Error correction in high-throughput short-read data on GPU

Aman Mangal, Chirag Jain

December 8, 2014

1 Abstract

Next generation sequencing (NGS) technologies have revolutionized the way of analyzing genetic information. It has eliminated the limitations of previous techniques in terms of throughput, speed, scalability and resolution. The effectiveness of NGS technologies is further improved by using error correction methodology on the imperfect though redundant sequence reads. In this project, we have utilized our HPC expertise gained in the CSE 6230 class and improved the run-time of GPU based solution (CUDA-EC) to this problem. We propose a parallel implementation for the same algorithm which reduces the kernel execution time by 14.81% which we have achieved by increasing warp efficiency from 3.5% to 64.6% and a few other code optimizations.

2 Introduction

Error correction of short NGS reads is an important component of genome remapping software programs, given huge length of human DNA (of the order of billion). The latest sequencing technologies like Illumina are known to generate more than million short reads of the genome with errors of about 0.5% to 2.5%. Fortunately, the redundancy in the data can be leveraged to bring this error down. Various proposed algorithms even use specialized hardware such as GPU to reduce the run time and achieve the results rather quickly. We have picked state of the art CUDA-EC[1] GPU implementation as the baseline error correction software to improve for this project. It builds a spectrum of substrings (or k-mers) that occur frequently in the datasets using a bloom filter. The spectrum thus constructed, is then used to correct each read in the dataset.

This report is organized as follows. We describe related work in section 3. Various profiling results for the original code are summarized in section 4. Section 5 describes some unsuccessful attempts which didn't have much effect on execution time. Section 6 lists various improvement we propose with corresponding experimental evaluation in section 7. We conclude and suggest some future work in section 8 and section 9 respectively.

3 Related Work

3.1 Parallel Implementations

Error correction of reads has been a widely discussed problem both in the sequential and recently for the parallel paradigms. Not only that the length of human genome is of the order of billion base pairs, on top of it, a big factor of redundancy is required for the error correction algorithms to precisely distinguish erroneous reads. As a result, parallel implementation becomes essential in order to quickly assemble genomes for practical purposes. Such algorithms exploit data parallelism as well as task parallelism for quick error correction. Shah et al. [2] propose a spectrum based parallel algorithm for distributed memory parallel computers and clusters. In this algorithm, each processor is allocated an equal share of the available reads.

3.2 GPU Implementations

In recent years, there have been multiple attempts to solve the error correction problem using Graphics Processing Units. All the solutions being described below, exploit the fact that individual reads can be corrected independently after the spectrum construction. These implementations have also utilized bloom filter[3], a space-efficient probabilistic data structure to hash the frequently occurring k-mers to cope up with the fact that the size of the spectrum or the k-mer frequency index built in the error correction problem can become as large as GPU's global memory size for big datasets.

Shi et al.[1] proposed the first GPU based solution which could achieve 3 to 63 times speedup against the sequential software EULER-SR[4]. They also report that the spectrum needs to be accessed repeatedly by querying the bloom filter throughout the error correction phase which makes it their runtime determining factor. We have chosen this software as the baseline implementation and improve upon it in this project.

Liu et al.[5] implemented the first hybrid CUDA-MPI distributed version of error correction algorithm. They solve the memory constraint problem that occurs for large-scale datasets by dividing the spectrum across the nodes. However, their algorithm doesn't show runtime scalability with respect to the number of nodes used due to all-to-all reductions involved.

4 Profiling Results

In the original CUDA-EC implementation, a single GPU thread corrects one read. The control flow of each thread depends on the contents of that read which makes efficient memory accesses and efficient utilization of compute units next to impossible. Many branches in the kernel depend upon the actual contents of the read and leads to warp divergence. The code also allocates memory within kernel which essentially uses the local memory of GPU. There is no use of shared memory at all. To confirm our intuition, we collected some statistics using *nvprof*. As shown in table 1, 3.5% is extremely low warp efficiency. The register usage of 85 per each thread limits the occupancy to 24.6%, thus limiting the kernel's ability to fully utilize the GPU.

Metric	**Value**
Warp execution efficiency	3.5%
L1/Shared bandwidth	57.6 GB/s
L2 bandwidth	7.5 GB/s
Device memory bandwidth	2.0 GB/s (Limit: 208 GB/s)
Shared memory usage	0 bytes
Register usage	85 per thread
Occupancy	24.6%

Table 1: Profiling results of original code computed on Kepler K20m GPU

5 Unsuccessful Attempts

Based on the profiling results obtained, we attempted some code improvements which didn't seem to have much effect on execution time. We describe these efforts in this section.

5.1 Bloom Filter Access Throughput

As the first modification step, we began by following CUDA-EC author's claim that the memory throughput for queries to the bloom filter is crucial for the overall performance. The current implementation makes random memory accesses to the texutre memory while checking the existence of an element in the standard implementation of bloom filter used in CUDA-EC. Size of the bloom filter has been chosen to be 64 times the number of elements which is roughly 18 MB for the dataset being used. It uses 8 hash bits from 8 hash functions to save the element in the table. Therefore, a positive query involves checking 8 bits, but a negative query involves checking 8 or less than 8 bits. Since the filter is not modified while code is executing on GPU, it has been saved in the texture memory of GPU.

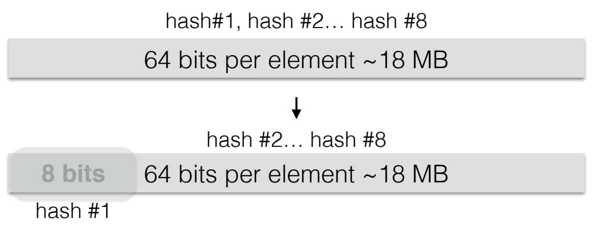

Figure 1: Modified bloom filter architecture

While profiling CUDA-EC on *SRR006331* dataset, we found that 97.4% of the queries are negative. Furthermore, the average number of bits checked during negative queries is 1.07. This made the optimization for memory throughput difficult in this case. We tried changing the bloom filter design to make it more cache efficient. We reduced the size of memory region for the first access to see if this direction yields any positive results (Fig 1). Our change improved the cache efficiency (Table 2), but it didn't yield good timings because of increment in the count of total memory accesses.

Metric	**Value (Before)**	**Value (After)**
Time	5.73 sec	6.16 sec
Texture L2 hit rate	1.96%	14.45%
L2 texture throughput	2.76 GB/s	6.29 GB/s
Total texture accesses	68,381,701	91,551,531

Table 2: Adverse effect of changing bloom filter on the execution time

We also reviewed some papers on implementing cache-efficient bloom filters such as cache partitioned filters [6] and blocked bloom filters[7]. These techniques improve cache efficiency by reducing the distance between the bits corresponding to same elements, in the bloom filter. In our case, only 1.25 bits per element are accessed while querying the bloom filter which indicates that using these techniques will not have much effect on execution time.

5.2 Reads in Shared Memory

Reads are frequently accessed in the current implementation of CUDA-EC which is stored in the global memory of GPU. To speedup the access, we copied the data from global memory to shared memory of GPU. This reduced GPU execution time to 5.11 seconds from 5.6 seconds. Though, in our final implementation, we saw no significant improvement because of using shared memory due to the fact that we use fewer reads at a time per block which allowed the reads to fit in cache which was not the case earlier.

6 Proposed Improvements

We suggest some improvements in the CUDA-EC code in subsequent subsections with their corresponding effects on GPU execution time. The final code comparison with the original code is show in table 3

6.1 Warp Divergence

Instead of having a single thread correcting a single read, we restructured the code by having a single warp correcting a single read to improve the warp execution coherence. We did this by parallelising multiple phases of the kernel. The availability of $\_\_any()$ and $\_\_all()$ calls on

Fermi GPUs helped us to communicate among threads in a warp. This step is supposed to help us in the following ways-

- Improving warp execution efficiency
- Improving memory throughput as memory accesses become more coalesced if all the warps do similar work
- Less on-chip and local memory usage as each GPU block with n threads works on n/32 reads instead of n reads at a particular instant
- No synchronization overhead because threads in a warp are synchronized by default

6.2 Using Shared Memory

We made following attempts in order to reduce the usage to local memory in the block and increase the shared memory usage-

- At each iteration, we copy the read needed by a warp into shared memory by making a warp access continuous chunk of global memory. We saved it back into the global memory after correcting it. Though, we later found out that this didn't have any significant improvement on the execution time as explained in section 5.2
- In order to find the consensus fix in a read, a voting buffer was being used in local memory earlier which is now shifted to shared memory
- An extra buffer is used (length = number of threads in a warp $\times$ number of warps in a block) to find out maximum value stored in the voting buffer while reducing it
- Removed other three buffers (solid, maxPos, maxMod) which were used in the code earlier as we found out an easy way to avoid them during the computation using local variables and intra-warp communication

6.3 Other Improvements

We performed some optimization on the original code which had significant impact on execution time of *kernel*. We describe them as follows-

- Loop unrolling, loops with fixed number of iterations are unrolled to decrease the executed instructions count. For example, there are only 3 mutation possible for a given position in a read (A $\rightarrow$ T $\rightarrow$ C $\rightarrow$ G)
- Reduced register count by removing unnecessary variables. This helps in increasing the occupancy of the kernel as it was restricted by high register count
- Avoided unnecessary copy of reads into another buffer stored in global memory which was present in the original code

- Removed unnecessary if-else branches where the condition could be computed at compile time or even before.

7 Experimental Evaluation

For the experiments, all the timings are measured on the M2090 GPUs of jinx cluster. Due to the unavailability of latest CUDA version on jinx, the code profiling was done on a separate system with Kepler K20m GPU running CUDA 6.5. All the experiments in this report have been done using Illumina dataset SRR006331 which contains 1.7 million reads of 36 characters each.

If we combine all the techniques used to improve the runtime, we see that all the metrics in our warp efficient code improve by a significant factor (Table 3). The detailed analysis of speedups achieved by the individual improvements done is shown in Table 4.

Metric	**Value (Prev)**	**Value (Now)**
Warp execution efficiency	3.5%	65.7%
L1/Shared bandwidth	57.6 GB/s	166.4 GB/s
L2 bandwidth	7.5 GB/s	40.3 GB/s
Device memory bandwidth	2.0 GB/s	37.31 GB/s
Shared memory usage	0 bytes	1.95 KB
Register usage	85 per thread	56 per thread
Occupancy	24.6%	47.1%

Table 3: Profiling results of the new implementation on Kepler K20m GPU

Step	**GPU execution time (sec)**	**Speedup**
Original code	5.67	-
Original code with only one thread executing per warp	26.17	1
Other Optimization	5.85	x
Loop unrolling	5.78	x
Using Shared Memory in place of global memory	5.94	x
Warp Divergence	4.94	x

Table 4: Speedup achieved after every modification made in the implementation

8 Conclusion

Improving warp efficiency and couple of other techniques helped us in improving memory bandwidth by significant factor at all levels of memory hierarchy and warp efficiency from

3.5% to 65.7%. Regardless of few failed attempts, we succeeded in improving the runtime of the software by X. Since this algorithm performs heavy arithmetic with the available data, the runtime of our code is compute and latency bound. As a check for the code being correct, we have made sure that the corrected output sequences of the software is precisely same as the new implementation we proposed.

9 Future Work

New instructions such as *shuffle* have been introduced in the newer GPU architectures which make it easier to carry out reduction operations and communication within a warp. We hope to further improve the warp efficiency with these instructions. CUDA-EC software is also not able to scale to large reads and data sets have more than 20 million reads. Making this software work on even bigger and latest NGS datasets will also be interesting and will have fruitful impact in this area.

We were also not able to execute CUDA-EC on various datasets with different parameters on jinx cluster. We assumed the constraint to modify the existing software in a way that the output does not change at all in order to prove that we have improved the efficiency of the software without changing the output. If we rewrite it from scratch using latest version of CUDA, we could make it compatible for more datasets as well as more efficient.

References

[1] H. Shi, B. Schmidt, W. Liu, and W. Müller-Wittig, "A parallel algorithm for error correction in high-throughput short-read data on cuda-enabled graphics hardware," *Journal of Computational Biology*, vol. 17, no. 4, pp. 603–615, 2010.

[2] A. R. Shah, S. Chockalingam, and S. Aluru, "A parallel algorithm for spectrum-based short read error correction," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 60–70, IEEE, 2012.

[3] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[4] M. J. Chaisson and P. A. Pevzner, "Short read fragment assembly of bacterial genomes," *Genome research*, vol. 18, no. 2, pp. 324–330, 2008.

[5] Y. Liu, B. Schmidt, and D. L. Maskell, "Decgpu: distributed error correction on massively parallel graphics processing units using cuda and mpi," *BMC bioinformatics*, vol. 12, no. 1, p. 85, 2011.

[6] I. Moraru and D. G. Andersen, "Fast cache for your text: Accelerating exact pattern matching with feed-forward bloom filters," 2009.

[7] F. Putze, P. Sanders, and J. Singler, "Cache-, hash-and space-efficient bloom filters," in *Experimental Algorithms*, pp. 108–121, Springer, 2007.

A Original Pseudo Code

```
Allocate vote[] ;                                              // In local memory
while $read\ chunk \neq Empty$ do
    /* Access to read for this thread $tid$                                   */
    $current\_read \leftarrow global\_mem\_all\_reads[tid]$;
    while $current\_read\_correction\_ongoing$ do
        for $Every\ tmpTuple\ k\text{-}mer\ in\ current\_read$ do
            if $tmpTuple\ is\ not\ solid\ i.e.\ rare$ then
                for $Each\ character\ position\ i\ in\ tmpTuple$ do
                    for $newtmpTuple \leftarrow mutate\ i\ with\ other\ nucleotides\ m : (0 \dots 3)$ do
                        if $newtmpTuple\ is\ solid$ then
                            Update $vote[i][m] += 1$
                        end
                    end
                end
            end
        end
        if $all\ tmpTuple\ were\ solid$ then
            $current\_read\_correction\_ongoing \leftarrow$FALSE
        end
        /* Sequential iteration over $vote[][]$ to find maximum
           $maxVote = vote[p][q]$                                              */
        /* Update $current\_read[p]$ = Nucleotide[q]                           */
    end
end
```

Algorithm 1: Original CUDA-EC algorithm

B Modified Pseudo Code

```
Allocate vote[] ;                                          // In shared memory
while read chunk $\neq$ Empty do
    /* Access to read for this warp tid                                     */
    /* Threads in warp load read to shared mem from global                  */
    current_read $\leftarrow$ shared_mem_all_reads[warp_id];
    while current_read_correction_ongoing do
        for Every tmpTuple k-mer in current_read do
            if tmpTuple is not solid i.e. rare then
                for Each character position $i = 0$ in tmpTuple;
                i+=WARPSIZE do
                    for newtmpTuple $\leftarrow$ mutate $i$ with other nucleotides $m : (0 \ldots 3)$ do
                        if newtmpTuple is solid then
                            Update $vote[i][m] += 1$;
                            tmpTupleUnChanged = false;
                        end
                    end
                end
            end
        end
        if __all(tmpTupleUnChanged) then
            current_read_correction_ongoing $\leftarrow$FALSE
        end
        /* REDUCTION within warp for vote[][] to find maximum
            $maxVote = vote[p][q]$                                          */
        /* Update current_read[p] = Nucleotide[q]                           */
    end
end
```

Algorithm 2: New CUDA-EC execution flow