction accuracy in Section 5. Finally, Section 6 concludes the article.

2. SPECTRAL ALIGNMENT APPROACH TO ERROR CORRECTION

Error correction is an important preprocessing step for many DNA fragment assembly tools. A common and accurate approach is to compute multiple alignments of shotgun reads and then detect and correct errors in certain columns of these alignments (Batzoglou et al., 2002; Tammi et al., 2003). Unfortunately, calculating multiple alignments is too time-consuming for HTSR data. The approach used in this article is based on spectral alignment (Pevzner et al., 2001; Chaisson et al., 2004), 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 $r_i \in \{A, C, G, T\}^L$ for all $1 \leq i \leq k$ 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*) for 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 Chaisson et al. (2004). The other is based on hamming distance and uses the iterative approach presented in Pevzner et al. (2001). We focus on Illumina technology, and therefore the latter approach is chosen in this article.

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 ($\Delta \geq 1$) corrections within a weak l-tuple. However, increasing Δ also increases the time complexity. Our implementation currently supports $\Delta = 1$ and $\Delta = 2$, which is usually sufficient in practices (assuming per-base error-rates of 1–3%). 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 ($a = N\cdot(L\text{-}l)/G$, where G is the genome length).

3. CUDA PROGRAMMING MODEL AND TESLA ARCHITECTURE

Compute Unified Device Architecture (CUDA) 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 ($threadIdx, blockIdx$) $\in \{0, \ldots, dimBlock - 1\} \times \{0, \ldots, dimGrid - 1\}$. 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. 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 (16 KB 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 64 KB) 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 (16 KB 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 16 KB (Fig. 1). 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 (Manavski and Valle, 2007; Schatz et al., 2007) and molecular dynamics simulations (Liu et al., 2008). MUMerGPU (Schatz et al., 2007) 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 article is targeted towards *de novo* sequencing where the reference genome sequence is unknown.

4. PARALLEL ERROR CORRECTION WITH CUDA

4.1. Bloom filter data structure

The most important operation that determines the runtime of the spectral alignment approach to error correction is the spectrum membership test, i.e., testing whether $s \in T_{m,l}(R)$ for a given l-tuple s. *Hashing* can

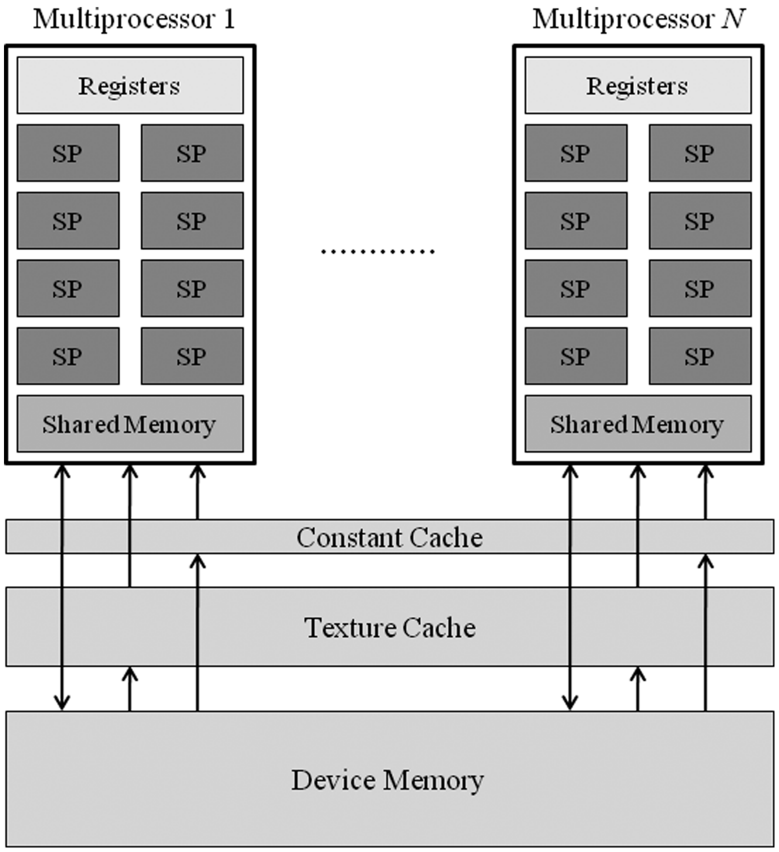

FIG. 1. Hardware model (Tesla) of CUDA-enabled GPUs.

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 represent the spectrum $T_{m,l}(R)$.

A Bloom filter represents a set of given keys in a bit-vector (Bloom, 1970). 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 particular, a false positive query might only lead to an unidentified sequencing error (i.e., false negative), and a small amount of false negatives is generally tolerated by an error correction algorithm. In the following, we briefly review the definition, programming, querying, and false positive probability of Bloom filters.

Definition 5 (Bloom filter): 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. K is the set of all l-tuples in this article.

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 (Fig. 2).

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 Bloom (1970).

The false positive probability (*FPP*) of a Bloom filter is given by Bloom (1970):

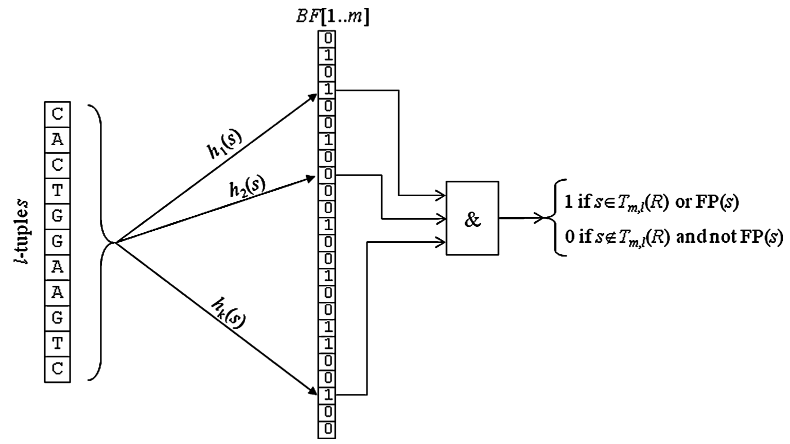

FIG. 2. Bloom filter data structure for querying the spectrum membership of the l-mer s (FP(s) is true if s is a false positive).

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

Obviously, *FPP* decreases as the bit-vector size m increases and increases as the number of inserted keys n increases. It can be shown that for a given m and n, the optimal number of hash functions k_{opt} is given by $k_{opt} = (m/n) \cdot \ln(2)$. The corresponding false positive probability is then $(1/2)^{kopt} \approx 0.6158^{(m/n)}$.

Hence, in the optimally configured Bloom filter, the false positive rate decreases exponentially with the size of the bit-vector. Furthermore, to maintain a fixed FPP, the bit-vector size needs to scale linearly with the inserted key set. 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.

4.2. Spectrum counting algorithm

Before correcting errors with the SAP approach, the spectrum $T_{m.l}(R)$ consisting of the set of all substrings of length l which have a multiplicity of at least m needs to be computed in a pre-processing step.

Algorithm 1. Spectrum pre-computation

```
Input: Read set $R = \{r_i[0 \ldots L-1]: i = 0, .., N-1\}$; tuple-length $l$; multiplicity $m$.
Output: Bloom filter $B(T_{m.l}(R)$.
for $k := 1 \ldots m$ do initialize($B_k$) // initialize $k^{\text{th}}$ Bloom filter with zeros
for $i := 0 \ldots N-1$ do
  for $j := 0 \ldots L-(l+1)$ do
    if (!query($r_i[j \ldots j+l-1]$, $B_m$) // check if $l$-tuple is already in $B_m$
      for $k = m-1 \ldots 1$ do
        if (query($r_i[j \ldots j+l-1]$, $B_k$) // check if $l$-tuple has already occurred $k$ times before
          program($r_i[j \ldots j+l-1]$, $B_{k+1}$); // inset l-tuple into bloom filter $k+1$
          break; // continue with next $l$-tuple
return($B_m$);
```

The spectrum is represented by the Bloom filter $B(T_{m.l}(R))$, which is subsequently transferred to the CUDA texture memory in order to be used for parallel error correction (see Section 4.3). An efficient way to compute $B(T_{m.l}(R))$ is shown in Algorithm 1. It uses m bloom filters, $B_1, \ldots, B_m$, to represent the multiplicity m. For each l-tuple, the algorithm queries the Bloom filters successively in descending order. Once a positive query is found for some k, B_{k+1} is programmed for the given l-tuple and the algorithm continues with the next l-tuple.

4.3. Parallel error correction

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 CUDA 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 read $r_i \in R$. The time complexity of the kernel is determined by Δ ($\Delta \geq 1$), the number of corrections within a weak l-tuple.

Our CUDA kernel for correcting *exactly* Δ mutations consists of two phases. The first phase is the Δ-mutations voting algorithm outlined in Algorithm 2. It identifies all l-tuples of the given read that are not in the spectrum (i.e., the Bloom filter). All possible Δ-point mutations of these l-tuples are then queried for membership in the spectrum. If successful, corresponding counters in the voting matrix are incremented at the positions corresponding to the mutations. Errors can be fixed based on high values in the voting matrix. In certain cases, for example, when $\Delta + 1$ errors are close two each other, the Δ-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 3 outlines the fixing/trimming/discarding procedure used as the second phase of our CUDA kernel.

Algorithm 2. Δ-Mutation voting algorithm used in the CUDA kernel

```
Input: Read $r_i[0 \ldots L-1]$, Bloom filter $B(T_{m.l}(R))$, and number of point-mutations $\Delta$ ($\Delta \geq 1$)
Output: Voting matrix $V_i[][]$ of size $L \times 4$
for $j := 0 \ldots L-1$ do
    for $c \in \{A, C, G, T\}$ do $V_i[j][c] := 0$;
for $j := 0 \ldots L-(l+1)$ do
    if (!query($r_i[j \ldots j+l-1]$, $B(T_{m.l}(R))$)) then
        mutate_and_vote($r_i[j \ldots j+l-1]$, $[-1, 0, \ldots, 0]$, $[0, \ldots, 0]$, 0);
return($V_i[][]$);
// recursive function to mutate all possible $\Delta$ positions in $l$-mer and do corresponding voting
void mutate_and_vote($t[0 \ldots l-1]$, $p[0, \ldots, \Delta]$, $c[0, \ldots, \Delta]$, $\delta$);
if ($\delta == \Delta$) then
    if (query($t$, $B(T_{m.l}(R))$)) then
        for $k := 1 \ldots \Delta$ do $V_i[p[k]][c[k]]++$; // increment voting matrix
else
    for $p[\delta] := p[\delta-1]+1 \ldots l-(\Delta-(\delta+1))$ do
        for $c[\delta] \in \{A, C, G, T\} \setminus \{t[p[\delta]]\}$ do
            $t[0 \ldots l-1] := t[0 \ldots p[\delta]-1] \cdot c[\delta] \cdot t[p[\delta]+1 \ldots l-1]$; // $\cdot$ denotes string concatenation
            mutate_and_vote($t[0 \ldots l-1]$, $p[0, \ldots, \Delta]$, $c[0, \ldots, \Delta]$, $\delta+1$);
return;
```

Algorithm 3. Δ-Mutation fixing/trimming/discarding procedure used in the CUDA kernel

```
Input: Read $r_i[0 \ldots L-1]$, Voting matrix $V_i[][]$ of size $(L-(l+1)) \times 4$, $B(T_{m.l}(R))$, and $\Delta$ ($\Delta \geq 1$)
Output: Fixed, trimmed, discarded, or unchanged read plus corresponding flag
if ($\max\{V_i[j][c]: j \in pos\_set;\ c \in \{A, C, G, T\}\} == 0$) return($r_i$, error_free);
$pos\_set = \{0, \ldots, L-(l+1)\}$;
$t[0 \ldots L-1] := r_i[0 \ldots L-1]$;
for $\delta := 1 \ldots \Delta$ do
    $(maxj[\delta], maxc[\delta]) := \text{argmax}_{(j \in pos\_set;\ c \in \{A, C, G, T\})}\{V_i[j][c]\}$;
    $pos\_set := pos\_set \setminus \{maxj[\delta]\}$
    $t[0 \ldots L-1] := t[0 \ldots maxj[\delta]-1] \cdot maxc[\delta] \cdot t[maxj[\delta]+1 \ldots L-1]$; // $\cdot$ denotes string concatenation
corrected_flag := true; trimmed_flag := false;
for $j := 0 \ldots L-(l+1)$ do
    if (!query(t[$j \ldots j+l-1$], $B(T_{m.l}(R))$)) then corrected_flag := false; else trimmed_flag := true:
if (corrected_flag) return($t$, fixed) // erroneous read has been corrected
else if (trimmed_flag) return($t[k_1 \ldots k_2]$, trimmed);
// return a trimmed read, where $t[k_1 \ldots k_2]$ is the longest substring of $t$ in which all $l$-tuples
//belong to $B(T_{m.l}(R))$
else return($\varepsilon$, discarded); // return discarded read as empty string
```

The time complexity of the kernel is dominated by the first phase. The operation that determines the runtime of the Δ-mutation voting algorithm is the Bloom filter membership test. The overall amount of membership queries by a single thread is $(L-l) \cdot p \cdot O(l^{\Delta})$, where p is the number of l-tuples of the read that do not belong to the spectrum.

Thus, our CUDA algorithm for correcting *up to* Δ mutations uses a filtration approach to reduce the amount of reads that are corrected with a large Δ-value. In the first step, Δ-mutation voting and Δ-mutation fixing/trimming/discarding is performed on the GPU only for $\Delta = 1$. In the next step, the CUDA kernel for $\Delta = 2$ is executed only for the set of reads that have been trimmed or discarded during the $\Delta = 1$ computation. This approach can then be continued for larger values of Δ. Filtration is quite effective since the amount of reads with no error or a single error is generally larger than the amount of reads with more than two errors (see Table 3 below).

The memory requirement for storing 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 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 4 KB 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 for error correction with up to two errors per read are as follows.

1. *Pre-computation on the CPU*: Compute $B(T_{m,l}(R))$ using Algorithm 1.
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:* Algorithms 2 and 3 for $\Delta = 1$.
4. *Data transfer from GPU to CPU*: Transfer set of error-free/corrected/trimmed/discarded reads to the CPU.
5. *Data transfer from CPU to GPU*: Transfer reads that are neither error-free nor corrected to the allocated global memory on the GPU.
6. *Execute CUDA kernel:* Algorithms 2 and 3 for $\Delta = 2$.
7. *Data transfer from GPU to CPU.* Transfer set of corrected/trimmed/discarded reads to CPU.

5. PERFORMANCE EVALUATION

We have evaluated the performance of our CUDA implementation using several simulated Illumina-style datasets as well as two real Illumina datasets. 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 0.58 to 4.71 Mbp). Three datasets have been created for each reference genome sequence using per-base error-rates of 1%, 2%, and 3%. The simulated dataset uses read length of $L = 35$ and $L = 70$ with coverage of $C = 70$. Thus, the size of simulated input datasets varies from

Table 1. Summary of 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%			
D3			3%			
E	S.aureus (NC_003923.1)	2.8M	1%	43		3.5M
F	Helicobacter (NC_008229)	1.6M	1.6%	190	36	8.2M
G1	H.inf (NC_007146)	1.9M	1%	70	70	1.9M
G2			2%			
G3			3%			

1.1M to 9.4M reads. The real datasets consist of 3.5M unambiguous reads (i.e., they do not contain any non-determined nucleotide) of length 35 each and have been downloaded from www.genomic.ch/edena.php and of 8.2M unambiguous reads of length 36 downloaded from http://sharcgs.molgen.mpg.de/download.shtml. The former has been obtained experimentally by Hernandez et al. (2008) using the Illumina Genome Analyzer for sequencing the *Staphylococcus aureus* strain MW2 (*H.Aci*) and the latter by Dohm et al. (2007) for sequencing *Helicobacter acinonychis.* We have estimated the error rate of the two real dataset as 1.0% and 1.6%, respectively, by aligning each read to the reference genome using RMAP (Smith et al., 2008). The 17 datasets used for performance evaluation are summarized in Table 1.

We have measured the runtime of these 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 1.3 GHz. The total size of the device memory is 1GB. The card is connected to an AMD Opteron dual-core 2.2-GHz CPU with 2-GB 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 August 14, 2008), which is available at http://euler-assembler.ucsd.edu. The code is a serial implementation of the Δ-mutation

Table 2. Runtime Comparison (in Seconds) between Parallel CUDA and Sequential Euler-SR Error Correction

		Euler-SR		*CUDA*		*Speedup*		
Dataset		*Count spectrum*	*Error correction*	*Count spectrum*	*Error correction*	*Count spectrum*	*Error correction*	*Total*
A1	1-error	37	127	41	10	0.90	12.70	3.22
	2-errors	41	345	41	26	1.00	13.27	5.76
A2	1-error	60	340	53	13	1.13	26.15	6.06
	2-errors	52	1454	53	70	0.98	20.77	12.24
A3	1-error	50	292	54	15	0.93	19.47	4.96
	2-errors	53	3239	54	92	0.98	35.21	22.55
B1	1-error	72	270	100	18	0.72	15.00	2.90
	2-errors	101	911	100	47	1.01	19.38	6.88
B2	1-error	81	505	102	24	0.79	21.04	4.65
	2-errors	86	2876	102	105	0.84	27.39	14.31
B3	1-error	91	696	104	27	0.88	25.78	6.01
	2-errors	94	6153	104	174	0.90	35.36	22.47
C1	1-error	135	556	177	32	0.76	17.38	3.31
	2-errors	149	1455	176	82	0.85	17.74	6.22
C2	1-error	158	888	180	42	0.88	21.14	4.71
	2-errors	161	5195	180	187	0.89	27.78	14.59
C3	1-error	192	1432	183	47	1.05	30.47	7.06
	2-errors	211	12042	183	315	1.15	38.23	24.60
D1	1-error	429	2122	410	101	1.05	21.01	4.99
	2-errors	430	4875	410	279	1.05	17.47	7.70
D2	1-error	588	2912	420	137	1.40	21.26	6.28
	2-errors	500	14793	421	669	1.19	22.11	14.03
D3	1-error	779	3573	430	163	1.81	21.92	7.34
	2-errors	763	30250	430	1153	1.77	26.24	19.59
E	1-error	130	641	183	35	0.71	18.31	3.54
	2-errors	134	13260	183	230	0.73	57.65	32.43
F	1-error	304	1322	362	92	0.84	14.37	3.58
	2-errors	376	7867	362	526	1.04	14.96	9.28
G1	1-error	246	1024	274	84	0.90	12.23	3.56
	2-errors	208	15758	274	229	0.76	68.81	31.74
G2	1-error	244	2521	281	122	0.87	20.66	6.86
	2-errors	245	578458	280	607	0.88	952.98	652.43
G3	1-error	306	2514	286	112	1.07	22.45	7.09
	2-errors	338	70312	286	832	1.18	84.51	63.19

Table 3. Percentage of Reads with a Certain Number of Errors (i.e., Mutations) for Various Error-Rates and Read-Length (Assuming a Non-Biased Error-Distribution)

Read-length	*Per-base error rate*	*Error-free reads*	*Reads with exactly 1 error*	*Reads with exactly 2 errors*	*Reads with more than 2 errors*
35	1%	70.34%	24.87%	4.27%	0.52%
	2%	49.31%	35.22%	12.22%	3.25%
	3%	34.43%	37.28%	19.60%	8.69%
70	1%	49.48%	34.99%	12.19%	3.34%
	2%	24.31%	34.73%	24.45%	16.51%
	3%	11.86%	25.67%	27.39%	35.08%

error correction algorithm presented in Section 4. However, different from our parallel method, it stores the spectrum in a sorted vector and then calls the efficient STL function std::lower_bound() for membership queries and does not use filtration for $\Delta \geq 2$ (but directly corrects errors for $1, \ldots, \Delta$, which is more efficient for a sequential implementation). CUDA generally does not support STL functions (as of CUDA 2.0), which was one of the reasons to use a Bloom filer data structure instead. Table 2 shows the runtime comparison between the sequential and the CUDA implementation for all datasets using the Euler-SR Illumina 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:

1) The speedup increases for higher error rates.
2) The speedup increases for $\Delta = 2$ (especially for longer read length).

Trend 1 can be explained as follows. The Δ-mutation voting algorithm contains the data-dependent conditional branch **"if** (!query($r_i[j \ldots j+l-1]$, $B(T_{m.l}(R))$)) **then"** (see Algorithm 2) 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 O(l^{Δ}) membership queries. 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 generally decreasing for higher error rates (Table 3). Thus, the number of waiting threads per warp is decreasing for higher error-rates, which in-turn improves the efficiency of the CUDA implementation.

Trend 2 is due to the filtration approach used in the parallel error correction algorithm. The double-mutation voting algorithm is only applied to the subset of reads that contain at least 2 errors (i.e., all reads that have not been be fixed by the single-mutation voting algorithm). Therefore, the data-dependent conditional branch **"if** (!query($r_i[j \ldots j+l-1]$, $B(T_{m.l}(R))$)) **then"** is true in most threads within a warp, resulting in a higher parallel efficiency compared to $\Delta = 1$. This is particularly evident for longer read lengths, since the amount of erroneous reads is larger (Table 3), which results in speedups of close to two orders of magnitude for the parallelized part for datasets G1 and G3 (even higher for G2; however, this appears to be an inefficacy of the sequential Euler-SR code).

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.

Table 4. Definition 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 5. Performance of the Read Identification Classification Test Measured in Sensitivity and Specificity for Each Dataset after Executing the CUDA Δ-Mutation Voting Algorithm

	TP	*FP*	*FN*	*TN*	*Sensitivity*	*Specificity*
A1	339700	23	701	813320	99.800%	99.997%
A2	578295	32	2380	573056	99.590%	99.994%
A3	746399	23	4330	403052	99.423%	99.994%
B1	643081	46	1353	1537422	99.790%	99.997%
B2	1093198	56	4552	1084129	99.585%	99.995%
B3	1410950	57	8489	762526	99.402%	99.993%
C1	1128525	48	2376	2698039	99.790%	99.998%
C2	1919892	47	8068	1901030	99.582%	99.998%
C3	2476109	73	15128	1337898	99.393%	99.995%
D1	2733995	170	5972	6539234	99.782%	99.997%
D2	4651239	321	19852	4608147	99.575%	99.993%
D3	5997457	300	37388	3244725	99.380%	99.991%
E	944906	106	3821	2480352	99.597%	99.996%
F	2820956	0	162698	5223249	94.547%	100.000%
G1	961586	18	2305	950587	99.761%	99.998%
G2	1435732	22	6818	471990	99.527%	99.995%
G3	1668830	20	11155	234628	99.336%	99.991%

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 4.

Sensitivity and specificity measures are then defined as:

$$\text{sensitivity} = TP/(TP + FN); \text{specificity} = TN/(TN + FP).$$

Table 5 shows the specificity and sensitivity measures for the 17 tested datasets. It can be seen that that the algorithm identifies erroneous reads with very high accuracy. Our implementation outputs four datasets:

Table 6. Reads in TP that Have Not Been Discarded (i.e., they are either corrected or trimmed)

	1-error (in %)			*2-error (in %)*		
Dataset	*Corrected*	*Trimmed*	*Accuracy*	*Corrected*	*Trimmed*	*Accuracy*
A1	75.7	22.5	99.5	83.5	15.7	99.4
A2	64.1	31.6	99.1	77.2	20.6	98.9
A3	53.7	38.9	98.9	69.9	26.2	98.4
B1	75.8	22.4	99.5	83.6	15.6	99.4
B2	64.0	31.7	99.1	77.2	20.7	98.9
B3	53.6	39.1	98.8	69.9	26.2	98.4
C1	75.8	22.4	99.5	83.5	15.6	99.5
C2	64.1	31.6	99.1	77.2	20.7	98.9
C3	53.6	39.1	98.8	69.9	26.2	98.4
D1	75.7	22.4	99.5	83.6	15.6	99.4
D2	64.1	31.6	99.1	77.2	20.7	98.9
D3	53.6	39.1	98.8	69.9	26.2	98.3
E	41.0	52.6	99.5	48.3	46.9	99.5
F	36.1	56.8	94.5	47.4	45.8	93.9
G1	76.5	20.3	99.5	87.5	10.3	99.4
G2	59.2	31.4	99.0	76.6	17.6	98.6
G3	43.8	38.7	98.7	63.6	25.8	97.6

The percentage of reads in the corrected/trimmed dataset that are accurately corrected/trimmed is also given.

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 6. The comparison of results for correcting up to one error and up to two errors is also provided. Furthermore, Table 6 shows the accuracy of the actual correction/trimming operation, i.e., whether correction/trimming is done at the correct read positions.

Table 6 shows that in 1-error and 2-error corrections, 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. For 2-error corrections, it can be seen that the percentage of corrected reads increased, and the trimmed and discarded reads decreased for higher error-rates compared with 1-error correction. For example, for corrected reads, there is a 10.3% increase in 2-error correction as compared to 1-error correction (1% error in dataset A1), 20.4% increase for dataset A2, and 30.2% for dataset A3. This shows that more erroneous reads can be fixed in 2-error correction algorithms. A further observation is that the percentage of corrected/trimmed 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. In a recent article, Dohm et al. (2008) have reported that error rates in Illumina reads range from 0.3% at the beginning of a reads to 3.8% at the end of reads.

6. CONCLUSION

Emerging HTSR sequencing technologies present 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. Error correction is an important but time-consuming pre-processing step for many *de novo* assembly tools. In this article, 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 in order to take advantage of the CUDA memory structure. Our performance evaluation on a commodity GPU, which is available for less than US$400, shows speedups between one and two orders of magnitude for datasets of various sizes and error rates at high correction accuracy. Our implementation is freely available for download from http://cuda-ec.sourceforge.net.

ACKNOWLEDGMENTS

The work was supported by an MOR Tier-1 grant and the NVIDIA Professor Partnership program.

DISCLOSURE STATEMENT

No competing financial interests exist.

REFERENCES

Batzoglou, S., Jaffe, D.B., Stanley, K., et al. 2002. ARACHNE: a whole-genome shotgun assembler. *Genome Res.* 12, 177–189.

Bloom, B. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 422–426.

Butler, J., MacCallum, I., Kleber, M., et al. 2008. ALLPATHS: de *novo* assembly of whole-genome shotgun microreads. *Genome Res.* 18, 810–820.

Chaisson, M.J., and Pevzner, P.A. 2008. A short-read fragment assembly of bacterial genomes. *Genome Res.* 18, 324–330.

Chaisson, M.J., Brinza, D., and Pevzner, P.A. 2009. *De novo* fragment assembly with short mate-paired reads: does the read length matter? *Genome Res.* 19, 336–346.

Chaisson, M.J., Tang, H., and Pevzner, P.A. 2004. Fragment assembly with short reads. *Bioinformatics* 20, 2067–2074.

Dohm, J.C., Lottaz, C., Borodina, T., et al. 2007. SHARCGS, a fast and highly accurate short-read assembly algorithm for *de novo* genomic assembly. *Genome Res.* 17, 1697–1706.

Dohm, J.C., Lottaz, C., Borodina, T., et al. 2008. Substantial biases in ultra-short-read data sets from high-throughput DNA sequencing. *Nucleic Acids Res.* 36, e105.

Hernandez, D., François, P., Farinelli, L., et al. 2008. *De novo* bacterial genome sequencing: millions of very short reads assembled on a desktop computer. *Genome Res.* 18, 802–809.

Lindholm, E., Nickolls, J., Oberman, S., et al. 2008. NVIDIA Tesla: a unified graphics and computing architecture. *IEEE Micro.* 28, 40–52.

Liu, W., Schmidt, B., Voss, G., et al. 2008. Accelerating molecular dynamics simulations using graphics processing units with CUDA. *Comput. Physics Commun.* 179, 634–641.

Manavski, S.A., and Valle, G. 2008. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinform.* 9, S10.

Mardis, E.R. 2008. The impact of next generation sequencing on genetics. *Trends Genet.* 24, 133–141.

Nickolls, J., Buck, I., Garland, M., et al. 2008. Scalable parallel programming with CUDA. *ACM Queue* 6, 39–55.

Pevzner, P.A., Tang, H., and Waterman M.S. 2001. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. USA* 98, 9748–9753.

Pop, M., and Salzberg, S.L. 2008. Bioinformatics challenges of new sequencing technology. *Trends Genet.* 24, 142–149.

Sanger, F., Nicklen, S., and Coulson, A.R. 1977. DNA sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci. USA* 74, 5463–5467.

Schatz, M.C., Trapnell, C., Delcher, A.L., et al. 2007. High-throughput sequence alignment using graphics processing units. *BMC Bioinform.* 8, 474, 2007.

Schmidt, B., Sinha, R., Beresford-Smith, B., Puglisi, S. 2009. A fast hybrid short read fragment and assembly algorithm. *Bioinformatics* 25, 2279–2280.

Smith, A.D., Xuan, Z., and Zhang, M.Q. 2008. Using quality scores and longer reads improves accuracy of Solexa read mapping. *BMC Bioinform.* 9, 128.

Strausberg, R.L., Levy, S., and Rogers, Y.H. 2008. Emerging DNA sequencing technologies for human genomic medicine. *Drug Discov. Today* 13, 569–577.

Tammi, M.T., Arner, E., Kindlund, E., et al. 2003. Correcting errors for shotgun sequencing. *Nucleic Acids Res.* 31, 4663–4672.

Warren, R.L., Sutton, G.G., Jones, S.J., et al. 2007. Assembling millions of short DNA sequences using SSAKE. *Bioinformatics* 23, 500–501.

Zerbino, D.R., and Birney, E. 2008. Velvet: algorithms for *de novo* short-read assembly using de Bruijn graphs. *Genome Res.* 18, 821–829.

Address correspondence to:
Dr. Bertil Schmidt
School of Computer Engineering
N40-2-32a Nanyang Avenue
Nanyang Technological University
Singapore 639798

E-mail: asbschmidt@ntu.edu.sg