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

Aman Mangal, Chirag Jain

December 6, 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 utilised our HPC expertise gained in the CSE 6230 class and improved the runtime of GPU based solution (CUDA-EC) to this problem. We propose a different parallel implementation for the same algorithm which reduces the kernel execution time by 17.49%. We have increased warp efficiency from 3.5% to 71%.

2 Introduction

Error correction of short NGS reads is an important component of the 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. We started with a goal to improve the timings of the CUDA-EC software that utilizes Graphics Processing Unit to do the error correction. In order to save the consensus substrings in minimal memory, it uses a bloom filter to hash them.

As the first step, we begin by following the authors claim that memory throughput for the queries to the bloom filter in the texture memory. We reviewed some papers on implementing cache-efficient bloom filters such as cache partitioned filters and blocked bloom filters. We tweaked the implementation by changing the size of bloom filter to test if improving the memory throughput from texture memory helps. But we didnt see the improvements in the timings. (TODO: FIGURES) This proved that there must be other bottlenecks in the code we should target first. In the subsequent sections, we will go over the code improvements that we further add and their impact on the memory throughput and timings of the code.

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. Length of the genome is of the order of billion base pairs in case of humans. 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.

Shah et al.[1] propose a spectum based parallel algorithm for distributed memory parallel computers and clusters. In this algorithm, each processor is allocated an equal share of the reads.

3.2 On the GPU

In the 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. Secondly, size of the spectrum or the k-mer frequency index built in the error correction problem can even be bigger than the GPU's global memory size. Therefore, all the GPU implementations so far have utilized the bloom filter[2], a space-efficient probabilistic data structure to hash the frequently occurring k-mers.

Shi et al.[3] proposed the first GPU based solution which could achieve 3-63 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 choose their software as the baseline implementation and improve upon it.

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 data sets 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 Bottlenecks & Profiling results

In the original implementation, a single GPU thread corrects one read. The execution flow of each thread depends on the contents of that read making efficient memory accesses and efficient utilisation of compute units impossible. Many branches in the kernel depend upon the actual contents of the read.

The code also allocates memory within kernel which essentially uses the slow local memory of GPU. Also, there was no use of shared memory at all. To confirm our intuition,

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
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

we collect some statistics from *nvprof* and profile this code. 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 utilise the GPU.

5 Improvements

5.1 Warp Divergence

Instead of having a single thread correct a single read, we restructured the code by having a single warp correct 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 warps. 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 needed among threads in a warp.

5.2 Using Shared Memory

We make the following attempts in order to reduce the usage to local memory in the block and increase the shared memory use.

- At each iteration, we load the read needed by a warp into shared memory by making a warp access continuous chunk of global memory. We save it back into the global memory after correcting it.

Warp execution efficiency	71%
L1/Shared bandwidth	199.0 GB/s
L2 bandwidth	29.4 GB/s
Device memory bandwidth	30.6 GB/s
Shared memory usage	1.45 KB
Register usage	64 per thread
Occupancy	46.9%

Table 2: Profiling results of original code computed on Kepler K20m GPU

- In order to find the consensus fix in a read, a voting buffer being used in local memory before was now shifted to local memory.
- The other three buffers were removed as we found an easy way to avoid them during the computation.

5.3 Other Improvements

- Loop unrolling
- Reducing register count by decreasing the count of variables needed. This helps in increasing the occupany of the kernel as it was restricted by high register count.

6 Experimental Evaluation

We see in table 2 that all the metrics in the warp efficient code improve by a big factor.

7 Conclusion

8 Future Work

9 Acknowledgement

References

[1] 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.

[2] 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.

[3] 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.

[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.