An Implementation of the Spectrum-based Short Read Error Correction Framework on GPU

Jammula Naga Kishore, GT ID: 902642756
Final Report, Individual Project, CSE 6140, Fall 2013

Abstract—The objective of this class project is to investigate the implementation of spectrum based error correction framework on the GPU platform. Improving the runtime of pre-assembly error correction holds great significance for performing efficient sequencing and accurate assembly of genomes which are billions of base pairs long. We describe in detail how each step of the framework can be implemented effectively on the GPU platform. Currently, we have the complete implementation for the k-spectrum construction phase and are working on the implementation for the error correction phase. In the later part of the paper, we describe how a significant shortcoming of the existing spectrum based error correction framework targeted for parallel computers and clusters can be addressed. Further, we provide insights into how this framework can benefit from the newly released Automata Processor accelerator platform.

Index Terms—error correction; hamming distance; hardware acceleration; GPU;

1 Introduction

DNA sequencing is a fundamental tool of genomics research. For a long period of time, Sanger sequencing, reading up to 1000 base pairs of DNA at a time, has been the primary sequencing methodology. In the past few years, a number of pivotal technologies have revolutionized DNA sequencing. Common to these systems is the ability to generate millions of concurrent reads, with throughput of some systems reaching over a billion reads per experiment. Collectively known as next-generation sequencers, these low cost-cost systems are ushering in revolutionary advances in genomics research by changing the type and/or scale of possible genomics projects.

The reads produced from a sequencing experiment are put back together to infer the original genome through a process called de-novo assembly. The software used for this purpose is referred to as assembler. However, the reads produced during the sequencing experiment contain errors owing to the limitations of the instruments used. The errors can be categorized into two types - substitution and insertion-deletion (indel). Identifying and correcting the errors prior to the assembly process is important to produce high quality assembly. Further, the memory and time requirements for assembly are significantly reduced by employing pre-assembly error correction.

There is considerable amount of prior work which targeted the improvement of error correction accuracy. We survey some of these works in the following section. Sequencing of genomes which are billions of base pairs long (human genome for example has more than 3 billion base pairs) using the next-generation sequencing (NGS) technology produces large quantities of data in the form of short reads. Existing error correction solutions do not scale to such large quantities of data. There are two works which investigated parallelizing spectrum based short read error correction - the first one suited for distributed memory parallel computers and clusters [1] and the second one suited for GPUs [2]. In this project, we developed a parallel spectrum-based error correction solution suitable for implementation on a GPU. Our solution approach is modeled after the one proposed by Shah et al. [1] and addresses the limitations of the approach proposed by Shi et al. [2].

The rest of the paper is organized as follows. Section 2 discusses the background material related to various flavors of error correction. In Section 3, we thoroughly analyze the previously proposed parallel error correction solutions and compare them with our approach. Our solution approach for implementing the spectrum-based short read error correction on a GPU is presented in Section 4 and the corresponding results in Section 5. Finally, we provide some interesting avenues for future work in Section 6 and our conclusions in Section 7.

2 Background

While generating reads from a genome, errors appear infrequently and randomly. When generating reads from a genome using NGS technology, each base in the genome is sampled multiple times, which is often referred to as the coverage. Based on these two observations, error correction for a particular base can be achieved by examining all the reads covering the specific position. Since the reference genome is not known, reads are assumed to belong to the same genomic location if they share sub-reads, such as k-mers. Genomic repeats and non-uniform sampling result in ambiguity in correction owing to multiple equally likely correction choices. The error correction methods that were proposed previously can be classified into three types - k-spectrum based, suffix tree/array based, and MSA based [3].

k-spectrum based: These methods are mainly targeted towards NGS datasets in which substitution type errors occur predominantly. Each read is decomposed into (n-k+1) sub-reads called k-mers by reading substrings of length *k* starting at each position. Once the pool of k-mers is generated, each is classified as either solid or insolid. A k-mer which occurs more number of times than a threshold value is labelled as solid. This threshold value is influenced by the coverage of the genome. The remaining k-mers are marked as insolid. The objective of the error correction process is to convert insolid k-mers into solid k-mers with a minimum number of edit operations. The correction choices for insolid k-mers are generated by examining the solid neighbors using the hamming distance metric. Reptile [4] is a recent error correction algorithm which is based on the k-spectrum approach.

Suffix tree/array based: The error correction methods based on suffix tree/array generalize the k-mer based approach and handle multiple *k* values and the corresponding threshold value. When a suffix tree is used, for each internal node, the concatenation of the edge labels from the root to the node represents a substring that occurs in the input. When handling substitution errors, the potential correction possibilities for the substring represented by a node are obtained by examining the substrings represented by the siblings of the node. Further these methods are capable of handling indel errors. The potential correction possibilities for the substring represented by a node are obtained by examining the siblings of the parent of the node and the children of its siblings in case of an insertion and deletion respectively. HiTEC [5] is an error correction algorithm based on the suffix array approach.

MSA based: In this approach, reads co-located on the unknown reference genome are identified by using k-mers as seeds. Multiple sequence alignment (MSA) is then obtained by considering all the co-located reads. The approach can be applied to read data sets containing substitution errors or indel errors by controlling penalties for edit operations. Corrections are applied if the number of reads involved in the MSA falls within a range, and the largest edit distance between any constituent read and the the consensus of the MSA is relatively small. Coral [6] and ECHO [7] are representative error correction algorithms which work based on multiple sequence alignment.

The errors produced by some of the next-generation sequencers such as those from Illumina are predominantly of substitution type while those produced by others such as those from Ion Torrent are predominantly of indel type. As the instruments manufactured by Illumina are deployed widely, most of the error correction works targeted substitution type errors. In this project, we developed a generic framework for implementing a spectrum based error correction algorithm on a GPU, targeted towards substitution type errors. In the following section, we provide a critical analysis of two previously proposed solutions for parallelizing spectrum based error correction.

3 Related Work

Shi et al. [2] implemented the spectrum based error correction framework on an Nvidia GPU using the CUDA framework. They used a counting-variant of the Bloom filter data structure to distinguish between solid and insolid k-mers. Recall that we covered this probabilistic data structure recently in class as a part of the lecture on *Randomized Algorithms*. The data structure provides a *may-be-yes* or a *definite-no* in response to a query. By appropriately choosing the set of hash functions and the size of the array, a low false positive rate can be achieved. In theory, the choice of the bloom filter data structure for representing the k-spectrum seems to be a very good decision. However, in the actual implementation, the data structure is mapped to the texture memory on the GPU using a 1D texture. The texture memory on Nvidia GPUs is backed by an L1 texture cache on each streaming multiprocessor (SM). The size of this cache on the latest generation (Kepler) GPUs from Nvidia is 48KB. As the bloom filter data structure needs to be accessed multiple times for each query, the performance will degrade significantly unless the entire data structure fits into the L1 texture cache. The size of the counter used by Shi et al. is 4-bits. With this counter size, the number of elements that can be cached is limited to 96K. This number is severely restrictive in comparison to the size of the genomes which are billions of base pairs long.

Shah et al. [2] developed a spectrum based error correction framework for implementation on distributed memory parallel computers and clusters. The framework comprises of the following steps: (1) the read set is evenly distributed among the processors (2) each processor generates an unordered multiset of k-mers (represented as integers) based on the read set allocated to it (3) each processor sorts its list of k-mers locally to eliminate duplicates and obtain frequencies (4) the entire list of k-mers across all the processors is then globally sorted using a parallel sample sort (recall that we covered parallel sample sort in our class) (5) k-spectrum comprising of only solid k-mers is then built (6) the k-spectrum is copied on all processors (7) reads are distributed among processors and the error correction is performed independently on each processor. The authors demonstrated that the implementation scales well as the number of processors are increased thus enabling error correction for very large data sets. Our GPU implementation is inspired by this approach and attempts to accomplish large scale error correction at a fraction of the cost incurred by distributed memory parallel computers and clusters. Next, we describe the details of the implementation of spectrum based error correction framework on the GPU.

4 Solution Approach

Our GPU based implementation for the spectrum based error correction framework works as follows.

1. Generation of the k-mer multiset: Once the read dataset is copied on to the device memory, this step can be parallelized in a very straight forward manner. Each read is assigned to a GPU thread and the k-mers for the read are generated by the corresponding GPU thread and stored in the global memory. Each read is accessed multiple times by a GPU thread to generate the full list of k-mers and therefore this step benefits from storing the reads in the shared memory or the hardware caches built into the GPU.

2. Sorting the k-mer multiset: As we specified previously, k-mers are represented as integers where each letter in the k-mer (A, C, G, or T) corresponds to two binary digits. We use a parallel implementation of the bitonic sort algorithm (Fig. 1) targeted for GPU to perform the sorting step. Further details about how we optimized the bitonic sort algorithm for the GPU and the corresponding improvement in performance obtained are discussed in the *Results* section.

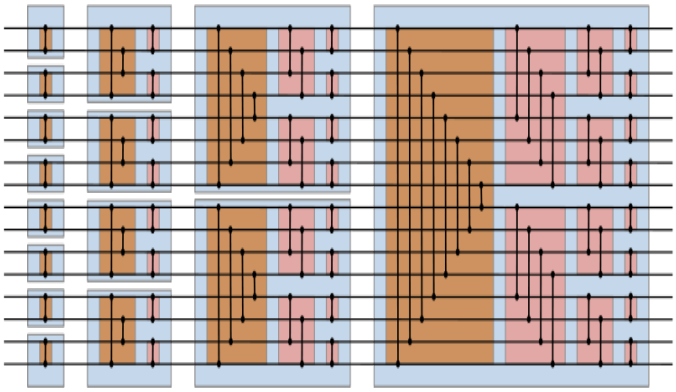

Fig. 1. An illustration of the bitonic sorting network for 16 inputs. Figure due to Wikipedia.

3. Frequency counting and duplicate elimination: During this step, a linear scan is performed through the sorted k-mer multiset. During the scanning process, all but one copy of the k-mer are eliminated and the frequency of occurrence of each k-mer in the list is computed. Based on the computed frequency, each k-mer is classified as either solid or insolid and inserted into the corresponding sublist. Note that multiple copies of the k-mer which exist in the multiset are located next to one another as a consequence of the sort operation. This step marks the completion of the spectrum construction phase of the algorithm. At the end of this phase, we are left with two sublists. The first sublist comprising of the solid k-mers and the second sublist comprising of the remaining (insolid) k-mers which need to go through the correction process.

4. Error correction: During this step, correction is attempted for insolid k-mers. Specifically, for each insolid k-mer, neighbors are generated based on the hamming distance metric (recall that we are only interested in substitution errors) and checked if they are valid. To check if a neighbor is valid, a binary search is performed on the list of solid k-mers which constitutes the k-spectrum. This step is the compute intensive part of the whole implementation and lends itself well to parallelization. However, implementing this step on the GPU involves overcoming some interesting challenges. We use the van Emde Boas layout [8] (cache-oblivious data structure covered in our class) to store the k-spectrum (sorted list of solid k-mers) and store as much of it as possible in the shared memory to support efficient accesses (Fig. 2). We assign each invalid k-mer to a GPU thread in order to find its valid hamming distance neighbors. Therefore, care should be taken to ensure that threads in a warp do not diverge. We address the divergence problem as follows. As the list of insolid k-mers is also sorted, the hamming distance between closely located insolid k-mers is expected to be small. As a consequence, there will be significant overlap among the set of hamming distance neighbors for such insolid k-mers.

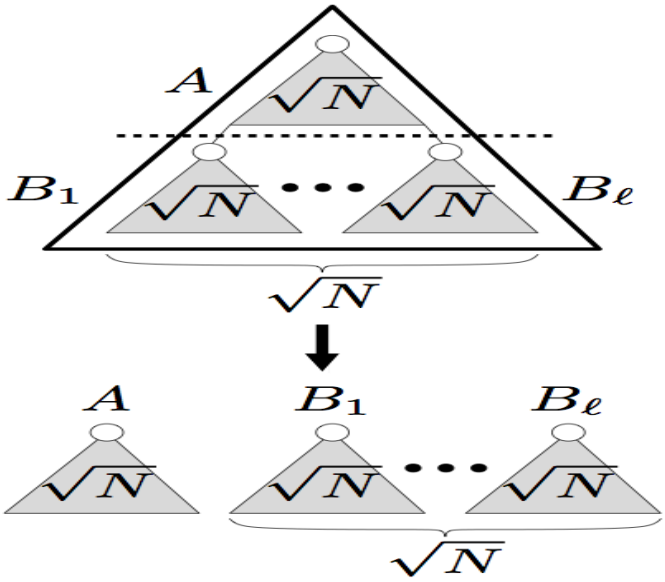

Fig. 2. A binary tree stored as a cache oblivious data structure using the van Emde Boas layout in a recursive manner. Figure due to Bender et al. [8].

Steps 1, 2, and 3 correspond to the spectrum construction phase and step 4 corresponds to the correction phase. As we previously mentioned, error correction phase incurs the most amount of running time, which further increases as the value of the hamming distance considered increases. In practice, neighbors of insolid k-mers which are atmost at a distance of 3 are explored.

5 Results

Currently, we have the implementation for the k-spectrum generation phase of the algorithm (steps 1-3) and are working on the implementation for step-4. We expect to complete the implementation for step-

4 very soon. Now, we explain how we optimized the bitonic sort step which is the most time intensive step in the k-spectrum generation phase and provide the corresponding improvement in performance obtained. All our experiments were conducted on a single node on the Jinx cluster and used a Nvidia Tesla M2090 GPU (Fermi generation).

With our initial implementation, we achieved a performance number of 5.70 GB/s. We measure the performance in terms of the number of bytes sorted per second. This is much faster than the sequential version but still comes up short compared to the maximum performance that can be achieved on the M2090 GPU. Starting from this version, we implemented a number of performance optimizations specific to the GPU architecture.

1. Shared memory: The first optimization that we implemented is to use shared memory in the last few rounds of the bitonic split kernel. Bitonic merging starts off with a bitonic sequence of a given array size, and at every round splits the given sequence into two equal subsequences. The communication required for doing this split is wholly contained within the sequence. Therefore, considering a maximum offset threshold of 1024, the next log2(1024), or 10 rounds of bitonic split can be conducted by working with only 2048 elements of a bitonic subsequence. We can copy these 2048 elements into the shared memory. After doing this, we would no longer need to access global memory until we write the result. In other words, we can conduct 10 rounds of bitonic split in a single kernel whose threads work with the shared memory. With this change, we were able to get a performance number of up to 6.77 GB/s.

2. Work while copying to shared memory: A simple optimization which we noticed next is as follows. Instead of purely copying to shared memory and then doing bitonic split, we could just do the first round right away, and do away with a loop. We are reading the value from global memory anyway; why not do a MIN and MAX comparison while we are at it? This quick change improved our performance slightly, up to 6.98 GB/s.

3. Algorithmic cascading (loop unrolling): As a next step, we implemented algorithm coarsening for those rounds which did not make use of shared memory. Specifically, we tried two solutions. In the first solution, we merged 2 independent kernel launches into 1. The corresponding performance is shown below (bitonicSplit_2). In order to be able to merge 2 launches into 1, we need to reduce the number of threads to one-half. The key idea is that we make a single thread responsible for handling two comparisons during each iteration to overcome synchronization problem. Based on the success from merging 2 kernels, we implemented merging of 3 kernels as well. The resulting performance is shown against bitonicSplit_3. bitonicSplit_4 can be envisioned in a similar manner but we did not implement it. bitonicSplit_2 and bitonicSplit_3 kernels use 15 and 29 registers respectively per thread. We also unrolled the computation in case of both kernels.
-bitonicSplit_2: 8.8 GB/s; -bitonicSplit_3: 8.4 GB/s

4. Use of registers: When we implemented the coarsening step above, we observed that we were reading from global memory and writing to global memory corresponding to each iteration. We realized that intermediate accesses to global memory can be avoided by storing the intermediate results into registers. This optimization enhanced performance both for bitonicSplit_2 and bitonicSplit_3 kernels as shown below. We hypothesize that such an optimization will be even more beneficial for bitonicSplit_4 provided we do not overuse the registers in a manner where occupancy is reduced.
-bitonicSplit_2: 10.4 GB/s; -bitonicSplit_3: 12.54 GB/s

5. Adjust block size: We varied block size over and above the previously described optimizations to see if we can obtain a further improvement in performance. The results are shown below for three different block sizes and they are nearly the same. We believe this is because we have good occupancy in all cases.
-128: 12.50 GB/s; -256: 12.54 GB/s; -512: 12.58 GB/s

6. Use of faster operations (modulus, division): One quick improvement that we thought is possible is to use more optimal processor instructions instead. The modulus operator can be substituted to a subtraction with a bitwise and operation, and the division operator can be substituted with a right shift operator. This optimization did not yield any performance improvements.

7. Re-arranging operations to improve ILP: Another possible improvement that might be useful is to rearrange the sequence of operations in our GPU kernel so that we can have additional operations to perform while the slower ones are still in progress. Our unrolled kernel had two types of operations; calculating the indices and doing the bitonic split on these indices. However, rearranging these had negligible effect on performance.

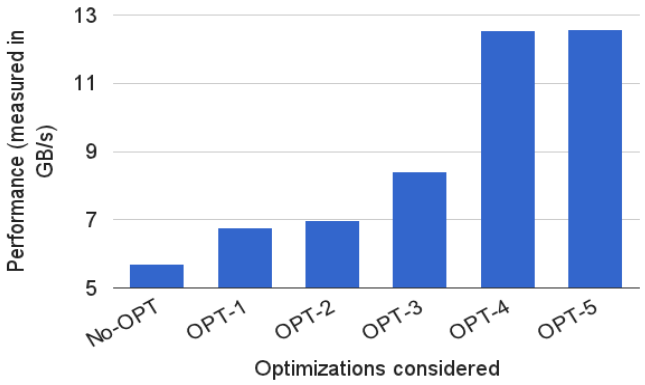

Fig. 3. Improvement in performance obtained corresponding to the optimizations applied for bitonic sort algorithm.

6 Discussion

While working on the GPU implementation for spectrum based error correction framework, we uncovered a shortcoming of the framework implemented by Shah et al. [1] for distributed memory parallel computers and clusters.

As we mentioned previously, error correction phase is the most time intensive step of the framework. During this phase, identification of solid hamming distance neighbors of insolid k-mers involves multiple binary searches over the k-spectrum. The number of searches increases as the power of k as the hamming distance considered is increased. Shah et al. used a linear layout of the sorted list which is inefficient, especially when performing large number of searches. The number of cache misses can be reduced and the performance can be improved potentially by a factor of up to (cache-line-size/element-size) by using the cache-oblivious van Emde Boas layout [8] discussed in class.

Automata Processor (AP, which was discussed in class on 12/03) is a recently released non Von Neumann architecture based accelerator platform from Micron [9]. The platform can simultaneously execute a large number of non-deterministic finite automaton (NFA) over streaming data. AP can be used to accelerate the spectrum based error correction framework as follows. The k-mer list is enumerated on the CPU by making a linear scan through the reads. Automata for identifying exact pattern match are then programmed onto the AP. These automata are labelled with k-mers. Then the list of k-mers is streamed through the AP to distinguish between insolid and solid k-mers. At the end of this step, the k-mers occurring with a frequency above a threshold value are classified as solid and the remaining ones as insolid. In the next pass, automata are only labelled with insolid k-mers and solid k-mers are streamed through the processor. During this step, by programming the AP with *Bounded Mismatch Identification Automaton* [10], all the solid hamming distance neighbors of the insolid k-mers can be identified. Note that the described implementation does not involve sorting of the multiset of k-mers, which is a requirement on all other platforms.

7 Conclusions

In this project, we investigated the implementation of spectrum based error correction framework on the GPU platform. Improving the runtime of pre-assembly error correction holds great significance for performing sequencing and assembly of genomes which are billions of base pairs long. Currently, we have the complete implementation for the k-spectrum construction phase and are working on the implementation for the error correction phase. In addition to the future work opportunities described in the previous section, following avenues exist for enhancing and extending the scope of the current project.

- We have only evaluated the bitonic sorting algorithm for this project. The sorting step takes up a significant fraction of the time incurred by the k-spectrum construction phase and warrants exploration of other flavors of competitive sorting algorithms proposed for the GPU platform (sample sort [11], GPUMemSort [12]).
- Other error correction paradigms such as suffix tree/array based methods and MSA based methods exist in addition to the k-spectrum based approach. It would be interesting to implement these other paradigms on the GPU platform.
- It would be worthwhile to explore how constant and texture memories, which are backed by on-chip caches, can be made use of by the proposed framework for enhancing the performance further.

References

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

[2] 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.

[3] X. Yang, S. P. Chockalingam, and S. Aluru, "A survey of error-correction methods for next-generation sequencing," *Briefings in bioinformatics*, vol. 14, no. 1, pp. 56–66, 2013.

[4] X. Yang, K. S. Dorman, and S. Aluru, "Reptile: representative tiling for short read error correction," *Bioinformatics*, vol. 26, no. 20, pp. 2526–2533, 2010.

[5] L. Ilie, F. Fazayeli, and S. Ilie, "Hitec: accurate error correction in high-throughput sequencing data," *Bioinformatics*, vol. 27, no. 3, pp. 295–302, 2011.

[6] L. Salmela and J. Schröder, "Correcting errors in short reads by multiple alignments," *Bioinformatics*, vol. 27, no. 11, pp. 1455–1461, 2011.

[7] W.-C. Kao, A. H. Chan, and Y. S. Song, "Echo: a reference-free short-read error correction algorithm," *Genome research*, vol. 21, no. 7, pp. 1181–1192, 2011.

[8] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious b-trees," in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, 2000, pp. 399–409.

[9] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, 2013, in press.

[10] I. Roy and S. Aluru, "Finding motifs in biological sequences using the micron automata processor," *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2014, under review.

[11] N. Leischner, V. Osipov, and P. Sanders, "Gpu sample sort," in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–10.

[12] Y. Ye, Z. Du, D. A. Bader, Q. Yang, and W. Huo, "Gpumemsort: A high performance graphic co-processors sorting algorithm for large scale in-memory data," *GSTF International Journal on Computing*, vol. 1, no. 2, pp. 23–28, 2011.