GPU BASED EULERIAN ASSEMBLY OF GENOMES

by

Syed Faraz Mahmood

A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Science

Committee:

Huzefa R	Dr. Huzefa Rangwala, Thesis Director
Sanjeev Setia	Dr. Sanjeev Setia, Committee Member
Pearl Wang	Dr. Pearl Wang, Committee Member
Sanjeev Setia	Dr. Sanjeev Setia, Department Chair
Kenneth S Ball	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: 12/04/12

Fall Semester 2012
George Mason University
Fairfax, VA

GPU based Eulerian Assembly of Genomes

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Syed Faraz Mahmood
Bachelor of Science
National University of Emerging Sciences, 2004

Director: Dr. Huzefa Rangwala, Professor
Department of Computer Science

Fall Semester 2012
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to Prophet Muhammad (May Allah pray on him and grant him peace) and his Companions, who remained a constant guiding factor for all aspects of my life.

Acknowledgments

'*All praise unto Allah, the Lord of all the worlds*' [Quran 1:1]. I am thankful to Allah Almighty for His boundless blessings which I have enjoyed throughout my life. It was nothing but His grace and will that I undertook this daunting task and accomplished it.

I am grateful to my thesis advisor, Dr. Huzefa Rangwala who not only directed and supervised my research activities, but as a mentor, helped me nurture them as well. I found prompt feed backs and timely advices, an indispensable aspect of his tutelage. The culmination of my research efforts in the form of this thesis is largely attributed to his constant motivation, constructive critique and dedicated mentorship.

I would like to extend special thanks to Dr. Sanjeev Setia and Dr. Pearl Wang for the time and effort they spent reviewing my thesis. The numerous invaluable discussion that I had with them, equipped me with essential knowledge to establish a sound understanding in the area Parallel and Distributed Computing.

I express my deepest gratitude for my parents, siblings and wife who filled my life with warmth and color, making it full of everlasting memories. Their constant support and prayers have always been an asset of my life.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Abstract

GPU BASED EULERIAN ASSEMBLY OF GENOMES

Syed Faraz Mahmood

George Mason University, 2012

Thesis Director: Dr. Huzefa Rangwala

Advances in sequencing technologies have revolutionized the field of genomics by providing cost effective and high throughput solutions. In this paper, we develop a parallel sequence assembler implemented on general purpose graphic processor units (GPUs). Our work was largely motivated by a growing need in the genomic community for sequence assemblers and increasing use of GPUs for general purpose computing applications. We investigated the implementation challenges, and possible solutions for a data parallel approach for sequence assembly. We implemented an Eulerian-based sequence assembler (GPU-Euler) on the nVidia GPUs using the CUDA programming interface. GPU-Euler was benchmarked on three bacterial genomes using input reads representing the new generation of sequencing approaches. Our empirical evaluation showed that GPU-Euler produced lower run times, and comparable performance in terms of contig length statistics to other serial assemblers. We were able to demonstrate the promise of using GPUs for genome assembly, a computationally intensive task.

An error correction step was also incorporated into GPU-Euler to be able to process reads containing some errors. Error correction output was benchmarked on simulated read on three bacterial genomes with different read length.

Chapter 1: Introduction

Knowing the entire DNA sequence of an organism is an essential step towards developing systematic approaches for altering its function. It also provide better insights into the evolutionary relations among different species. In the last few years, we have seen several new, high-throughput and cost-effective sequencing technologies that produce reads of length varying from 36 base pairs (bp) to 500 base pairs (bp). Technological advancement in sequencing techniques has also increased the volume of data produced during a sequencing process. Sequence assembly algorithms stitch together short fragment reads and put them in order to get long contiguous stretches of the genome with few gaps.

Several assembly methods have been developed for the traditional shotgun sequencing and new sequencing technologies. Examples include, greedy approaches like VCAKE [1], graph oriented approach that find Eulerian tours [2], and bi-directed string graph representations [3]. ABySS [4] is one of the first distributed memory assembler. It has a unique representation for the de-Bruijn graph that allows for ease of distribution across multiple compute processors as well as concurrency in operations. Jackson et al. [5, 6] proposed a parallel implementation for bi-directed string graph assembly on large number of processors available on supercomputers like the IBM Blue Gene /L.

In this work, we develop a GPU-based sequence assembler, referred to as GPU Euler. Specifically, we follow the Eulerian path based approach that was developed for Euler [7]. Our method is motivated by the current advances in multi-core technologies and the use of graphic processor units (GPUs) in several computing applications. In this thesis, we investigated the effectiveness and feasibility of graph-based sequence assembly models on GPUs using the CUDA programming interface. nVidia GPUs along with CUDA provides massive data parallelism, that is easy to use and can be considered cost-effective in comparison to loosely coupled Beowulf clusters.

GPU Euler is benchmarked on three previously assembled genomes: (i) Campylobacter Jejuni, (ii) Neisseria Meningitidis and (iii) Lactococcus Lactis. We simulated three sets of reads for each genome with read lengths equal to 36, 50 and 256 base pairs (bps). The scope of this work was to develop an assembler that would exploit the GPU computing resources effectively. As such, we were focused in improving the run times of different phases of the algorithm. We compare the performance (run-time, contig accuracy and length statistics) of GPU-Euler to a previously developed assembler, EulerSR [8]. A detailed discussion of the implementation, experiments and results has been presented in chapter 3.

We also extended our GPU based sequence assembler to perform error correction on GPU. Our implementation is adapted from the Spectral Alignment that was proposed as part of Euler [7]. Spectral Alignment belongs to embarrassingly parallel class of problems and appeared to be a good candidate for GPU based applications. We benchmarked our error correction implementation on the same genomes with simulated set of error containing reads. We collected profiling statistics and accuracy measurements for our error correction procedure in order to analyze the correctness of the implementation. The experiments, results and discussion pertaining to error correction is presented in chapter 4.

1.1 Motivation

This work is motivated by a number of factors, most important of which is the cost effectiveness. The project aims to find a lower cost and reliable solution for assembling genomes. There are existing tools for genome assembly but most of them e.g Abyss[4, 9] require a cluster of machines to achieve maximum throughput. A GPU capable of CUDA capabilities typically range between $200-$500, allows a common man's personal computer to be utilized as a massively parallel computing device.

Another important aspect is the performance of the solution. Current generation assembly tools usually use MPI based communication pattern, which involves network latency. An on-chip array of processor working on a solution, removes the network latency. However,

it may introduce contention and synchronization latency issues.

Last but not the least, with current pace of advancement in microprocessor technology, we are moving towards an era of multi-core programming. It is inevitable to have solutions that can fully exploit the potential of the emerging paradigm of parallel computing.

1.2 Problem Statement

GPU based data parallel applications are proving to be excellent cost effective alternatives to large scale data processing solutions. GPU based sequence assemblers will be helpful in providing a handy tool to analyze the genome assembly at a reduce cost. This thesis aims to develop a data parallel GPU-based de-novo sequence assembler. Evaluation will be performed to assess the performance and correctness of the implementation. Moreover, it will also provide a thorough assessment on the challenges presented by the General Purpose GPU Computing platform for implementation of graph centric algorithms.

1.3 Contributions

- A GPU based parallel Eulerian path sequence assembler ,
- GPU based Error Correction for eulerian path sequence assembler,
- Publication : *"GPU-Euler: Sequence Assembly Using GPGPU"* [10], published by IEEE Banff, Canada.

1.4 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 provides a thorough review of different sequence assemblers, parallel computing models and CUDA architecture. Chapter 3 discusses GPU based implementation of parallel eulerian assembly construction. Chapters 4 presents a GPU based approach for error correction. Chapter 5 describes several directions for future work and provides concluding remarks.

Chapter 2: Literature Review

2.1 Introduction

This chapter provides an explanation of important concepts concerning sequence assembly and parallel computation along with a literature survey of the recent research results.

2.2 Sequence Assembly

Genome of an organism serves as a biological blue print, that documents every aspect of that organism from physical appearance to behaviours and from diseases to life span [11]. Genome sequences comprise entirely of Deoxy Ribonucleic Acid (DNA) which is present in the nucleus of every cell of a living organism in the form of large complex compounds. There are four nucleotides : Adenine, Guanine, Cytosine, Thymine (Table 2.1) which are the basic building block of a DNA molecule. For the purpose of computational analysis, these nucleotides are often referred to as A, G, C or T. A DNA molecule comprises of two parallel strands consisting of these nucleotides known as *Forward* and *Reverse* strands. The nucleotides exhibit a complementary relationship with other nucleotides as given in Table 2.1, such that the two strands are held together through bonding between complementary nucleotides. For instance, *Adenine* on one strand binds to *Thymine* on the opposite strands and similarly *Cytosine* binds to *Guanine*. This bonding give rise to the well-known double helical structure for DNA. Given a sequence whether forward or reverse, its complement can be easily computed by reversing the sequencing and substituting the bases with their complements. Most computational tools therefore, deals only with one sequence only and generates the complementary one if required.

Genome technologies do not produce complete genomes, instead they only generate a

Table 2.1: Nucleotides and their Complements

Nucleotide	Complement
Adenine	Thymine
Cytosine	Guanine
Guanine	Cytosine
Thymine	Adenine

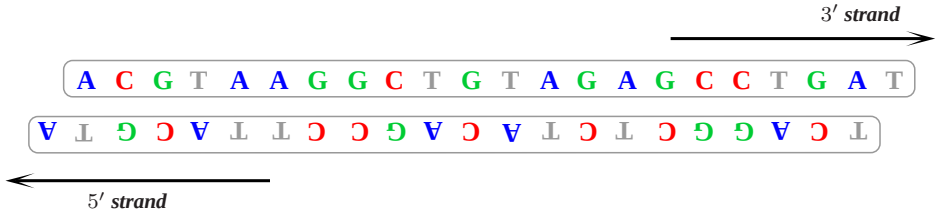

Figure 2.1: Double helix structure of DNA with forward and reverse strands

small sub-sequence of the whole genome. These smaller sub-sequences are known as **reads**. The past few years have seen rapid development in new, high-throughput and cost effective sequencing technologies; Roche 454, Illumina Genome Analyzer 2 (GA2) and the ABI SoLID platform in addition to the well established Sanger sequencing protocol. These approaches vary in their output, cost, throughput and errors produced. All approaches rely on shotgun sequencing [12], where the genome is randomly sheared into many small sub-sequences or reads (see Figure 2.2).

2.2.1 Sequence Assembly

The process of combining sequence reads after sequencing a genome, in order to reconstruct the source genome is called the **sequence assembly**.

Sanger [13], produces longer sequence reads, of size 750 to 1000 base pairs (bps) whereas the next generation sequencing (NGS) technologies produce reads that are shorter from 36 to 500 bps. The volume of data produced by NGS technologies demands a robust solution to the data management, assembly, and the development of derived information. Pop [14]

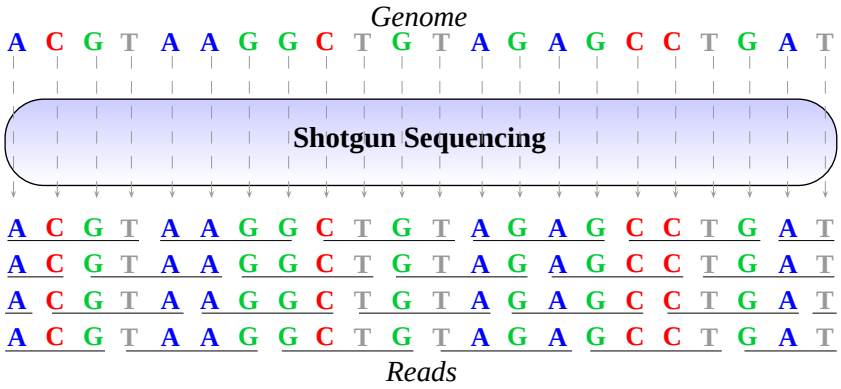

Figure 2.2: Shotgun Sequencing

and Myers et al. [15] provide a detailed review of the computational challenges involved with sequence assembly, along with a study of the widely used approaches.

2.2.2 Sequence Assembly Algorithms

De novo assembly algorithms stitch together short fragment reads and put them in order to get long contiguous DNA fragments called **contigs** [16]. Contigs are further extended to get super-contigs and finally placed in order, to get the assembled genome. The approaches for de novo sequence assembly can be grouped into three categories: (i) greedy [1], (ii) overlap-layout-consensus [17] (OLC) and (iii) eulerian-based approaches.

Greedy assemblers, follow an iterative approach where at each step, the reads (or contigs) that have the longest possible overlap with other reads are extended. An effective indexing mechanism is used to accelerate the discovery of the reads to be used for further extension to produce longer contigs. For assembling NGS data, greedy assembly algorithms, like SSAKE [18], SHARCGS [19], QSRA [20] and VCAKE [1] have been developed. Due to their greedy nature, these algorithms produce several mis-assemblies due to repeat regions within the genomes.

The OLC approach finds potentially overlapping reads between fragments by computing pairwise alignments between the reads (overlap). The overlap between the reads can be

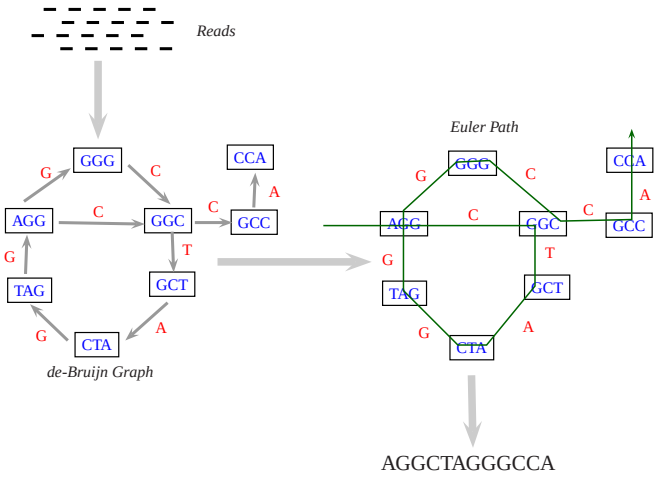

Figure 2.3: Eulerian Assembly

modeled using edges of a graph with the reads as vertices (layout). Determining a Hamiltonian path, i.e., a valid path that visits every vertex exactly once will lead to a sequence assembly. However, finding the paths in presence of repeats leads to NP class of problems, and as such the DNA sequence (consensus) is derived using several heuristics as illustrated in methods like Celera [21], Arachne [22] and EDENA [17].

The Eulerian based de novo methods have always been widely used and were inspired by the sequencing-by-hybridization approach [7, 23]. These algorithms represent each read by its set of *tuples* (smaller sub-sequences of some fixed length k) and construct a de-Bruijn graph. A *de-Bruijn* graph is a directed graph where vertices are tuples, and there exists an edge between two vertices if they are from the same read and have an overlapping sub-sequence of length $(k-1)$ between them. Finding the Eulerian path or tour, where each edge in the de-Bruijn graph will be visited exactly once will lead to the sequence assembly solution. Before performing the Eulerian tour, these approaches use different heuristics to remove erroneous nodes and edges from the de-Bruijn graph, that are created due to sequencing errors and repeat regions within the genome.

Myers presented another graph oriented approach based on the notion of bi-directed string graph[3]. A *bi-directed string graph* has direction associated with both end points

of an edge produced by modeling the forward and reverse orientation of sequence reads. The Eulerian-tour of such a graph enforces additional constraints that leads to improved accuracy and length of produced sequence contigs.

Euler-SR [8], Velvet [24], SHORTY [25], ALLPATHS [26] and ABYSS [4] are examples of the different Eulerian-based approaches, developed for NGS read data. These algorithms differ on the heuristics that they employ to perform the graph simplification, and on the data structures used to construct the de-Bruijn graph for modeling the reads.

2.2.3 Parallel Sequence Assemblers

ABySS [4] is one of the first distributed memory de novo assembler. It has a unique representation for the de Bruijn graph that allows for ease of distribution across multiple compute processors as well as concurrency in operations. The location of a specific k-mer within the reads can be computed from the sequence of the reads, and the adjacency information for a k-mer is stored in a compact fashion that is independent of the location. Jackson et al. [5,6] proposed a parallel implementation for bi-directed string graph assembly on large number of processors available on supercomputers like the IBM Blue Gene/L.

2.2.4 Error Correction

With the advancement in sequencing technology, the sequencing of genome is getting less error prone, but still sequencing error are still an important consideration during the assembly process. Sequencing tools try to overcome the error in their output by generating multiple readings (known as coverage) of same location. This improves the confidence level that an incorrect base call will be outnumbered by the correct one. Different assembly algorithms devise different techniques to detect erroneous read. Some assemblers uses consensus to determine error that are expected to be highly infrequent at any given locations. Pevzner et. al [2] proposed a pre-processing phase to fix error prior to sequence assembly as part of EulerSR. The idea is based on the observation that for a high coverage genome, tuples from reads would appear more frequently. If the read contains an incorrect base than the tuple

containing that particular base would result in less frequent tuples. The error correction routine then tries to introduce modification for that read such that the tuples from the modified read would cause a reduction in number of unique tuples from the whole read set (known as read spectrum) and all its tuples would appear as more frequent tuples.

2.3 Compute Unified Device Architecture.

Graphic processing units (GPUs) have long been serving the need of parallel computation due to the very nature of the graphics applications which benefit from ever increasing parallelism in the hardware. For the past few years, efforts have been conducted to tap the potential of GPU's parallel computation capability so that it can be utilized for general purpose computing.

Compute Unified Device Architecture (CUDA) is an initiative from nVidia that provides programming interface to utilize the compute capabilities of GPUs. CUDA SDK provides a compiler front end implemented as an extension of C language, augmented with several device-specific constructs. It contains a set of run-time libraries that provide an API for device management. CUDA-enabled GPU devices provide parallel thread execution environment known as PTX, and can be viewed as multiple threads concurrently executing the same piece of code called kernel. This form of architecture, can be mapped to the single program multiple data (SPMD) style of parallel computation. The logical view of CUDA device groups a set of thread into blocks and set of blocks into grids. Physical view of CUDA consists of a number of streaming multiprocessors (SMs), having a set of scalar processors (SP). The number of SPs on SMs varies depending upon the hardware revision. CUDA threads are scheduled to run on these SPs by a CUDA thread scheduler. The actual number of threads scheduled to run concurrently depends on the resources available to SMs.

GPU thread manager schedules the cores and manages the threads executing on the SMs. Memory is a key resource in CUDA applications and its efficient utilization is the basis of major performance improvements. There are different types of memory available to the CUDA threads, which vary in terms of their presence on and off the chip, latency and

Table 2.2: CUDA Memory Hierarchy

Memory Type	Latency	Location	Accessibility
Global Memory	High	off-chip	Host, All threads
Shared Memory	Low	on-chip	Shared within block
Register File	Low	on-chip	Local to thread
Texture Memory	Low	on-chip	All threads
Constant Memory	Low	on-chip	(read only) Host, All threads

accessibility to threads or to blocks. A summary of memory hierarchy present on CUDA devices is presented in Table 2.2.

One major advantage of CUDA lies in the abstraction of computing model from the hardware implementation, which decouples the code from the underlying hardware. This allows the developer to focus more on the algorithmic aspect rather than the device (GPU) specific implementation. CUDA provides a virtual instruction set, augmented with software interfaces as the CUDA programming API to exploit the GPU devices. Applications developed against these specifications will be able to run across different compliant GPU devices with CUDA middle-ware providing the logical to physical thread mapping. Developers can model their solution by decomposing their problem as multiple sub-tasks. Each sub-task can be worked upon by a group of threads in a block. The concurrent execution of sub-tasks is handled by the GPU.

2.4 Parallel Computing Models

Developing parallel algorithms requires some assumptions regarding the underlying computing model. Over the years, researchers have proposed different models considering different aspects of parallel computation. These models formulate how different components of a parallel system would work to perform computation at hand. How many processors would be in the system? What is inter-connect network and communication pattern among the processors? How the memory would be accessed by the processors? Whether it will be shared among the processors? Whether the model support atomicity and synchronization

between reads and writes from different processor on a single memory location? Each computing models has its advantages and disadvantages. For instance, computing model with concurrent read / write usually have simpler algorithms but the hardware implementation often prove to be quite daunting.

2.4.1 PRAM Model

PRAM - Parallel Random Access Machine is a theoretical model for parallel machines where the number of processor required to execute the algorithm is defined by the algorithm itself. All processors are assumed to be synchronous for each execution of the instructions. PRAM model incorporates a shared memory model and all processors are assumed to have equal access time for all the memory locations. Based on the handling of concurrent read and concurrent write, the PRAM model is further divided into 4 categories:

- CRCW (concurrent read concurrent write),
- CREW (concurrent read exclusive write),
- ERCW (exclusive read concurrent write)
- EREW (exclusive read exclusive write)

CRCW is considered to be the strongest model, while EREW is the weakest. Strength of a particular model is based on the ability of concurrent access, which allows more complex algorithm to be easily implemented. Concurrent writes are resolved by defining different strategies e.g. first writer succeed, last writer succeed, lower numbered writer succeed or reduction operation etc. Generally CRCW algorithms can be converted to CREW model with $O(\log)$ overhead for $O(n)$ items to resolve write conflicts. For GPU Euler, PRAM model is opted since it provides closer approximation for CUDA platform.

2.4.2 Flynn's Taxonomy

Flynn Proposed a classification of computers based on the number of data and instruction stream available [27]. Depending on the number of streams, a computing device can be

classified as one of the four classes.

SISD

Single Instruction, Single Data (SISD) machines have one instruction stream which operates on single data stream. Everyday computer systems are a common example of SISD.

SIMD

Single Instruction, Multiple Data (SIMD) machines have one instruction stream and multiple data streams. During a single fetch-decode-execute cycle, single instruction is fetched and executed on data items from the available data stream. This class of machine represents classic vector processing machines.

MISD

Multiple Instruction, Single Data (MIMD) machines have multiple instruction streams but single data stream. Multiple fetch units work in tandem, to retrieve instructions from their respective streams and operate on the data. This class of machines are not common.

MIMD

Multiple Instruction, Multiple Data (MIMD) machines have multiple instruction and multiple data streams. Each instruction stream can operate on the data from various stream.

CUDA devices offer SIMT (Single Instruction, Multiple Thread) model for parallel computation which resembles closely to the SIMD class. SIMT emphasize the presence of multiple ALU that are utilized by each thread to execute a certain instruction. Moreover, SIMT differs from SIMD with respect to the execution of conditional instructions in a sense that CUDA threads can diverge and takes different paths when they execute conditional instructions.

2.5 Parallel Graph Algorithms

Designing efficient graph algorithms for parallel computing architecture has always been challenging. Computational problems dealing with graphs are inherently difficult to decompose into balanced sub-problems, and the speedup achieved is usually not linear [28–31].

2.5.1 Euler Tour Construction

The "Euler tour" construction problem for a connected graph is defined as the traversal of a graph by visiting every edge exactly once. Linear time algorithm with respect to the number of edges, exist for the sequential algorithm. Makki [32] proposed a $O(|E| + |V|)$ algorithm for distributed memory model using a modification of the original, Fleury's sequential algorithm. It works by simulating the Fleury's algorithm on the vertices distributed among the nodes, and each node tries to identify the successive edge.

Awerbuch et al. [33] proposed a $O(\log n)$ parallel time algorithm for a concurrent-read and concurrent-write, CRCW model which requires $O(|E|)$ processors. In this approach, the concurrent writes do not require any specific ordering of the write operations, which makes it similar and applicable on CUDA-enabled GPU platforms. Concurrent writes can be achieved by using atomic operations.

In our work, Awerbuch's approach has been opted for implementing the Eulerian tour construction. This approach for determining the Euler Tour, first constructs a successor graph by defining a successor relationship between edges. In the successor graph, the vertices corresponds to the edges of original de-Bruijn graph and an edge represents the end points that are related to each other by successor function. Further, the connected components of the successor graph identifies the circuits in the graph. Two circuits will be related to each other if they have edges incident on the same vertex. This relation is represented in the form of circuit graph. A spanning tree of this circuit graph will yield a path connecting each circuit, which in turn represents edges with their successors. Swapping the successors of the edges of two circuits at the same vertex (identified by edge of the spanning tree), will result in a Euler tour. To lower the number of edges in the circuit graph, the algorithm

extracts connected sub-graphs from the circuit graphs. GPU Euler uses a parallel connected component algorithm as part of parallel Eulerian tour algorithm execution.

As part of the Euler tour construction, a spanning forest is computed from the circuit graph. The edges in the spanning forest will identify the edges of the de-Bruijn graph that can exchange their successors, yielding to the final euler tour. The different methods used for identifying connected components (see below) can also be used for identification of spanning forests. There are several efficient parallel spanning forest algorithms available for different computing models [34]. In this implementation, Kruskal's Spanning tree algorithm as implemented in BOOST graph library [35], is selected to calculate the spanning tree on CPU.

2.5.2 Connected Components

For a given graph, finding connected components involves partitioning the graph such that the vertices in the partitioned sub-graph (subset) are connected to each other through some path and there does not exist any path between vertices of different sub-graphs.

Various approaches available for connected components are equally useful in finding the spanning forest. In fact most of the algorithms uses a connected component construction for constructing the spanning forest.

Johnson-Metaxas [36] proposed an $O\left(log^{\frac{3}{2}}n\right)$ parallel time algorithm for CREW PRAM model. Awerbuch and Shiloach[34] also proposed a CRCW PRAM algorithm which was based on Shiloach-Vishkin's [37] approach. Greiner [28] compared different parallel algorithms for finding connected components, but the study was limited to the CRCW model.

Shiloach-Vishkin [37] proposed a $O(\log n)$ parallel time algorithm to find the connected component of graph using $n + 2m$ processors where $n = |V|$ and $m = |E|$ for CRCW PRAM model. This algorithm finds the connected component by iterating over four steps. For every vertex, a root pointer is maintained which points to the lowered numbered vertex within the connected component. The algorithm starts with each vertex as a component, with its root pointer pointing to itself. During the execution, the root pointer is repeatedly

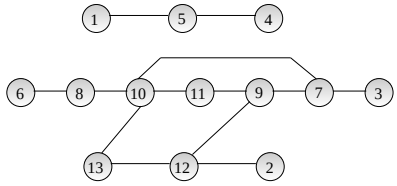

Figure 2.4: Connected Component Example

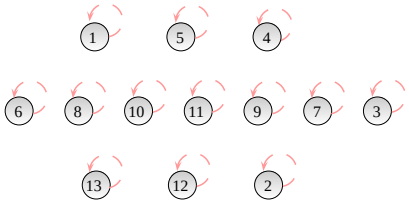

Figure 2.5: Connected Component Example Initialization

updated. Update steps correspond to the merging of components to form larger components. Within the iteration, the first step updates all the root pointers, in case the root vertex itself has been updated with a new root during previous iteration. The next step requires all vertices to examine their neighbours and update their root pointers, if the neighbour has root pointer with smaller vertex number. Root vertices of the component that remained unchanged in the first step, now try updating their root pointers with neighbouring lowered number root vertices. All vertices then perform a short-cutting step to update their root pointer with their root's root pointer (grand-parent). These steps are performed iteratively till there are no more updates to the root pointers.

Figure 2.4 is the example used by Shiloach-Vishkin [37]. There are two component formed with 13 vertices and 13 edges. Figure 2.5 shows the initialization step where root pointers for all vertices point to themselves and algorithm start the first iteration. Since there is no prior updates to root pointer performed, the algorithm moves to second step Figure 2.6, examining the neighbours and hooks up to smaller roots e.g vertices 5 now has 4 as its root pointer. In step 3 and 4, the algorithm performs short cutting, completing

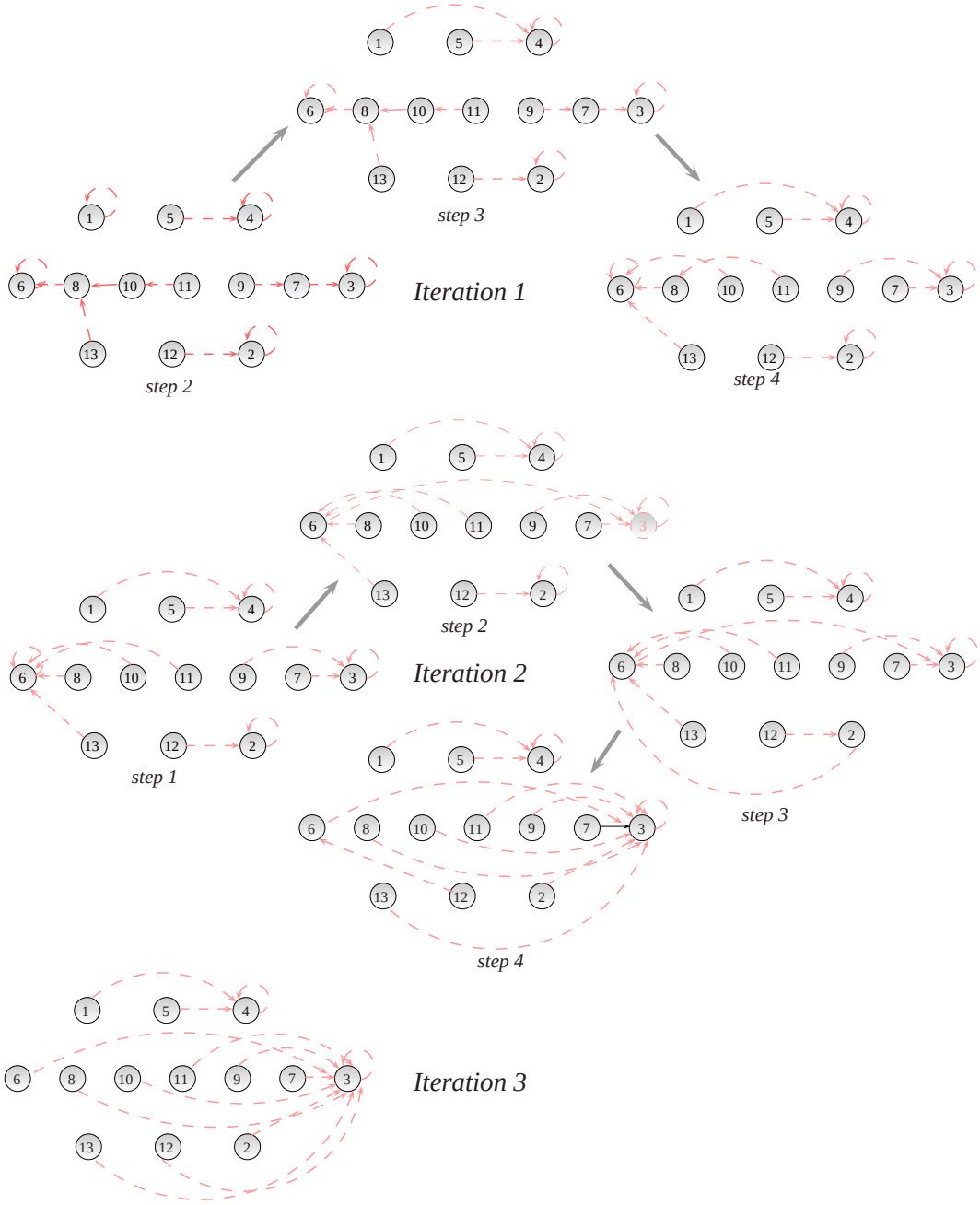

Figure 2.6: Iterations for Connected Component Example

the first iteration. Since the vertices have updates applied to their roots, the algorithm continues to the second iteration and updates the vertices to get the new root information. At the end of iteration 3 , all vertices now point to either of the roots of two components (vertex 4 and 3).

2.5.3 Parallel Spanning Tree Algorithms

Bader et. al. [38] proposed a parallel tree algorithm for symmetric multiprocessors which gives an expected running time that scales linearly with number of processors. This algorithm proceeds in two phases. In the first phase, a single processor generates a stub spanning tree using p steps. Then, in the next phase this stub spanning is equally distributed among the p processors which continue to work on the sub-graph of the original graph. Finally each sub spanning tree is joined using the stub to form the final spanning tree. This approach provides good performance for shared memory architecture with both sequential and parallel processors working simultaneously on the problem.

Awerbuch-Shiloach[34] proposed an $O(\log n)$ parallel time algorithm using the parallel connected component construction algorithm proposed by Shiloach-Vishkin. The connected component algorithm needed a concurrent write approach, which was modified such that the lower ranked processor would have priority in case of a write conflict. Write conflicts represent the condition when different edges compete to be included in the spanning tree and only the least weighted one will be included at a step.

Chin et. al.[30] proposed an $O\left(\log^2 n\right)$ algorithm for computing the minimum spanning tree. The algorithm, first identifies the connected components and then grows the minimum spanning tree by incrementally determining the least weighted edge connecting the vertices within the same components. An adjacency matrix is used for representing the graph, while computing the minimum spanning tree.

Chapter 3: GPU based Sequence Assembly

3.1 Introduction

This chapter discusses the initial implementation of GPU Euler, describing the basic structure and assumptions of the application. An experimental assessment is also included to evaluate different aspects of the implementation.

3.2 Workflow

The GPU Euler workflow is composed of following three steps:

1. de-Bruijn graph construction.
2. Euler tour construction
3. Contigs generation.

In the first step, the de-Bruijn graph is constructed by extracting tuples from the input reads and calculating vertex and edge information. We use tuples of length l and k (where $l = k + 1$) to represent edges and vertices, respectively. Tuples of length l are referred to as ***l-mers*** while tuples of length k are referred to as ***k-mers*** . During the next step, an Euler tour is constructed and de-Bruijn graph is annotated with the tour information. In the final step, a walk on the graph is performed using the annotations which generates the contigs. Figure 3.1 represents a schematic work flow of the complete process highlighting each step. The output of each phase is labelled on the transition edges. At this moment, GPU Euler does not employ any error correction heuristics.

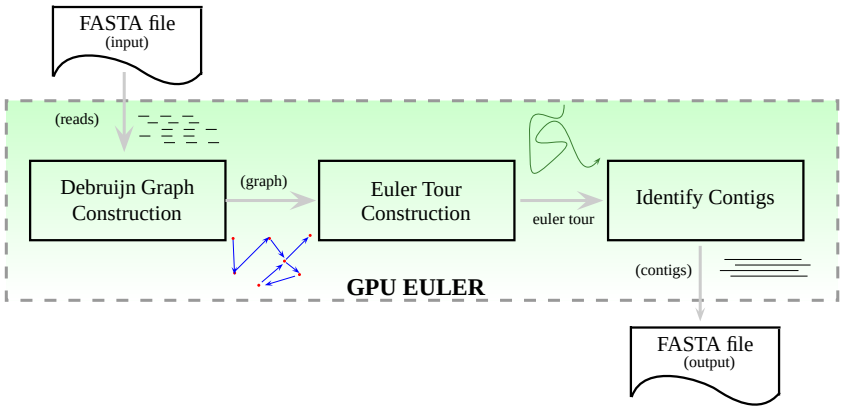

Figure 3.1: GPU Euler Workflow

3.3 Data Structure

The ever increasing accessibility of massive computing power has transformed the computed bound problems into I/O bound problems. Memory bandwidth and architecture play an important role in CUDA based applications. A desirable solution for handling massive amount of data often associates the design of data structure with the architecture specific recommendations and best practices. CUDA memory architecture also guides various aspects of the problem decomposition and data structure design. Some of the CUDA features are described below which have heavily influenced our data representation and algorithm.

3.3.1 Design Considerations

The data structure design of GPU Euler has been influenced by the following aspects of CUDA platform in order order to achieve optimal resource utilization.

Consecutive Thread Access

The memory manager on CUDA devices employs techniques to optimize the access to GPU memory by combining reads originated from threads of a block. When successive threads in a thread block issue a read from successive memory locations, i.e, thread i access memory

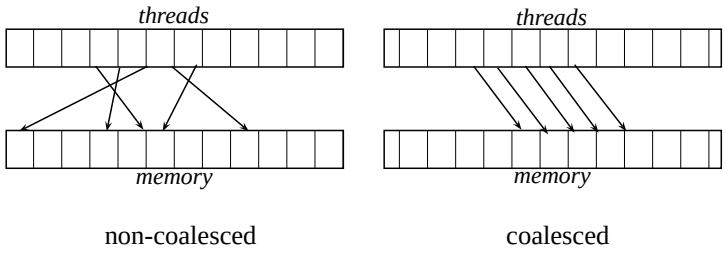

Figure 3.2: Memory access pattern of threads

address j and thread $i + 1$ access memory address $j + 1$ and so on , the are combined into single fetch, thus improving memory bandwidth. A major pre-requisite to exploit this optimization dictates that the data residing in the GPU memory must be aligned at word boundaries. Furthermore, algorithm implementations often required to revise their data access pattern to use a sequential pattern rather than a random one.

Pointers Free Representation

Although the mode of communication between GPU memory and System memory is through memory mapped I/O, it only maps a small portion of system memory on to GPU memory but not the other way around. In GPU based computations, special APIs are used to move large amount of data from System to GPU memory and then back to system memory. Objects constructed on the GPU heap use an entirely different address space than the system heap, due to which object graphs becomes void when moved from one address to the other. Data structure based on dynamic memory allocation and pointer reference (Linked List, Graph etc) , therefore, can not be moved around easily and requires translation.

3.3.2 Data Structure Layout

It is often desired to strike a balance between these requirements. A compact representation might adversely affects the thread access pattern, often yielding a random access across threads. On the other hand, favouring thread access pattern might lead to data redundancy. One approach for data representation is to use *Structure of Arrays* (see Appendix A.3.3)

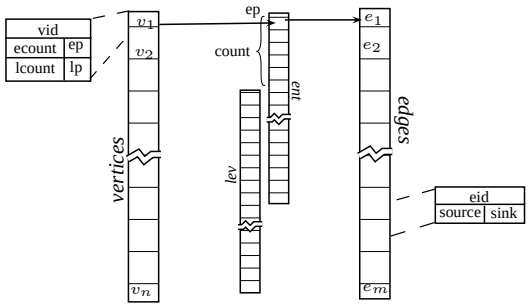

Figure 3.3: de-Bruijn graph data structure layout

instead of Array of Structures to represent large number of data objects.

Our de-Bruijn graph G is defined as a $G = \{\mathcal{V}, \mathcal{E}, lev, ent\}$ where

- $\mathcal{V}$: list of vertices
- $\mathcal{E}$: list of edges
- *lev*: list of outgoing edge indexes for each vertex
- *ent*: list of incoming edge indexes for each vertex

Each entry in *lev* and *ent* is an index to $\mathcal{E}$. The index lists *lev* and *ent* are simply concatenated list of indexes for all the vertices. Thus, instead of keeping the list of edges within each vertex, a vertex only contains two indexes *ep* and *lp* for storing starting index into *ent* and *lev* respectively. A vertex also keeps respective counts i.e. *ecount* and *lcount* for entering and leaving edges respectively. This graph structure is computed during program execution. *lev* and *ent* thus provide an alternative representation for memory references (pointers).

Following this scheme, each edge and vertex will be stored only once. On the other hand, storing edge list out side the vertex allows the vertex list to be aligned in the memory facilitating a sequential access pattern, as these list can be of arbitrary lengths. Figure 3.3 is a pictorial representation of the data structure used for de-Bruijn graph.

Table 3.1: Nucleotides Encoding

Nucleotide	Abbreviation	$Base_4$ Encoding	$Base_2$ Encoding
Adenine	A	0	00
Cytosine	C	1	01
Guanine	G	2	10
Thymine	T	3	11

The proposed data structure provides a compact representation of the whole de-Bruijn graph as all vertices and edges along with their associated data are only stored once and accessed through indices. The data structure also provides an alternative solution for objects models which are dynamic in nature (e.g., graphs) since CUDA device do not allow memory allocation inside an executing kernel.

GPU Euler excessively utilizes the pointer free representation by exploiting element's index as a pseudo-pointer. Vertices and edges are linked to each other using their respective indices.

Graph problems usually don't facilitate sequential data access pattern. Especially during de-Bruijn graph construction and euler tour construction, there exists an entirely random data access pattern which can be predicted ahead even after analyzing the data.

3.3.3 Tuple Encoding

The input to the graph construction phase consists of a set of reads from the FASTA file. A genome is composed of four nucleotides as listed in Table 3.1. These nucleotides are often referred to with their initials for the purpose computational analysis. Formally, a genome can be defined as a string S over alphabet $\Sigma = \{A, C, G, T\}$ of arbitrary length . A *read*, thus, is a sub-string of S. Since there are only four characters in our alphabet, we can easily represent them as $base_4$ digits requiring only 2 bits per nucleotide. This will lead to a 3/4 reduction in the memory if each nucleotide were to be represented as an 8 bit ASCII character. Using this scheme, a tuple of length k can be represented as a sequence of $2 * k$ bits. Table 3.1 also shows the encoding used to represent each nucleotide as $Base_4$ and

Table 3.2: Tuple Encoding Example

Tuple Length	Encoding
5	A A G G C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 10 10 01
18	T C T C G G A A C C C A C A C G C C 00 00 00 00 00 00 00 00 00 00 00 00 00 11 01 11 01 10 10 00 00 01 01 01 00 01 00 01 10 01 01
24	A T C C C A T T C T C T G G G C G C C G C G A A 00 00 00 00 00 00 00 00 00 11 01 01 01 00 11 11 01 11 01 11 10 10 10 01 10 01 01 10 01 10 00 00
32	G C G C T A T C G T C G T A C G A T C G T A C G T A C G T A A C 10 01 10 01 11 00 11 01 10 11 01 10 11 00 01 10 00 11 01 10 11 00 01 10 11 00 01 10 11 00 00 01

Base_2 numbers.

With 2-bit encoding per base, each tuple is mapped to a unique 64-bit value (GPU Euler therefore only supports $k \leq 32$). This scheme allows easy computation of tuple's neighbours by simple bit-wise operations (Shift, And, OR). A neighbour for a tuple of length k x, is one that shares either the prefix or suffix of length $k-1$ with x.

3.4 de-Bruijn Graph Construction

de-Bruijn graph construction has multiple steps. It starts with reading the input set of reads from the FASTA file. Tuples of length l(l-mers) are extracted and encoded in an space efficient manner for later stages. After collecting the l-mers, CUDA kernel are invoked to construct the graph. These are described in the following subsections.

3.4.1 Tuple Extraction and Encoding

During the initial step, input file is processed by splitting the reads into batches. The size of batch depends on the length of reads and available global memory on GPU. In a read of length r, $r-l+1$ l-mers can be extracted. A naive CUDA kernel (Algorithm 3.1) can be designed to process these reads to extract l-mers. This kernel would operate on a batch of n reads with n blocks, and each block having $r-l+1$ threads to extract $r-l+1$ l-mers

in parallel by reading the read data from GPU memory.

Algorithm 3.1 Tuple extraction and encoding (simple)

Input:
R : Read Set
l : tuple length
Output:
T : Set of tuples of length l

```
1:  procedure EXTRACT-TUPLES(R,l)
2:      T ← ∅
3:      for all R_i ∈ R do
4:          for all thread_j ∈ block_i do                ▷ Each block process one read
5:              t_{i,j} ← ∅
6:              for k := 1 → l do
7:                  base ← R_{i,j+k}                     ▷ get next base
8:                  t_{i,j} ← APPEND(t_{i,j}, base)      ▷ Shift Left and OR with next base
9:              end for
10:             T ← T ∪ t_{i,j}
11:         end for
12:     end for
13:     return T
14: end procedure
```

This is clearly an inefficient approach as it would incur a lot of over-head. Same memory location would be accessed multiple times from different thread. There will be a total of $l \times (r - l + 1)$ read operations that will be executed if Algorithm 3.1 is used, with $(l \times (r - l + 1) - r)$ number of redundant read operations.

In order to avoid the redundant reads and to reduce the memory latency, GPU Euler exploits low latency shared memory as a cache. Algorithm 3.2 describes a technique where instead of computing each tuple individually by accessing global memory directly, all threads in a block work in tandem to fetch the read in parallel (line 5) , storing it in shared memory and proceeds on to compute the tuple using shared memory (line 9). This would lead to r read operations performed in parallel on global memory and $l \times (r - l + 1)$ read operations on shared memory. The improved kernel is invoked with r threads instead of $r - l + 1$ threads to simplify the process of copying data to shared memory. It starts with performing a parallel copy from GPU global memory to the block shared memory where each of r thread copies

a single base into the shared memory buffer. After copying, block level synchronization is perform so that the write to shared memory is available to rest of the threads in the same block. After synchronization, each thread proceeds to compute its l-mer. The $l-1$ extra threads are wrapped around to read from successive memory locations and their result is masked in the output.

Algorithm 3.2 Tuple extraction and encoding (with shared memory)

Input:
R : Read Set
l : tuple length
Output:
T : Set of tuples of length l

```
1:  procedure EXTRACT-TUPLES(R,l)
2:      T ← ∅
3:      for all R_i ∈ R do
4:          for all thread_j ∈ block_i do                ▷ Each block process one read
5:              shared[j] ← R_{i,j}
6:              Synchronize
7:              t_{i,j} ← ∅
8:              for k := 1 → l do
9:                  base ← shared[(j + k) mod l]        ▷ Get next based from shared memory
10:                 t_{i,j} ← APPEND(t_{i,j}, base)     ▷ Shift Left and OR with next base
11:             end for
12:             T ← T ∪ t_{i,j}
13:         end for
14:     end for
15:     return T
16: end procedure
```

3.4.2 Hash Table Construction

A tuple can be a vertex of a graph and it will have corresponding information like incoming edges, out going edges, etc. How should this kind of information be associated with a tuple to be accessible in a multi-threaded environment without afflicting space and performance penalties is a pertinent question that we address.

GPU Euler employs a slightly different mechanism for storage and retrieval of tuples and their information. Consider the fact that the smallest possible representation of all

tuples would be a list or array of tuples. Information associated with each tuple can be stored separately in associated arrays. So, for tuple t_i where i is the index of the tuple, then in all the associated arrays containing pieces of tuples' information, the i^{th} index would contain the attributes associated with t_i. At a more abstract level, the index itself becomes an attribute of the tuple. A hash table can be used to store this information which maps a tuple to an index for an associative array based retrieval. Tuples and their indices are only needed to be stored once as a key-value pair in the hash table.

GPU Euler uses a modified hash table implementation (Algorithm 3.3) of parallel hash table construction proposed by Alcantara et al. [39]. The original algorithm uses two level hashing scheme to construct an space efficient hash table. First, the keys are split into buckets using a hash function, then each bucket is further placed in 3 sub-tables using Cuckoo hashing with hash function for each sub-table. Cuckoo hashing is a simple hash table implementation where multiple hash functions are used to locate the keys. During insertion, if a key hashes to a location already occupied by an earlier key, then the earlier key is evicted to insert the new key and the evicted entry is re-inserted into the hash table by using another hash function. The hash table construction on CUDA, proceeds by placing keys into bucket and then each bucket is assigned to a CUDA block where each thread process one key and tries to insert the key in one of the 3 sub-tables, so that after construction, a maximum of three probe would be required to locate any key. The original algorithm depends on CUDA support for atomic operations on shared memory.

Instead of having 3 sub-tables with the 2nd level hashing, GPU Euler uses sorted buckets and sorts (line 16- 18)all the items in every bucket. This would imply that the retrieval operation will be implemented as a binary search operation on a single bucket. Although the retrieval operation has increased the performance penalty, but it is bounded by $O(\lg n)$ or in case of GPU Euler with buckets of size 512, at binary search would require at most 9 steps.

Algorithm 3.3 GPU based Hash Table construction

Input:
K : Key List
V : Value List
$k_i \in K$, $v_i \in V$, $k_i \Leftrightarrow v_i$
Output:
T : HashTable such that $v_i = T(k_i)$

1: **procedure** ConstructHashTable(K,V)
2: $totalBuckets \leftarrow$ ComputeBucketCount(K)
3: $bucketSize[\,] \leftarrow 0$
4: **for all** $k_i \in K$ **do** ▷ $thread_i$ works on k_i
5: $bucket \leftarrow \mathrm{H}(k_i)$ ▷ H is a hash-function
6: $buketSize[bucket]++$ ▷ atomic increment operation
7: **end for**

8: Compute $bucketOffset$ with CUDPP Parallel-Prefix-Scan
9: **for all** $bucket_i : 1 \leq i \leq totalBuckets$ **do** ▷ one block for each bucket
10: **for all** $thread_j : 1 \leq j \leq bucketSize[i]$ **do**
11: $index \leftarrow bucketOffset[i] + bucketSize[i]--$ ▷ atomic decrement
12: $T\_K[index] \leftarrow k_p$ ▷ copy to buffer
13: $T\_V[index] \leftarrow v_p$
14: **end for**
15: **end for**
16: **for all** $bucket_i : 1 \leq i \leq totalBuckets$ **do** ▷ one block for each bucket
17: Sort each bucket ▷ One thread for each block
18: **end for**
19: **return** T
20: **end procedure**

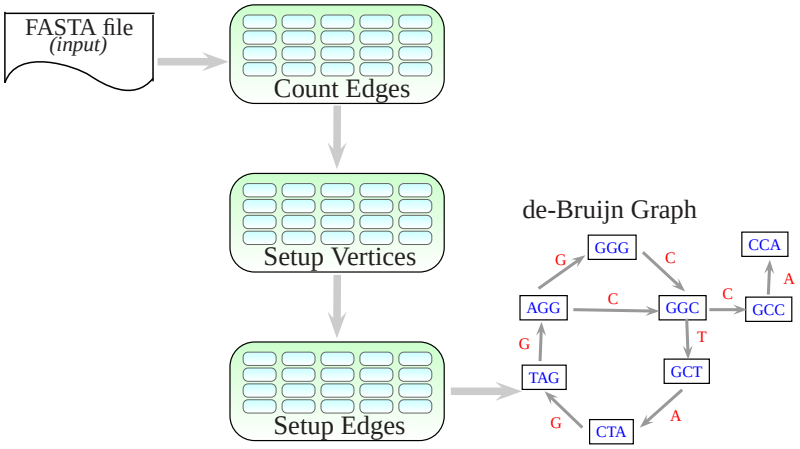

Figure 3.4: de-Bruijn Graph Construction.

3.4.3 Graph Computation

The de-Bruijn graph construction is a 3 step process which involves counting the edges to allocate memory, setting up the vertices, and creating the edges. Each of these steps are performed as CUDA kernel invocations using the GPU as a PRAM machine. Figure 3.4 presents an schematic workflow of all three steps with each box representing a single CUDA kernel launch.

The construction start with lists of tuples representing edges (l-mers) and vertices (k-mers). Although its seems fairly trivial to construct the de-Bruijn graph, the GPU based de-Bruijn graph construction requires some important considerations, such as the amount of memory required to allocate for the complete graph (as dynamic memory allocation is not available to the kernel). During the construction, edges are linked to corresponding vertices and the graph data structure gets updated with the edge and vertex information.

The first step is to count the number of edges each vertex is going to have in the de-Bruijn graph, which prepares the de-Bruijn graph data structure to hold edge information for each vertex. In de-Bruijn graph each vertex will be the suffix or prefix of a tuple representing an edge. Thus, a vertex is also a tuple from read with length one less than the length of the edge tuple. This tuple can be represented as a 64 bit value using 2-bit encoding scheme and

it can serve as unique identifier for each vertex as well. As discussed earlier, a hash table is used to build an index map, so that a vertex information like edge count can be accessed quickly using the vertex tuple representation.

Given an edge tuple (l-mer), with a 64 bit representation, both its source and sink vertices can be easily identified by determining the tuple suffix and prefix using bit operations (shifting and masking). Each vertex maintains the list of entering and leaving edges separately. Once the edge vertice is determined, their respective edge counts is incremented to reflect that the edge is part of the de-Bruijn graph. The CUDA kernel in the first step is invoked in a PRAM model, where each thread works on one edge and updates the associated vertices edge count. The entering and leaving edge counts are maintained in a list which is indexed using the hash table. Since, a vertex can be part of multiple edges, updates to the counters should be consistent across multiple read and writes. In order to achieve the consistency during updates, the edge count is updated using atomic operations provided by the CUDA platform.

The data structure in GPU Euler maintains one incoming edge list and one outgoing edge list for the whole graph, which is nothing but the concatenation of all the edge list of the individual vertices. In order to build this structure, a simple prefix scan on individual edge count is required to calculate the offset of each individual list in the combined edge list. GPU Euler uses CUDPP Parallel Prefix Scan [40–42] which computes the prefix scan in parallel on GPU. With offset information calculated, the graph construction proceeds to the next step.

During the second step of de-Bruijn graph construction, the algorithm utilizes the information gathered in the previous step, to initialize the vertices. There are several pieces of information that must be set for every vertices before the de-Bruijn graph construction is completed. In this step, vertices will be assigned a unique identifier based on the tuple they represent. This step is also executed in a PRAM fashion where each thread is assigned to initialize one vertex. The edge pointer is set to the offset calculated previously and edge count is also updated similarly. Since there is no writes involved on shared resources,

Algorithm 3.4 de-Bruijn Graph Construction on CUDA-based GPUs.

Input:
L : Edge List (l-mers)
K : Vertex List (k-mers)
T : HashTable ($index = T(k_i)$)
Output:
G : de-Bruijn Graph

1: **procedure** CONSTRUCT-DEBRUIJN-GRAPH(L,K,T)
2: Initialize edge counters
/* *Kernel 1 for Counting Edges* */
3: **for all** $l_i \in L$ **do**
4: $p \leftarrow$ PREFIX(l_i)
5: $s \leftarrow$ SUFFIX(l_i)
6: $ecount[T(s)]++$ ▷ increment edge count
7: $lcount[T(p)]++$
8: **end for**
9: **Synchronize**
10: $eoffset \leftarrow$ PREFIX-SCAN($ecount$) ▷ CUDPP Library Prefix-Scan
11: $loffset \leftarrow$ PREFIX-SCAN($lcount$)
12: $\mathcal{V} \leftarrow$ALLOCATEMEMORY
/* *Kernel 2 for Vertices Setup* */
13: **for all** $k_i \in K$ **do** ▷ $\mathcal{V}[T(k_i)] \in \mathcal{V}$
14: $\mathcal{V}[T(k_i)].vid \leftarrow k_i$
15: $\mathcal{V}[T(k_i)].ep \leftarrow eoffset[T(k_i)]$
16: $\mathcal{V}[T(k_i)].lp \leftarrow loffset[T(k_i)]$
17: **end for**
18: **Synchronize**
19: $\mathcal{E} \leftarrow$ALLOCATEMEMORY
/* *Kernel 3 for Edge Setup* */
20: **for all** $l_i \in L$ **do**
21: $p \leftarrow$PREFIX(l_i)
22: $s \leftarrow$PREFIX(l_i) ▷ $e_i \in \mathcal{E}$
23: $e_i.source \leftarrow T(p)$ ▷ set source index
24: $e_i.sink \leftarrow T(s)$ ▷ set sink index
25: $\mathcal{V}[T(p)].lev \leftarrow \mathcal{V}[T(p)].lev \cup i$
26: $\mathcal{V}[T(s)].ent \leftarrow \mathcal{V}[T(s)].ent \cup i$
27: **end for**
28: **return** $G \leftarrow (\mathcal{E}, \mathcal{V}, lev, ent)$
29: **end procedure**

therefore no synchronization is required between the parallel writes originating from all the threads.

After initializing all vertices, we set the edge information in the de-Bruijn graph. To complete this step, a CUDA kernel is invoked, (PRAM) assigning one thread to each edge. Each thread, retrieves a tuple representing each edge, and recomputes its source and sink vertices using tuple's suffix and prefix as done in the first step. Edge structure is also update with the source and sink vertex identifiers (k-mer). Next, the outgoing edge list of the source vertex and the incoming edge list of the sink vertex are updated to include the current edge as well. Atomic operation are used to retrieve an offset and use it to store the edge information.

The complete de-Bruijn graph construction has been outlined in Algorithm 3.4 which constructs de-Bruijn graph using three CUDA kernels. The algorithm expects two tuple list L and K which represents edge and vertex tuples, respectively. Tuples in set L are referenced as l_i and tuples in set K are referenced as k_i. The length of tuples in K is one less than the tuples in L (i.e $|k_i| = |l_j| - 1$). After some internal initialization first kernel is launched to compute number of edges for each vertex. After computing the edge count, CUDPP library is used to perform a parallel prefix scan to compute the offsets. The memory required to store edge and vertex information is allocated and the algorithm proceeds to the next step.

During the next phase, a second CUDA kernel is invoked to set the vertices with the edge list offset and count information. During the final step, a third CUDA kernel is invoked with each thread working on an edge to compute information regarding the source, sink and multiplicity of the edges. Within the CUDA framework, implicit synchronization is performed at the end of each kernel call i.e., all threads are automatically synchronized.

3.5 Euler Tour Construction

Euler tour construction is the crux of eulerian path sequence assembly. An eulerian tour of a graph is a path that visit every edge of the graph exactly once. A de-Bruijn graph

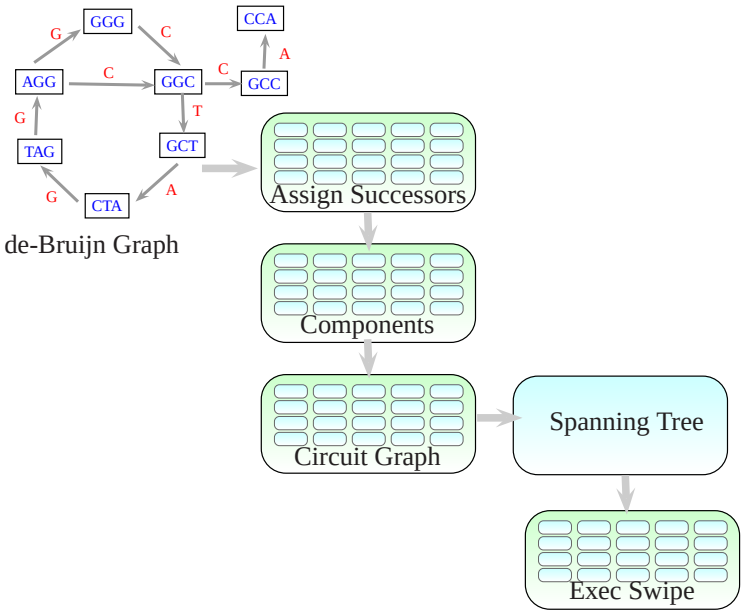

Figure 3.5: Parallel Euler Tour Construction Work-flow.

constructed from the tuples of the reads, represents the bases of the final sequence as its edges, where vertex provides a context for a base pair. As sequence assembly seeks to combine smaller sub-sequence to form a larger sequence, euler path for de-Bruijn graph fulfils the requirements by providing a path visiting all edges in certain order that will give the target sequence.

In any graph there can be more than one solutions for the Euler path. Moreover, error in sequencing generates incorrect reads which result in extraneous and erroneous edges in the de-Bruijn graph. Erroneous edges adds to the complexity of the graph, adding more places to arbitrate which path should be part of the final Euler tour. Pevzner's [2] proposal offers two solutions to handle each situation separately. To combat, erroneous data, it uses a pre-assembly error correction step to collect high quality reads. To reduce the complexity and number of available solutions for Euler path, their approach executes several graph transformation before computing the euler path using the read information which reduces the graph to smaller number of vertices and edges. Graph transformation are currently not

a part of GPU Euler.

The Euler tour construction step is a modified implementation of the algorithm proposed by Vishkin et al.[33] for the CRCW PRAM model. The CRCW PRAM model expects that all processors synchronize after executing each instruction. Processor synchronization is a way through which all participating processors communicate with each. Instruction level synchronization implies that after the execution of each instruction, values computed by any thread should be available to all the other threads. CRCW PRAM also permits concurrent read/write to the same memory location, making it easier to develop parallel algorithms. Concurrent write is always considered to be a nuisance even for the current generation computing devices. One solution that is mostly by different devices to allow concurrent writes is to sequential executes all the writes occurring concurrently in some order.

The modifications were necessary, to adjust algorithm according to CUDA synchronization semantics. CUDA follows SIMT (Single Instruction Multiple Thread) paradigm without inherent instruction level synchronization. In addition to that, CUDA does not support concurrent writes. However, concurrent reads are supported. CUDA offers two kinds of synchronizations[43];

1. **Block Level** All threads in a single block can be synchronized using an API command. However, threads across the blocks aren't affected by this method.

2. **Kernel Level** All threads are synchronized at the launch and completion of CUDA kernels.

These two synchronization mechanism are often used while working on CUDA platform, to accomplish a closer implementation of PRAM model. The referenced algorithm defines concurrent writes as one where arbitrary threads (processors) can win and it does not matter which write succeeded. CUDA atomic APIs can be used to mimic this behavior easily as the atomic APIs which serialize all the concurrent accesses to a memory location, so the access would appear in a sequential manner, without any prescribed order. The net effect of such an implementation is that only the last write will be visible. Synchronization and atomic

operations are costly operations, and as such can lead to performance overhead. Atomic instruction set is available on CUDA devices with compute capability 1.1 and later.

Algorithm 3.5 GPU assisted Euler Tour Construction.

Input:
G : de-Bruijn Graph
Output:
G' : de-Bruijn graph Annotated with Euler Tour Information.

```
1: procedure ComputerEulerTour(G)
2:     Assign Successor Edges.
3:     Construct Successor Graph.
4:     Find Connected Component in Successor Graph.
5:     Compute Circuit Graph.
6:     Compute Spanning Tree on Circuit graph.
7:     Execute Swipe on Spanning Tree.
8:     return G'
9: end procedure
```

The input to the euler tour construction consists of a graph (de-Bruijn graph in our case) represented as a list of vertices V, list of edges, E and two supplementary lists LP and EP, that store the leaving and entering edges for each vertex, respectively. The intuition behind this algorithm is to identify circuits in the given graph, and then change the successor of those edges of a circuit, which are adjacent to an edge belonging to another circuit. To lower the complexity of the circuit graph, a sub-graph is extracted by finding the connected components, a spanning tree of this sub-graph would yield the edges that are required to be switched with their neighbors. Figure 3.5 shows major components of the Euler tour construction and are discussed below.

3.5.1 Successor Assignment

In the successor assignment step, each edge in the input de-Bruijn graph is assigned a successor. The CUDA kernel works on each vertex, assigning one of the leaving edges as the successor for one of the entering edges. This assignment follows a sequential pattern i.e., the first entering edge is paired with first leaving edge, second entering edge to the second leaving edge and so on. Successor assignment is presented in Algorithm 3.6, which

Algorithm 3.6 AssignSuccessors

Input:
G : de-Bruijn graph
Output:
G : de-Bruijn graph with edge successor assigned

```
1: procedure AssignSuccessors(G)
2:     for all v_i ∈ V(G) do
3:         count ← 0
4:         max ← MIN(v_i.ecount, v_i.lcount)
5:         while count < max do                    ▷ connect incoming edge to outgoing
6:             E[ent[v_i.ep + count]].succesor ← lev[v_i.lp + count]
7:             count + +
8:         end while
9:     end for
10:    return G
11: end procedure
```

is executed by all the threads, with each thread working on a single vertex. It iterates the edge list and connects one leaving edge to the incoming edge.

3.5.2 Successor Graph Creation

In this step, a successor graph is generated to represent the successor information we computed in the previous step. This step is required, so that we can identify the circuit to which each edge belongs. The successor graph contains all the edges as vertices and an edge exists between two vertices, if one of them is a successor to the other. This step can be performed in constant time, provided the number of processors are equal to the number of edges. For the successor graph, the number of edges in our algorithm is linear in terms of vertices. The CUDA kernel is launched for each de-Bruijn graph edge, which then sets the information in the graph being constructed. Algorithm 3.7 describes the successor graph creation. It uses a slightly different graph representation as an edge can be related to at most two other edges (preceding and following).

Algorithm 3.7 Successor Graph Creation

Input:
G : de-Bruijn graph with successor information
Output:
V : Vertex set of Successor Graph
E : Edge set of Successor Graph

```
1: procedure CreateSuccessorGraph(G)
2:     V ← E(G)                          ▷ Edges of G, becomes vertices
3:     E ← ∅
4:     for all v_i ∈ V do
5:         if v_i.succ ≠ ∅ then
6:             E ← E ∪ (v_i, v_i.succ)
7:         end if
8:     end for
9:     return (V, E)
10: end procedure
```

3.5.3 Connected Components

Our implementation is based on the algorithm proposed by Uzi Viskin et al.[37] for CRCW PRAM model that requires $O(\log n)$ parallel time to completion. We modified the CRCW model such that based on the iteration step, each thread would be in-charge of either a vertex or edge from the successor graph, with the four steps involving updating of root vertices performed iteratively. Algorithm 3.8 outlines the modified connected component construction as implement for GPU Euler.

3.5.4 Circuit Graph Creation

The circuit graph creation works by calculating the edges between two circuits. At any vertex, instead of considering all possible combination of the edges, the algorithm picks edges which are adjacent to each other in the edge list, i.e. e_i and e_{i+1}. In order to maintain a consistent behavior across different runs, we have stored the edge list of the circuit graph in canonical order.

Algorithm 3.9 describes the process of computing a Circuit graph from the Component labelling of a Successor Graph. It starts with determining the number of vertices in Circuit

Algorithm 3.8 Parallel Connected Component

Input:
V : Vertex set of Successor Graph
E : Edge set of Successor Graph
D : Vertex component identifiers such that d_i identifies component for v_i
Output:
D : Vertex component identifier array.

1: **procedure** ConnectedComponent(V,E,D)
/*temporary variables*/
Q : iteration marker for each vertex ▷ q_i is the iteration d_i is updated
val : temporary arrays to hold intermediate values
$temp$:
s : current iteration count
$s\prime$: next iteration count

2: $s \leftarrow 0$
3: $s' \leftarrow 0$
4: **for all** $d_i \in D$ **do**
5: $d_i \leftarrow i$ ▷ Each vertex is connected to itself
6: $q_i \leftarrow 0$
7: **end for**
8: *Synchronize*
9: **while** $s = s\prime$ **do**
10: **for all** $d_i \in D$ **do**
11: $D[i] \leftarrow D'[D'[i]]$
12: **end for**
13: *Synchronize*
14: **for all** $d_i \in D$ **do**
15: **if** $D[i] \neq D'[i]$ **then**
16: $Q[D[i]] \leftarrow s$
17: **end if**
18: **end for**
19: *Synchronize*

Algorithm 3.8 Parallel Connected Component . contd.

```
20:     for all d_i ∈ D do
21:         for all edge_j ∈ E(v_i) : edge(v_source, v_sink) do
22:             if D[i] = D'[i] then
23:                 temp[j][i] ← D[i]
24:                 val[j][i] ← D[sink]
25:             end if
26:         end for
27:     end for
28:     Synchronize
29:     for all d_i ∈ D do
30:         for all edge_j ∈ E(v_i) : edge(v_source, v_sink) do
31:             if D[temp[j][i]] > val[j][i] then
32:                 D[temp[j][i]] ← val[j][i]
33:             end if
34:             Q[val[j][i]] ← s
35:         end for
36:     end for
37:     Synchronize
```

Algorithm 3.8 Parallel Connected Component . contd.

```
38:     for all d_i ∈ D do
39:         for all edge_j ∈ E(v_i) : edge(v_source, v_sink) do
40:             if D[i] = D[D[i]] ∧ Q[D[i]] < s then
41:                 if D[i] ≠ D[sink] then
42:                     temp[j][i] ← D[i]
43:                     val[j][i] ← D[sink]
44:                 end if
45:             end if
46:         end for
47:     end for
48:     Synchronize
49:     for all d_i ∈ D do
50:         for all edge_j ∈ E(v_i) : edge(v_source, v_sink) do
51:             if D[temp[j][i]] > val[j][i] then
52:                 D[temp[j][i]] ← val[j][i]
53:             end if
54:         end for
55:     end for
56:     Synchronize
```

Algorithm 3.8 Parallel Connected Component . contd.

```
57:         for all d_i ∈ D do
58:             temp[i] ← D[D[i]]
59:         end for
60:         Synchronize
61:         for all d_i ∈ D do
62:             D[i] ← temp[i]
63:         end for
64:         Synchronize
65:         for all d_i ∈ D do
66:             if Q[i] = s then
67:                 s' ← s' + 1
68:             end if
69:         end for
70:         s ← s + 1
71:         D' ← D
72:     end while
73:     return D
74: end procedure
```

graph, by computing the number of root vertices identified during connected component labelling. After allocating the memory for vertices, the algorithm moves on to determine the number of edges in the circuit graph, by counting edges of a vertex in de-Bruijn graph that are are labelled to be in a different connected component.

3.5.5 Spanning Tree

This step is implemented serially on the CPU using the boost graph library [35], which implements the Kruskal spanning tree algorithm and requires $O(|E| \log |V|)$ time. In our situation $|E|$ is of the order of $O(|V|)$ and the actual time will be dependent on the number of circuits identified in the successor graph.

3.5.6 Swipe Execution

The final step requires a traversal of the edges in spanning tree identified in the previous step. Each edge of the spanning tree corresponds to a pair of edges in de-Bruijn graph incident on same vertex, swapping their successors will connect path containing one edge with the path containing the other edge (Algorithm 3.10), resulting in a connected Eulerian tour. Contigs are generated by identifying the source vertices and following the successor

Algorithm 3.9 Circuit Graph Creation

Input:
G : de-Bruijn graph
V : Vertex set of successor graph
Output:
CG : Circuit Graph

```
1:  procedure CreateCircuitGraph(G, D)
2:      for all e_i ∈ E(G) do                          ▷ one thread for each edge
3:          C[D[i]] = 1                                ▷ mark root vertex
4:      end for
5:      Compute total number of root vertice
6:      Allocate memory for vertices
7:      for all v_i ∈ V(G) do                          ▷ one thread for each vertex
8:          for all edge ∈ v_i do
9:              if D[edge_i] ≠ D[edge_{i+1}] then
10:                 edgeCount++
11:             end if
12:         end for
13:     end for
14:     Allocate memory for CG edges
15:     for all v_i ∈ V(G) do                          ▷ one thread for each vertex
16:         for all edge ∈ v_i do
17:             if D[edge_i] ≠ D[edge_{i+1}] then
18:                 add edge (D[edge_i], D[edge_{i+1}]) in CG
19:             end if
20:         end for
21:     end for
22:     return CG
23: end procedure
```

edge information pertaining to each edge.

Algorithm 3.10 Swipe Execution

Input:
G : de-Bruijn graph
CG : Circuit Graph
STt : Spanning Tree for Circuit Graph
Output:
G : Annotated de-Bruijn graph

```
1: procedure ExecuteSwipe(G,ST,CG)
2:     for all v_i ∈ V do
3:         if (D[e_i], D[e_{i+1}]) ∈ ST then
4:             exchange successor of e_i and e_{i+1}
5:         end if
6:     end for
7:     return G
8: end procedure
```

3.6 Time Complexity Analysis

The de-Bruijn Graph construction is a constant time operation $O(1)$ given $O(n)$ processors, which can be arranged by assigning each k-mer to a single CUDA-based GPU thread. The CRCW Euler tour construction algorithm [33] has a run time complexity of $O(\log n)$ for n vertices. Our modifications, introduce a constant step that does not affect the complexity of Eulerian tour construction phase, which is bounded by $O(\log n)$. Specifically, identifying the component requires $O(\log n)$ parallel steps, and the circuit graph creation and successor graph creation is a constant time operation in terms of number of vertices. Prefix scan for calculating the required memory also takes $O(\log n)$ parallel time. The Kruskal spanning tree algorithm is bounded by $O(|E| \log |V|)$ where $|V|$ is number of vertices and $|E|$ is number of edges in the successor graph. The number of vertices in the successor graph tend to be far lesser than the number of vertices in the de-Bruijn graph, and the dominating factor in the time complexity analysis is not affected by the spanning tree algorithm. Hence, the overall run time complexity of our GPU-Euler is bounded by $O(\log n)$, as shown by analyzing different phases of the algorithm.

Table 3.3: Genome size and number of simulated reads for different read lengths.

Genome	Length	36 bp	50 bp	256 bp
CJ	1,641,481	911,934	656,593	128,241
NM	2,184,406	1,213,559	873,763	170,657
LL	2,635,589	1,314,216	946,236	184,812

3.7 Experimental Setup

3.7.1 Dataset

To evaluate the performance of our GPU-based assembler we used three previously assembled genomes: (i) Campylobacter Jejuni (CJ), (ii) Neisseria Meningitidis (NM) and (iii) Lactococcus Lactis (LL). These genomes have been used for benchmarking various other assembly algorithms including Euler [7]. The genome sizes (lengths) of CJ, NM and LL are 1.6Mbps, 2.1 Mbps and 2.3 Mbps, respectively. For each of the assembled genomes we simulated error free reads using MetaSim [44]. The read lengths were varied to be 36 bp, 50 bp and 250 bp for three independent set of experiments, representing the NGS technologies. For each experiment, the number of reads simulated achieved a 20x coverage for the genomes and are summarized in Table 3.3. Since, we were focused on illustrating the performance of GPU-Euler in terms of run times we simulated only error free reads.

3.7.2 Experimental Protocol

We performed a comprehensive set of experiments that assessed the run time performance of GPU-Euler across the three different genome benchmarks, and with varying read lengths. We performed run time profiling of our method, evaluating and optimizing the speed of different phases of the GPU-Euler algorithm. We also compared the performance of our approach to well established sequence assemblers like EulerSR [8]. We ran each experiment multiple times to ensure that the run-times remained consistent due to load factors on the workstation. We did not notice any significant variability across multiple runs with the same parameters, and as such do not report them in this study. The GPU-Euler algorithm has

Table 3.4: Profiling of GPU-Euler.

Phase	Computation
I/O and k-mer Extraction	CPU + **GPU**
Hash Table Construction	**GPU**
de-Bruijn Graph Construction	**GPU**
Euler Tour Construction	**GPU** + CPU
Sub-steps for Euler Tour Construction	
Finding Connected Component	**GPU**
Circuit Graph Creation	**GPU**
Spanning Tree	CPU
Swipe Execution	**GPU**
Traversal (Other)	**GPU**
Contig Generation (O/P)	CPU

different phases of the algorithm run on the GPUs using CUDA kernel launches, whereas some of the phases are run on the CPU and some of the steps are run on the CPU as well as GPUs. We show in Table 3.4 the different steps of the GPU-Euler algorithm and their execution pattern.

For contigs greater than 100 bp we report the total bases within the contigs, mean and maximum length of contigs obtained. We also compute the N50 score, which is defined as the length of smallest contig such that 50% of the genome length is contained in contigs of size N50 or greater. To compute the accuracy and coverage statistics, we used the NUCMER pipeline of MUMMER [45] that allowed for quick and fast alignment of assembled contigs to the input genomes. NUCMER uses a suffix-tree based string matching algorithm to search for exact matches, and extends these matches using a dynamic programming based alignment that is considerably faster than BLAST. Using the alignment we calculate the length weighted accuracy.

3.7.3 Hardware Configuration

The benchmarking of GPU-Euler was performed on a Dell workstation which has a quad core Intel Xeon 2.00 GHz processor with 8 GB primary memory. This system has a nVidia Quadro FX 5800 GPU, which has a clock rate of 1.30 GHz, 240 cores, 4 GB GPU RAM and

Table 3.5: Run Time Performance for CJ Genome using GPU-Euler.

l	I/O	**GPU Encoding**	**Hash Table**	**de-Bruijn**	Euler Tour				Output	GPU	CPU	Total
					Comp.	Sp. Tree	**Swipe**	**Other**				
					read length : 36 bp							
16	30.847	**0.361**	**0.131**	**0.484**	**4.178**	0.065	**0.004**	**0.127**	1.537	5.286	34.393	39.679
18	25.952	**0.372**	**0.138**	**0.511**	**4.245**	0.011	**0.004**	**0.052**	1.576	5.321	29.500	34.821
22	22.659	**0.42**	**0.139**	**0.510**	**4.269**	0.001	**0.003**	**0.094**	1.667	5.435	26.352	31.787
24	21.076	**0.444**	**0.138**	**0.509**	**4.229**	0.001	**0.003**	**0.063**	1.692	5.386	24.789	30.175
26	19.610	**0.469**	**0.138**	**0.504**	**4.197**	0.001	**0.003**	**0.043**	1.745	5.355	23.370	28.725
28	17.947	**0.494**	**0.137**	**0.499**	**4.162**	0.001	**0.003**	**0.036**	1.771	5.331	21.732	27.063
30	16.457	**0.519**	**0.136**	**0.498**	**4.115**	0.001	**0.003**	**0.031**	1.812	5.302	20.257	25.559
32	12.702	**0.545**	**0.132**	**0.475**	**3.856**	0.001	**0.003**	**0.03**	1.832	5.041	16.475	21.516
					read length : 50 bp							
16	34.155	**0.325**	**0.129**	**0.488**	**4.197**	0.077	**0.004**	**0.151**	1.544	5.294	37.715	43.009
18	29.08	**0.339**	**0.137**	**0.512**	**4.263**	0.018	**0.004**	**0.063**	1.589	5.317	32.654	37.971
22	26.511	**0.387**	**0.137**	**0.515**	**4.298**	0.006	**0.003**	**0.146**	1.658	5.486	30.185	35.671
24	25.44	**0.413**	**0.136**	**0.504**	**4.323**	0.005	**0.003**	**0.162**	1.714	5.541	29.174	34.715
26	24.312	**0.437**	**0.137**	**0.504**	**4.316**	0.004	**0.003**	**0.163**	1.742	5.56	28.071	33.631
28	23.227	**0.462**	**0.136**	**0.505**	**4.298**	0.002	**0.003**	**0.146**	1.78	5.551	27.033	32.584
30	22.11	**0.487**	**0.136**	**0.504**	**4.306**	0.001	**0.003**	**0.109**	1.823	5.544	25.954	31.498
32	20.961	**0.511**	**0.136**	**0.503**	**4.248**	0.001	**0.003**	**0.086**	1.871	5.487	24.849	30.336
					read length : 256 bp							
16	41.054	**0.559**	**0.128**	**0.488**	**4.365**	0.126	**0.004**	**0.24**	1.587	5.785	44.699	50.484
18	35.224	**0.505**	**0.136**	**0.510**	**4.37**	0.051	**0.004**	**0.144**	1.641	5.669	38.87	44.539
22	34.335	**0.532**	**0.135**	**0.510**	**4.413**	0.036	**0.004**	**0.266**	1.727	5.86	38.096	43.956
24	34.053	**0.550**	**0.137**	**0.503**	**4.415**	0.033	**0.004**	**0.296**	1.763	5.905	37.871	43.776
26	34.114	**0.572**	**0.136**	**0.503**	**4.424**	0.03	**0.004**	**0.317**	1.787	5.956	37.911	43.867
28	33.880	**0.589**	**0.136**	**0.503**	**4.433**	0.026	**0.004**	**0.323**	1.826	5.988	37.711	43.699
30	33.661	**0.608**	**0.136**	**0.502**	**4.484**	0.026	**0.004**	**0.321**	1.872	6.055	37.563	43.618
32	33.729	**0.628**	**0.136**	**0.503**	**4.389**	0.027	**0.004**	**0.314**	1.914	5.974	37.664	43.638

The run-times are reported in seconds. The phases indicated in bold i.e.,GPU Encoding Hash Table, Component, Swipe and Other are performed on the GPUs.

CUDA compute capability 1.3. We used the CUDA SDK version 2.3 to build GPU-Euler. The GPU-Euler uses both CPU and GPU, whereas the EulerSR (used for comparison) was run on a single core of the Intel processor.

3.8 Result

3.8.1 Runtime Performance

Table 3.5 shows the run times (in seconds) across the different phases of GPU-Euler performed by varying the overlap parameter, k-mer size from 16 to 32 across the CJ genome. We performed experiments across reads of length, 36, 50 and 256 bps. We report the run-times for the different phases of the algorithm across the GPU and CPU as shown in Table 3.4. We also report the total run-time along with the total GPU and CPU run-times.

As we increase the value of k, the CPU run-time gradually increases for all the three

genomes. A large percentage of time (approximately 90%) is spent in extracting k-mers from the several input read sequences (denoted as I/O). During this phase, a file containing the sequence reads (in FASTA format) is processed sequentially and a list of k-mers along with their occurrence frequency is generated.

The different phases of GPU-Euler that are run on the GPUs do not vary significantly, in terms of run-times as the k parameters is increases. The GPU run-time is dominated by the connected component phase algorithm which uses 8 CUDA kernel launches in an iterative fashion. These kernels use CUDA synchronization API to serialize writes for given memory execution, which explains the relatively higher execution time for this phase.

When comparing the performance across the three genomes, we notice that as size of genome increases from CJ to NM to LL, the number of input reads increase to maintain the 20x coverage (Table 3.3). As such, the I/O and k-mer extraction increases with increasing genome sizes. The average GPU run times for the CJ, NM and LL genomes are 5.656, 7.520 and 8.184 seconds, respectively. This change in the GPU-time is primarily because of larger de-Bruijn graph size with increasing number of input reads. In Figure 3.6 we show a plot of the GPU run times across these three genome benchmarks for the different read lengths.

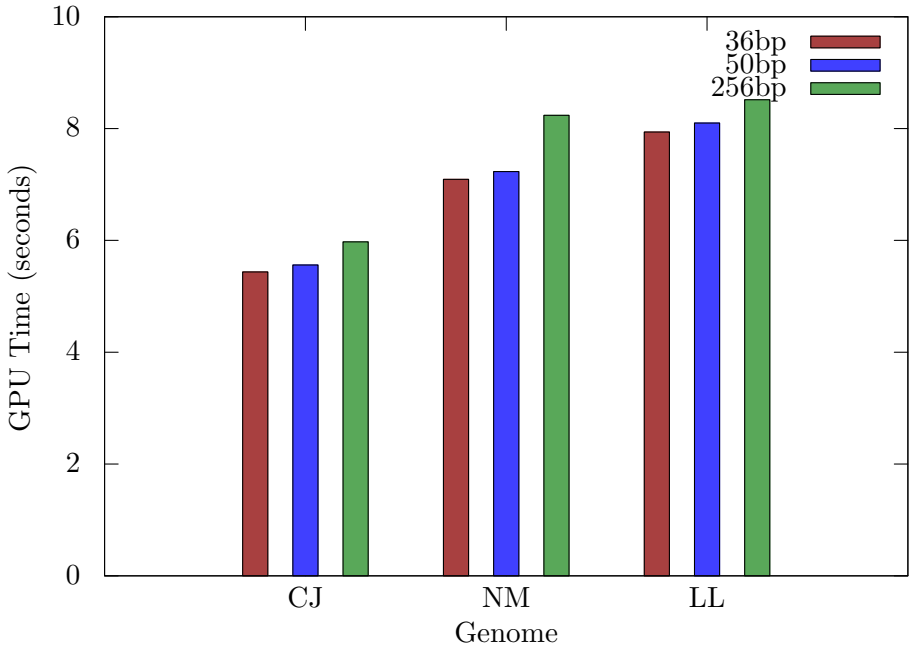

Figure 3.6: GPU Time comparison across different genomes

3.8.2 Comparative Performance

We compared our assembler with EulerSR [8], a widely used Eulerian-based assembler developed for short reads. Experiments for EulerSR were performed on a single CPU core of our workstation. Table 3.6, Table 3.7 and Table 3.8 show the contig length statistic, contig accuracy results and the run-time for 36, 50 and 256 bps for the CJ,NM and LL genome benchmark, and compare it to EulerSR. We report those results for the different assemblers that achieved the best N50 score. For EulerSR we also invoked a parameter (minMult = 20) that will filter k-mers that do not occur at least twenty times. These results are reported in the Tables as "EulerSR*".

Across the three genome benchmarks, we notice that GPU-Euler consistently outperforms EulerSR in terms of run times. However, note GPU-Euler is utilizing the computing capacity of the GPUs whereas EulerSR is benchmarked on single processor. As the read length increases from 36 to 256 base pairs, we notice that the run time for GPU-Euler

Table 3.6: Comparative Performance for GPU-Euler on CJ genome (Contigs ¿= 100 bp).

Read Len.	Assembler	l	Time (s)	N50	N	Mean	Max	TB	WA
36 bp	EulerSR*	16	122.391	5345	21	503.810	5345	10580	99.35
		22	100.676	136	17	144.882	244	2463	100
	EulerSR	20	176.858	17827	227	7408.696	57201	1681774	97.37
		22	162.582	7386	387	4192.140	36866	1622358	98.62
	GPU Euler	22	31.787	8480	720	4491.206	40255	3233668	83.23
50 bp	EulerSR*	22	101.220	6085	25	441.840	6085	11046	99.88
		26	96.496	1686	9	849.667	3637	7647	99.09
	EulerSR	21	151.444	79838	114	14404.965	155588	1642166	95.33
		26	141.378	46497	124	13042.726	97486	1617298	97.22
	GPU Euler	26	33.631	47766	307	10527.147	158836	3231834	91.01
256 bp	EulerSR*	18	114.383	511	1529	394.891	3476	603788	78.01
		32	-	-	-	-	-	-	-
	EulerSR	27	103.104	112428	69	24544.275	191547	1693555	95.38
		32	-	-	-	-	-	-	-
	GPU Euler	32	43.638	13806	597	5521.774	59742	3296499	98.25

EulerSR* represents a run of EulerSR with `-minMult=20`. We report for each assembler the results with k-mer size which produces the best N50 score. For EulerSR we also report the results for the k-mer chosen for GPU-Euler. For $k > 30$, EulerSR produces a memory error. N50, N, Mean, Max are the N50 scores, total number of contigs, mean contig lengths and maximum contig lengths, respectively. TB and WA denote the total number of aligned bases within the contigs and the weighted accuracy, respectively.

increases. This is primarily because of processing longer reads during the I/O phase.

With respect to contig length statistics and accuracy, we notice that EulerSR shows better performance. GPU-Euler shows better or comparable N50 scores and mean contig lengths for few of the cases. The implemented GPU-Euler works on the full de-Bruijn graph without any compaction and translation. EulerSR implements several heuristics that analyze the de-Bruijn graph structure and help resolve repeat regions within a genome.

3.9 Conclusion

We investigated the potential of using GPUs for performing genome sequence assembly task. We developed an Eulerian-based sequence assembler that used the GPU in conjunction with

Table 3.7: Comparative Performance for GPU-Euler on NM genome (Contigs ¿= 100 bp).

Read Len.	Assembler	l	Time (s)	N50	N	Mean	Max	TB	WA
36 bp	EulerSR*	20	145.105	2063	23	482.826	3830	11105	83.87
		22	144.716	1083	19	510.000	2852	9690	79.63
	EulerSR	21	279.277	4538	894	2307.614	21195	2063007	96.73
		22	257.953	3742	1011	2028.121	16374	2050430	96.90
	GPU Euler	22	40.670	3909	2275	1804.153	17635	4104447	81.37
50 bp	EulerSR*	26	128.601	1020	32	399.594	5949	12787	87.88
		23	136.295	688	54	321.759	6283	17375	88.27
	EulerSR	25	222.046	6808	601	3370.797	25857	2025849	96.93
		23	225.512	6574	594	3395.434	23693	2016888	97.89
	GPU Euler	23	45.653	6596	1964	2121.377	25383	4166385	79.03
256 bp	EulerSR*	16	198.608	481	2245	367.697	5227	825479	69.87
		31	22997.298	-	-	-	-	-	-
	EulerSR	27	162.595	31614	346	9365.142	79346	3240339	47.26
		31	-	-	-	-	-	-	-
	GPU Euler	31	57.817	7226	1610	2715.053	30965	4371235	79.72

EulerSR* represents a run of EulerSR with `-minMult=20`. We report for each assembler the results with k-mer size which produces the best N50 score. For EulerSR we also report the results for the k-mer chosen for GPU-Euler. For $k > 30$, EulerSR produces a memory error. N50, N, Mean, Max are the N50 scores, total number of contigs, mean contig lengths and maximum contig lengths, respectively. TB and WA denote the total number of aligned bases within the contigs and the weighted accuracy, respectively.

the CPU. Our empirical results showed that this GPU-based assembler had better run time performance in comparison to EulerSR on three bacterial genome benchmarks, across reads representing NGS data. We also showed competitive contig length statistics but in terms of accuracy there is room for improvement.

Table 3.8: Comparative Performance for GPU-Euler on LL genome (Contigs ¿= 100 bp).

Read Len.	Assembler	l	Time (s)	N50	N	Mean	Max	TB	WA
36 bp	EulerSR*	19	158.887	5358	7	1380.143	5358	9661	92.81
		21	152.476	5222	3	3045.000	5222	9135	97.52
	EulerSR	20	276.670	9180	619	3862.969	31070	2391178	98.89
		21	270.720	9154	631	3753.751	33986	2368617	97.12
	GPU Euler	21	46.080	7234	1500	3061.075	37629	4591612	79.17
50 bp	EulerSR*	24	139.019	5381	18	590.000	5381	10620	93.68
		26	133.875	5379	10	988.700	5379	9887	97.78
	EulerSR	25	216.472	30133	317	7264.991	83247	2303002	96.92
		26	216.281	27804	318	7168.132	83249	2279466	99.13
	GPU Euler	26	48.061	23998	877	5259.621	95876	4612688	83.68
256 bp	EulerSR*	17	189.402	479	2319	377.916	3719	876388	74.71
		32	-	-	-	-	-	-	
	EulerSR	27	167.085	74484	193	11967.648	198384	2309756	96.99
		32	-	-	-	-	-	-	-
	GPU Euler	32	63.522	13789	976	4853.170	66109	4736694	93.84

EulerSR* represents a run of EulerSR with `-minMult=20`. We report for each assembler the results with k-mer size which produces the best N50 score. For EulerSR we also report the results for the k-mer chosen for GPU-Euler. For $k > 30$, EulerSR produces a memory error. N50, N, Mean, Max are the N50 scores, total number of contigs, mean contig lengths and maximum contig lengths, respectively. TB and WA denote the total number of aligned bases within the contigs and the weighted accuracy, respectively.

Chapter 4: Error Correction

This chapter discusses an error correction technique and its implementation on GPU to improve assembly accuracy.

4.1 Introduction

The process of genome sequencing is often prone to error. Different techniques yields reads containing different error profiles. These error containing reads can be problematic during the assembly phase. They may introduce erroneous paths in the de-Bruijn graph which lead to invalid paths and incorrect assembly. Other assembly techniques such as greedy or OLC based approaches are also affected by the error during the sequencing. These error usually lead to tangles and small branches in the read graph and thus mislead the assembly algorithm. Each assembly algorithm handles the error containing reads in some manner that can complement their algorithm. One common method is to estimate the errors and filter out reads that contain errors. In order to estimate errors correctly, the genome should be sequenced with a high coverage. **Coverage** is defined as the average number of times a base is read at a position while sequencing a genome. Higher coverage can help the OLC based approach, where during consensus phase, errors can be outnumbered by the correct reads. Another similar technique is employed by AbySS which performs trimming as a post processing step after constructing the read graph. During this post processing step, Abyss tries to remove bulges and branches which are shorter than some threshold length.

The Sanger technique which attempts to collect longer reads (˜1500 bp long) produces reads with a higher error rate. With newer sequencing technologies like Solexa, the focus has been shifted to produce shorter reads with less error rate.

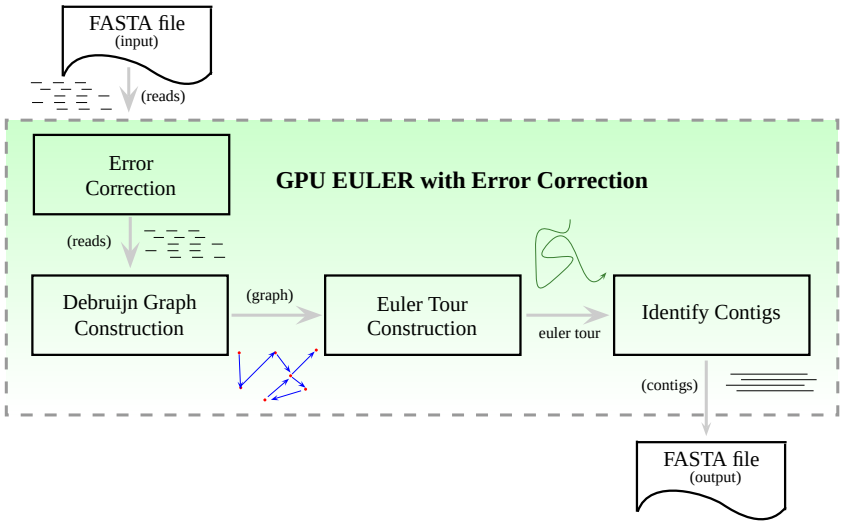

Figure 4.1: GPU Euler Workflow with Error Correction

4.2 Method

The selection of an error correction algorithm depends on a number of factors. Each assembly method employs a certain heuristics to reduce error based on the error profiles of different sequencing technologies to guide the error correction mechanism. Some assembly algorithms like AbySS, incorporates error correction as an integrated step during the assembly, while some assembly methods (e.g Euler), introduce error correction as a pre-processing step.

In GPU Euler, error correction is performed as a pre-processing step before constructing the de-Bruijn graph as shown in Figure 4.1. Since erroneous reads will cause a large number of invalid edges in the de-Bruijn graph which can not only increase the graph complexity but will also introduce invalid paths. This situation can even lead to graphs with no solution at all. In order to minimize the impact of error containing reads, GPU Euler therefore, performs error correction before performing any other operation.

4.2.1 Spectral Alignment

Spectral Alignment is an intuitive method for error correction introduced by Pevzner [2] for Euler. It is based on a simple principle that requires mutating an error containing read such that it can resemble high coverage reads. Consider all tuples of length t from a set of reads. With enough read coverage, the occurrence of certain tuple from a read should be approximately equal to the read coverage. Now consider tuples from an error containing read. A read with one erroneous base would give rise to approximately $2 \times (n - t + 1)$ tuples with very low coverage. Based on this observation, a correct mutation of an error containing read would reduce the number of less occurring tuples by $2 \times (n - t + 1)$ tuples.

In Pevzner's formulation of spectral alignment problem, the tuples are partitioned into two set; one having high occurrence rate known as *solid* tuple set while the other low occurrence rate are know as *weak* tuples set. Tuples in the *weak* set are likely appeared as a result of an error during sequencing. The goal of spectral alignment is to find minimum mutations that can reduce the set of *weak* tuples. Based on the error model, a read can contain error at multiple positions, which requires multiple mutation to be introduced during the error correction. Often a threshold is specified to limit the number of mutations, and any reads requiring more mutations than the threshold are simply discarded as poor reads. This cut-off is necessary and threshold is usually kept at a minimum to avoid introducing error as part of the error correction itself.

4.2.2 Problem Decomposition

In GPU Euler a simple method of error correction has been adopted which is based on the spectral alignment. Spectral alignment belongs to a class of problem of known as embarrassingly parallel problems. These problems are termed as their solution can be obtained by decomposing them into smaller sub-problem and computing their result in parallel. Such problems are good candidates for GPU based implementation as there is less data dependence between the sub-problems.

Error correction in GPU Euler can be represented as the generation of solution space

containing all possible mutations of all the reads and selection of the best mutation from the generated solution space. Let a read r be defined as a string of some length n over the alphabet A, G, C, T.

$$r = (b_1, b_2, \ldots, b_n) : b_i \in \{A, G, C, T\}$$

The complement of the base b can be represented as b' based on Table 2.1.

$$b' = \begin{cases} A & \text{if } b = T \\ G & \text{if } b = C \\ C & \text{if } b = G \\ T & \text{if } b = T \end{cases}$$

The reverse complement r' of a read r can be defined as a string of complement of bases of r in reverse order.

$$r' = (b'_n, b'_{n-1}, \ldots, b'_1)$$

A tuple t_i of read r is a sub-string of r starting at position i with t bases. t'_i can be similarly defined for r'.

$$t_i = (b_i, b_{i+1}, \ldots, b_{i+t-1})$$

As part of error correction procedure, each read is mutated by substituting all four bases at every position of that read, in order to obtain $4 \times n$ mutated reads of a single read. Let Q be the set of mutations such that q_j be a mutation representing a substitution at jth position of a read r.

$$Q = \{q_j \in Q \colon 1 \leq j \leq n\}$$

Let $\otimes$ be the mutation operator defined over a read r and mutation q_j such that it

mutates the read r to four possible reads, by substituting each nucleotide at jth position.

$$r \otimes q_j = \{r_{A,j}, r_{G,j}, r_{C,j}, r_{T,j}\} = \{r_{s,j} \colon s \in \{A, G, C, T\}, b_j = s\}$$

In order to determine if a mutated read r has tuples from the solid set, a tuple set $tuples_t(r)$ needs to be calculate first. This set will contain all tuples of length t for a read r and its complement r'. $tuples_t(r)$ can be defined as,

$$tuples_t(r) = \{\{t_i, t_i'\} \colon 1 \leq i \leq n - t\}$$

Let R be the set of reads such that r_k is the kth read.

$$R = \{r_k \in R \colon 1 \leq k \leq m\}$$

$$Q \otimes R \rightarrow T_{b,j,k}$$

$$\otimes(q_{b,j}, r_k) \rightarrow T_{b,j,k} = \{t_{i,b,j,k} \in T_{b,j,k} \colon where t_{i,b,j,k}\}$$

Given the set of all possible mutations, a scoring function is required that can define the precedence of one mutation over the other and identify the best substitution and its position in a read from the set of its possible mutation. GPU Euler uses a simple scoring function which is based on the number of solid tuples in the tuple set of a mutated read. A threshold M is selected such that tuples occuring atleast M times are considered as solid tuples.

GPU Euler initially computes the frequency count of all tuples from the input set of reads in a function $H(t_i)$. Any tuple not present in the input set of reads is assigned 0 as its frequency count. Using this function with a threshold M, an scoring function for any tuple can be defined as having a value of 1 if its frequency count is at least M and 0 otherwise. In other words, the function $f(t_i)$ simply classifies a tuple as either weak or solid.

$$f(t_i) = \begin{cases} 1 & H(t_i) \geq M, \\ 0 & otherwise \end{cases}$$

The score of a mutated read $r_{s,j}$ can be represented as the number of its solid tuples. The number of solid tuples in a read can be computed as sum of all $f(t_i)$ from the tuple set of $r_{s,j}$.

$$score(r_{s,j}) = \sum_{i=1}^{n} f(t_i), t_i \in tuples_t(r_{s,j})$$

$score(r_{s,j})$ is computed for all possible mutations of every read. Using this score, best substituted nucleotide at every position of all the reads is computed as the one having maximum score among all substitutions. For every position, there will be four possible scores pertaining to four nucleotide substitutions. The substitution function selects maximum score from the available four score.

$$substitution(r \otimes q_j) = \bigvee_{s \in \{A,G,C,T\}} score(r_{s,j})$$

Finally, the best position for a mutation is determined by selecting the position with maximum substitution score among all the position of a read.

$$position(r \otimes Q) = \bigvee_{j=1}^{n} substitution(r \otimes q_j)$$

This solution can be applied in parallel for every read leading to a simple parallel application. The error correction procedure can spawn parallel task for each read which can independently compute the mutated reads, generate the tuples and assign score to each

Table 4.1: Computations for Single Read

Read Length (bp)	tuples ($t = 16$)	($t = 20$)	($t = 26$)	($t = 32$)
36	6048	4896	3168	1440
50	14000	12400	10000	7600
256	493568	485376	473088	460800

read. This process can further parallelized by computing all $4 \times n$ mutated reads of a single read. Tuple generation can be performed in parallel as well as explained in the next section.

4.3 Implementation

Error correction in GPU Euler is performed as an exhaustive search of mutated reads to find one such that all of its tuples would belong to the *solid* set. The search space consists of possible mutations of every read from the input set. As part of computations, all the mutations of every read are calculated by substituting each nucleotide (A, G, C, T) at every position of a read and calculate resulting tuples. For a read of length n, there are n positions to mutate and for each position, there are four possible mutations, which leads to $4 \times n$ possibilities. Each of the $4 \times n$ possible mutation would lead to $2 \times (n - t + 1)$ tuples (including forward and reverse strands), for a total of $4 \times n \times 2 \times (n - t + 1)$. Table 4.1 shows the number of tuples computed for a single read of various length. It is evident that with the increase in the length of reads, the tuples required to be computed increases significantly.

Error correction step in GPU Euler performs most of its computation and filtering on the GPU, to exploit its parallel architecture. As part of error correction, a small pre-processing is required to count the frequency of tuples in the input set. The tuples are encoded to 64-bit integer values. This information is stored in a hash table as key-value pair. A hash table is a preferred data structure here, since it can store sparse indexes (tuples) and provides near constant time look up for the required value.

The rest of the error correction steps process the reads in batches to keep the memory requirements within the GPU specification.The selection of batch size requires some consideration as well. The computational resources (such as SMs and shared memory) on a GPU are limited. If a large batch is selected then each kernel invocation would take longer to finish, as it would increase the number of blocks required to finish the task. A smaller batch size would increase the synchronization events between each kernel invocations.

Algorithm 4.1 Error Correction on GPU

Input:
R : read list

Output:
R' : Modified Reads

1: $T \leftarrow$ ExtracTuples(R, t)
2: $batchCount \leftarrow |R| /$MaxRead
3: Allocate Memory for $buffer, mutation, score, substitution, position, result$
4: **for** i: $1 \rightarrow batchCount$ **do**
5: $\quad buffer \leftarrow$ CopyBatch(R, i)
6: $\quad mutation \leftarrow$ CalculateMutation$(buffer)$
7: $\quad score \leftarrow$ AccumulateScore$(mutation)$
8: $\quad substituion \leftarrow$ SelectSubstitution$(score)$
9: $\quad position \leftarrow$ SelectPosition$(substituion)$
10: $\quad$ **for** j: $1 \rightarrow$ MaxRead **do**
11: $\quad\quad result[i \times$ MaxRead $+ j] \leftarrow position[j]$
12: $\quad$ **end for**
13: **end for**

14: **for all** $r_i \in Read$ **do**
15: $\quad R' \leftarrow R' \cup$ ApplyMutation$(r_i, result[i])$
16: **end forreturn** R'

17: **procedure** ApplyMutation(r,result)
18: $\quad$ **if** $result >$ Threshold **then**
19: $\quad\quad r' \leftarrow r \oplus result$
20: $\quad$ **else**
21: $\quad\quad r' \leftarrow \emptyset$
22: $\quad$ **end if**
$\quad$ **return** r'
23: **end procedure**

Algorithm 4.1 presents the work flow of the error correction in GPU Euler. It expects a set or read as an argument alongwith the tuple length t. The algorithm then proceeds with

extracting tuples of length t from the input set of reads, and builds a lookup table T (line 1). After collecting the tuple set and their occurrence count, Algorithm 4.1 then computes the number of batches and allocates the GPU memory required for each iteration (line 2 – 3).

The error correction algorithm then continue to process reads in batches containing $MaxRead$ reads by computing mutation and selecting the best mutation for each read (line 4 – 13). In each iteration of this loop , algorithm 4.1 invokes following 4 cuda kernels:

1. Compute Mutated tuples.
2. Accumulate Mutation Score.
3. Substitution Selection.
4. Position Selection.

At the end of each iteration, the result of CUDA kernel is copied from GPU memory to the system memory (line 10 – 12). After processing all the batches, error correction procedure then transform the reads using the gathered results.

During the read processing, Algorithm 4.1 invokes 4 CUDA kernels which are further described below.

4.3.1 Mutation

ComputeMutations is first of the four CUDA kernel in the error correction process, which performs the mutation and marks each generated tuple as solid or weak. In order to fully exploit the CUDA shared memory, threads are grouped together in a way that can maximize the shared memory access across a thread block. Computations are represented as launch configurations such that one block is assigned to compute four possible mutations at a single position of a read. For a batch of m reads of length n each, a grid of dimensions $n \times m$ blocks is specified in the launch configuration, with each block having n threads as shown in Figure 4.2. Each column in the grid represents mutations on a single read for a total of

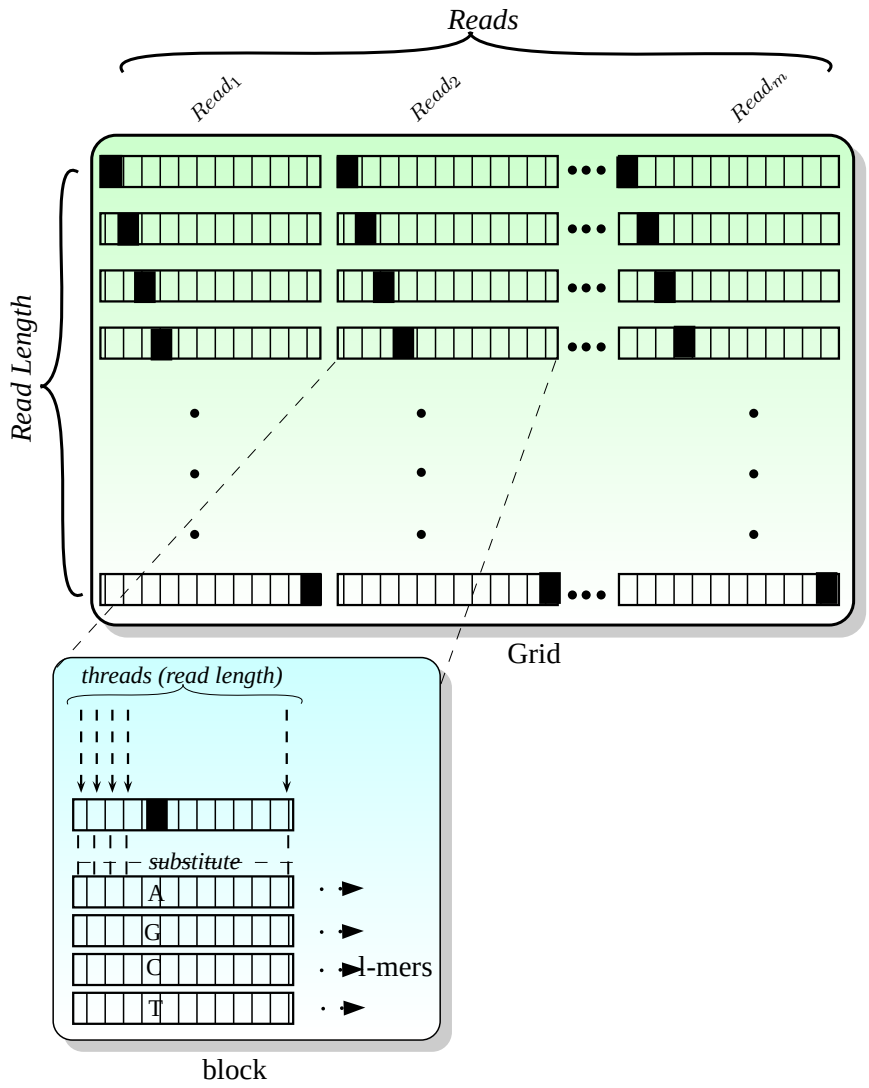

Figure 4.2: Computing Mutation

m columns. Each row of the grid column computes mutation of a read at a single position for a total of n rows. Each block of the grid contains n threads which maps to each base position in the read.

All blocks in a column performs scatter operation, replicating a read n times for each block. All n threads of a block take part in replicating read to block's shared memory by copying single base from GPU global memory. After computing the mutation each blocks

extract tuples and check their occurrence in the spectrum through constructed hash table. Since the kernel extract all tuples from a read in parallel, there must be at least n-l+1 threads required in each block. To avoid unnecessary conditional statements we use n threads and their computations are masked off at the later stages.

Algorithm 4.2 Computing Mutation

Input:
R : read list
T : Tuple Spectrum

Output:
L : Mutated tuples

```
1:  procedure MUTATE(R)
2:      for all thread_i ∈ block_{j,k} do                    ▷ copy each base
3:          shared[i] ← R_j[i]
4:      end for

5:      Synchronize
6:      for all thread_i ∈ block_{j,k} do
7:          for n ∈ (A, G, C, T) do                          ▷ mutate
8:              tuple ← ϵ
9:              for q: 1 → tupleSize do
10:                 base ← (q = k)?n : shared[i + q]         ▷ compute ith tuple
11:                 tuple ← tuple + base                     ▷ append base
12:             end for
13:             L ← L ∪ tuple                                ▷ add to list
14:         end for
15:     end forreturn L
16: end procedure
```

Algorithm 4.2 describes the operations performed by a block of thread. A $block_{j,k}$ computes the mutation at jth position of kth read. Block indexing is used to enumerate all the sub-problems of mutation problem. At the start of computation, it is required to copy the read from GPU global memory to the shared memory so that the access time can be lowered for later memory read operation. As there are n threads,each thread takes part in the copy operation, and copies a single base to the shared memory.

After copying, ith thread calculates the tuple starting at ith position and also its reverse complement. This kernel makes use of look-up tables stored in constant memory to speedup

much of its computation. At the start, it calculates a mask based on the block column, to determine which position should be mutated in this block. This is done by shifting an existing mask of length n bytes by column index. This mask is stored in constant memory, so that each thread in every block can easily access it.

A thread then looks up the occurrence of computed tuple in T. If it is found and has occurrence of above some threshold (i.e. belong to solid set),

4.3.2 Accumulate

The execution of first kernel thus produces the l-mer weak - strong set membership information. The next kernel performs summation on the l-mer score to obtain read score for a given mutation. This kernel is launched with a grid of $m \times 4$ blocks, with each block having n threads as shown in Figure 4.3. Each thread in the block independently accumulates the tuple score.

Algorithm 4.3 Accumulate Mutation Score

Input:
M : mutations

Output: $Score$: Score for all mutations for all reads

```
1: procedure ACCUMULATE(M)
2:     for all thread_i ∈ block_{j,k} do
3:         for i ← 1 → n − t + 1 do                    ▷ mutate
4:             sum ← sum + M[i]
5:         end for
6:         Score[i][j] ← sum
7:     end for
8: end procedure
```

Threads perform their operation without collaborating with other threads in the same block. The choice of launch configuration is made by considering the wrap size (16) on the GPU devices, in order to maximize the device occupancy. A larger number of threads executing in parallel would result in better utilization of the gpu resources. Algorithm 4.3 is a simple representation of the operation performed by each thread while accumulating the mutation score.

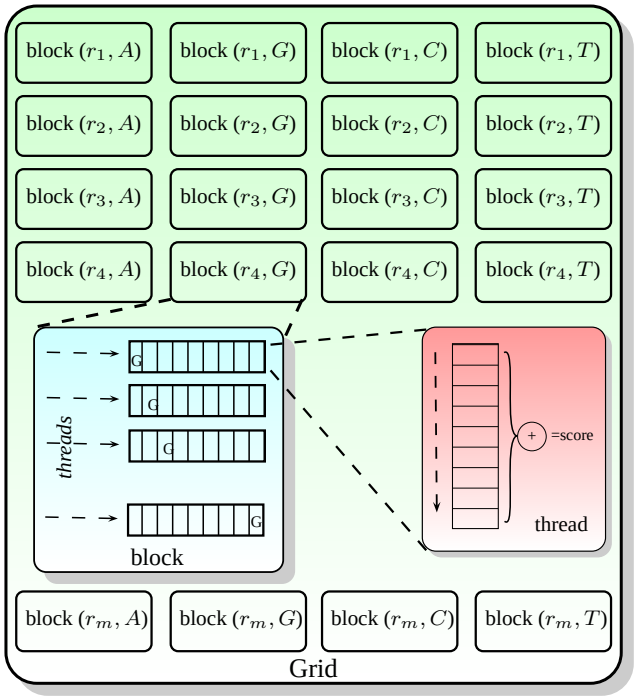

Figure 4.3: Accumulate Mutation Score

4.3.3 Substitution Selection

After the score has been calculated, another kernel is invoked to select the best substituted nucleotide for each position of a read. Each position in the read has four possible substitution. This kernel is invoked as grid of m blocks containing n threads. During the execution of this kernel, threads in the block work independently of one another. Each thread scan through the set of 4 values calculated in the previous steps, to select the mutation representing highest score. Figure 4.4 shows a schematic arrangement and processing of the grid, blocks and threads for this step.

4.3.4 Position Selection

As the last step of error correction, another kernel is invoked to select the best position for mutation depending on the values calculated in the previous step. This CUDA kernel determines which of the n position if mutated, can produce l-mers from strong set. It is

Algorithm 4.4 Substitution Selection

Input:
Scores : Mutation scores for all posibble mutations

Output:
Substitution: Best Substitution at every position of all reads

```
1:  procedure FindBestSubstitution(M)
2:      for all thread_i ∈ block_{j,k} do
3:          for i ← 1 → 4 do
4:              if max < Scores[j][k][i] then
5:                  max ← Scores[j][k][i]
6:                  pos ← i
7:              end if
8:          end for
9:          Substitution[i][j] ← (max, pos)
10:     end for
11:     return Substitution
12: end procedure
```

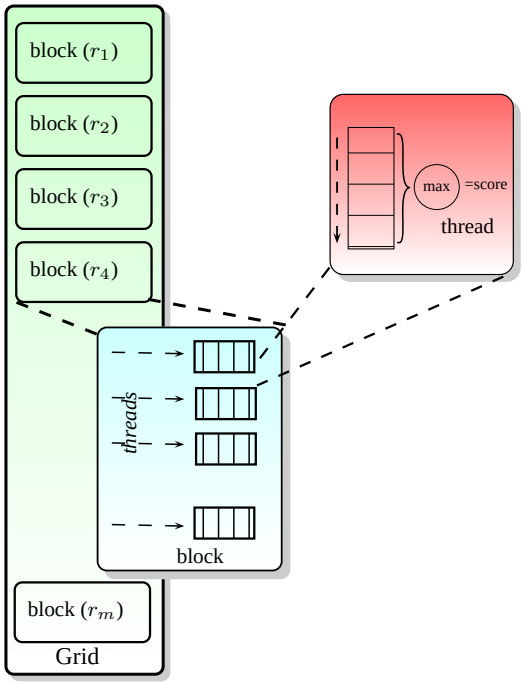

Figure 4.4: Substitution Selection

launch as grid of m blocks each containing single thread. Each thread scans n values in parallel and records the maximum score and its position as the output.

Algorithm 4.5 Position Selection

Input:
Substitution : Best Substitution for all positions of all reads

Output:
Positions : Best position for substitution for all reads.

```
1:  procedure FindBestPosition(Substitution)
2:      for all thread_i ∈ block_j do
3:          for i ← 1 → n do
4:              if max < Substitution[j][i] then
5:                  max ← Substitution[i]
6:              end if
7:          end for
8:          Positions[j] ← max
9:      end for
10: end procedure
```

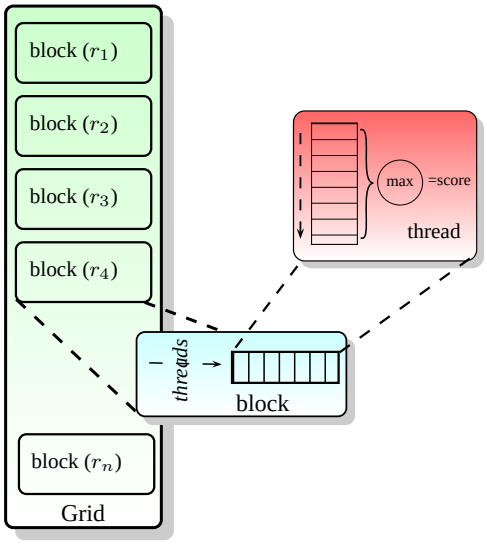

Figure 4.5: Position Selection

After computing the best mutations for all the reads in a batch, reads are transformed according to their mutation score. If the score is too low than the read is discarded, otherwise the mutation is applied to the input set of reads and placed in a buffer for output.

4.4 Time Complexity Analysis

The problem complexity of error correction in GPU Euler depends on the read length n and number of reads m. For every read, there are $4 \times n$ mutations and each mutation contributes $2 \times (n-t+1)$ tuples of length t. The number of tuples for a single read is thus bounded by $O((4 \times n) \times (2 \times (n-t+1)))$ or $O(n^2)$. For m reads, the total number of tuples is bound by $O(m.n^2)$. Therefore, the solution space is linear with respect to the number of reads m and expands quadratically with the length of read n. Time required to generate the $O(m.n^2)$ is also bounded by $O(m.n^2)$.

The search of best mutated reads requires computing the mutation score of every mutated read. Calculating the mutation score for a single read is bounded by $O(n)$ that will sum up all the n values to get a score. For all the $O(m.n)$ mutated reads, the computation time is bounded by $O(m.n.n) = O(m.n^2)$. Time required to select best substitution on any position of any read is bounded by $O(m.n)$ for all the reads. Hence total time required for error correction in a linear fashion is bounded by $O(m.n^2)$.

Due to the independent nature of the sub-problems, a simple approach for parallel computation can be implemented by performing the computation for each read in parallel with m processors can lead to $O(n^2)$ parallel time yielding a linear speed up $O(m)$ with respect to the number of processors.

GPU Euler employs similar technique but since the number of threads are limited on a CUDA device, therefore, m has to be divided into smaller batches, in order to process the complete read set. Furthermore, tuple computation of a read is also done in parallel, which leads to $n-t+1$ threads computing $n-t+1$ tuples concurrently. This parallel step considerably reduces the run-time for computing the complete tuple sets and bounds the computation by $O(\frac{m}{batchSize}.n)$ parallel time.

Table 4.2: Expected Error Correction on GPU-Euler for 36bp & 50bp reads

Genome	Expected U / M / D	Error Free	Erroneous	Error Free (%)
	36 bp			
CJ	596814 / 257437 / 57669	596814	315106	65.45%
LL	859860 / 370978 / 83362	859860	454340	65.43%
NM	793747 / 342360 / 77433	793747	419793	65.41%
	50 bp			
CJ	164013 / 233044 / 259523	164013	492567	24.98%
LL	236284 / 336012 / 373924	236284	709936	24.97%
NM	218415 / 310216 / 345129	218415	655345	25.00%

U, M and D denote Unmodified, Modified and Discarded decisions respectively.

4.5 Results

In order to evaluate the quality of error correction, we design an experiment using our three reference genome (CJ,LL,NM) and simulated error containing reads. Error simulation was performed using ART simulator [46], with Solexa error profile along with quality files. The ART simulator generated error containing reads of length 36 and 50 base pairs. The total set contained 6 error containing reads file (2 for each genome). GPU Euler is then invoked to perform error correction with tuple size 10-32 for these 6 samples. The ART simulator generated read files in FASTQ format which is then converted to FASTA format using Galaxy [47–49] , a web based tool for genome analysis.

Table 4.2 summarize the read quality by reporting the number of error free and erroneous reads across all the 6 samples. Reads of length 36 base pairs have approximately 65% error free reads while 50 base pairs long reads have approximately 25% error free reads. The Table also lists the expected number of decision (Unmodified, Modified and Discarded abbreviated as U, M and D) required to remove all the errors from the sample.

Table 4.3: Error Correction on GPU-Euler for 36bp reads

CJ

l	GPU Euler Output (%)		Correct Decision (%)		Error Free (%)
	Total Reads	U / M / D	Total	U / M / D	
10	911920	91.80 / 8.20 / 0.00	66.73	98.07 / 1.93 / 0.00	66.73
16	909330	65.59 / 34.13 / 0.28	92.15	70.22 / 29.55 / 0.24	92.20
22	888553	64.66 / 32.78 / 2.56	91.76	69.65 / 29.30 / 1.04	93.19
28	653485	45.76 / 25.90 / 28.34	64.73	68.53 / 25.88 / 5.59	85.28
32	227405	6.44 / 18.50 / 75.06	15.28	41.14 / 25.04 / 33.83	40.55

LL

l	GPU Euler Output (%)		Correct Decision (%)		Error Free (%)
	Total Reads	U / M / D	Total	U / M / D	
10	1314200	94.61 / 5.39 / 0.00	65.92	99.25 / 0.75 / 0.00	65.92
16	1310274	65.75 / 33.95 / 0.30	92.51	70.15 / 29.59 / 0.26	92.55
22	1281399	64.72 / 32.78 / 2.50	91.86	69.64 / 29.29 / 1.07	93.21
28	950281	46.74 / 25.56 / 27.69	65.96	68.71 / 25.80 / 5.49	86.21
32	345370	7.55 / 18.73 / 73.72	16.81	43.87 / 25.42 / 30.71	44.31

NM

l	GPU Euler Output (%)		Correct Decision (%)		Error Free (%)
	Total Reads	U / M / D	Total	U / M / D	
10	1213540	93.42 / 6.58 / 0.00	65.88	99.27 / 0.73 / 0.00	65.88
16	1209778	62.58 / 37.11 / 0.31	88.20	69.52 / 30.17 / 0.31	88.21
22	1184479	64.71 / 32.90 / 2.39	91.55	69.74 / 29.21 / 1.05	92.81
28	889563	47.83 / 25.47 / 26.70	67.33	68.86 / 25.72 / 5.42	86.88
32	349960	9.75 / 19.09 / 71.16	19.63	48.51 / 25.13 / 26.35	50.13

U, M and D denote Unmodified, Modified and Discarded decisions respectively.

4.5.1 Error Correction Performance

For any read, GPU Euler is expected to take either of three decision during error correction. If the read has all tuple from the strong set, then it should be left **unmodified (U)** else if the read can be mutated to have tuple as part of tuple spectrum, then it should be **modified (M)** accordingly, else if it cannot be modified, then it must be **discarded (D)**. It must be noted that error correction is a heuristic approach and it tries to remove as much error as it can during the execution, but it may incur some error due to false positive and false negatives. The decisions of error correction are based on the information collected from the input set of reads having erroneous read, as GPU Euler can only estimate the quality of a read based on its coverage, the resultant output might still contain incorrect reads. For

Table 4.4: Error Correction on GPU-Euler for 50bp reads

CJ

l	GPU Euler Output (%)		Correct Decision (%)		Error Free (%)
	Total Reads	U / M / D	Total	U / M / D	
10	656580	86.26 / 13.74 / 0.00	26.01	95.95 / 4.05 / 0.00	26.01
16	627927	24.25 / 71.38 / 4.36	63.34	38.00 / 55.55 / 6.45	61.96
22	579649	25.61 / 62.67 / 11.72	67.39	35.39 / 49.62 / 14.99	64.89
28	471041	22.69 / 49.06 / 28.26	67.67	30.05 / 41.67 / 28.29	67.65
32	332075	16.28 / 34.29 / 49.42	59.57	23.30 / 32.08 / 44.62	65.23

LL

l	GPU Euler Output (%)		Correct Decision (%)		Error Free (%)
	Total Reads	U / M / D	Total	U / M / D	
10	946220	93.48 / 6.52 / 0.00	25.09	98.96 / 1.04 / 0.00	25.09
16	902254	26.63 / 68.73 / 4.65	62.96	39.18 / 53.90 / 6.92	61.46
22	836588	12.42 / 76.00 / 11.59	56.44	20.93 / 60.82 / 18.25	52.19
28	685049	22.70 / 49.70 / 27.60	67.95	30.07 / 41.88 / 28.05	67.53
32	485114	16.66 / 34.61 / 48.73	60.09	23.72 / 32.32 / 43.96	65.68

NM

l	GPU Euler Output (%)		Correct Decision (%)		Error Free (%)
	Total Reads	U / M / D	Total	U / M / D	
10	873760	94.03 / 5.97 / 0.00	25.06	99.36 / 0.64 / 0.00	25.06
16	834255	26.20 / 69.27 / 4.52	62.34	39.07 / 54.09 / 6.83	60.83
22	772694	25.18 / 63.25 / 11.57	65.95	34.88 / 49.63 / 15.50	63.02
28	636684	23.89 / 48.97 / 27.13	68.89	31.07 / 41.15 / 27.78	68.28
32	461093	17.38 / 35.39 / 47.23	60.83	24.51 / 32.71 / 42.79	65.95

U, M and D denote Unmodified, Modified and Discarded decisions respectively.

instance, an incorrectly discarded read is a false negative, it should be part of the corrected reads after either as modified or unmodified read. On the other hand a read incorrectly left unmodified is a false positive, as it must only be part of the output with modification otherwise it should be discarded. In this implementation, GPU Euler modifies only those reads that require single modification. Also, in this implementation GPU Euler can not identify if it has false negatives and false positives, resulting into certain incorrect reads.

Table 4.3 and Table 4.4 summarizes the performance of error correction procedure for reads of length 36 and 50 base pairs, respectively. Error correction is performed on a simulated set of reads with varying tuple length l from 10 to 32. Tables 4.3 and 4.4 report total reads generated by GPU Euler as it performed the error correction along with the percentage of each decision (Unmodified, Modified and Discarded which are abbreviated as

U, M and D respectively in the Tables.) carried out by GPU Euler to generate those reads. For the complete process. The percentage of total correct decisions taken by GPU Euler is also reported along with the contribution of each decisions.

For 36 base pair reads, with tuple length equal to 10 , error correction does little to improve the percentage of error free reads providing more or same less similar number of error free reads (approximately 65%). This behaviour is explainable considering the fact that performing error correction with tuple length 10, generates smaller length tuples with high degree of multiplicity, thus directing the GPU Euler to conclude that most reads are already correct. As the length of tuple increases, GPU Euler receives better hints to guide the correction heuristics. With longer tuples, reads with less coverage can be identified quickly as potential candidate for correction (discard or modify). The increase tuple length contributes significantly towards the improvement of the ratio of error free reads which reaches up to approximately 93% (93.31% , 93.35% and 92.81% for CJ ,LL and NM respectively). It also is observed that maximum performance is achieved with tuple length at 18-22. An increase in tuple length beyond this range impacts the correction decision adversely. Longer tuples lies at the other extreme end of the spectrum which skews the spectrum to contain a smaller number of tuples. This situation leads GPU Euler to discard a lot of reads as they could not be mutated to match the spectrum, as evident by the increase in discard decision taken by GPU Euler for longer tuple length, which decrease the percentage of correct decisions.

Similar pattern is observed for read length of 50 base pairs where the best output is observed with tuple length between 28-30 improving the number of error free reads from 25% to 68% (68.09%, 67.53% and 68.28% for CJ, LL and NM respectively). The increase in read length increases the possible locations of error on a given reads. As GPU Euler only considers single position mutation to be fixable, it discards a large number of reads when processing longer reads, concluding that they are damaged beyond repair.

Figure 4.6 summarizes the result of error correction when performed on genome CJ with read length 36 base pairs. The three decisions taken by error correction for each tuple

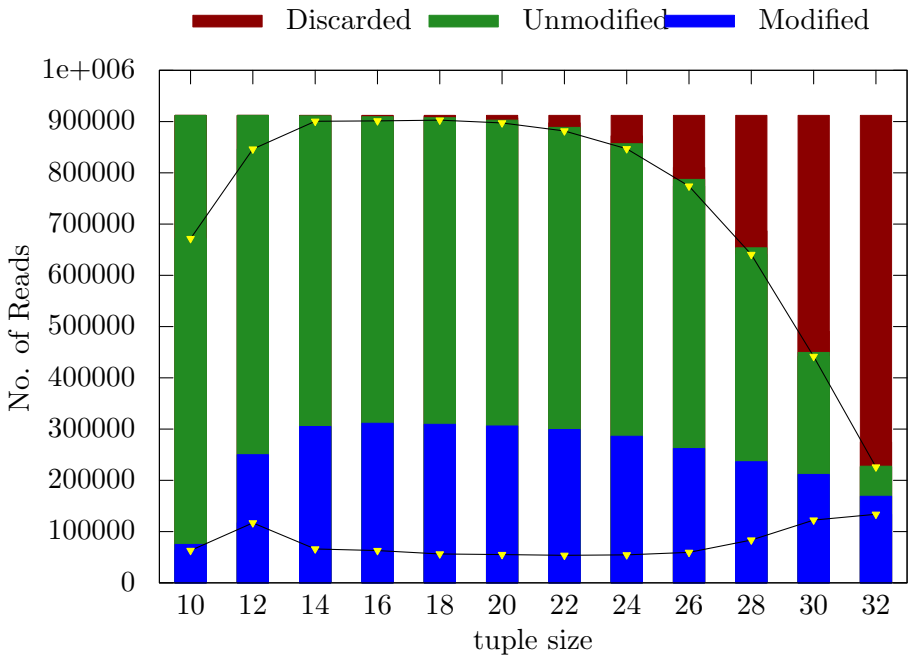

Figure 4.6: Correction Comparison (CJ 36bp)

length are shown in different colors. The number of correctly modified as well correctly left unmodified are also marked by two line such that the region between them represents the total number of correct decision taken by error correction procedure for various tuple length. For CJ 36 base pairs, the number of correct decision increased as the tuple length increase from 10 base pairs onwards. Error correction procedure reached its peak performance at 18-20 base pairs after which the number correct decisions continue to fall due to small number of strong tuples in the spectrum.

The number of error free reads for both 36 and 50 base pairs is summarized in Figure 4.7 where the percentage of error free reads is plotted against the increasing tuple sizes from 10 to 32 for all three genomes (CJ, LL, NM) and read lengths.

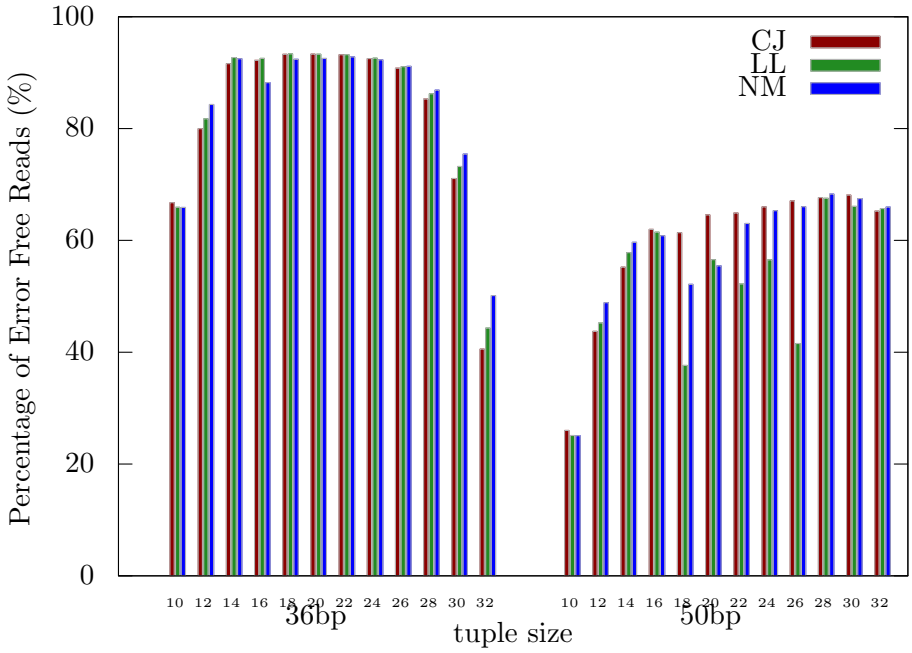

Figure 4.7: Error Free Reads (CJ, LL, NM)

4.5.2 Run-time Performance

We also collected profiling statistics to analyze the performance of different steps of the GPU based error correction. In Table 4.5, we report execution time in seconds for four steps of error correction (*Mutation*, *Accumulate*, *Substitution*, *Position*) when performed on CJ for 36 and 50 base pair long reads. The *Encoding* column reports the time that GPU Euler took to encode tuples into 64 bit representation as pre-processing step for error correction. The *Other* column reports the time spent in miscellaneous operations like moving data to GPU memory, modifying the reads according to error correction output, reading and writing to files, etc. The error correction procedures spent most of its time during enumerating all the mutations. The reason for such behaviour is that GPU Euler uses a very simple hash table construction, which stores all the keys in sorted buckets. A look-up for key-value performs a binary search in key's bucket. Since the hash table is kept in global memory and it heavily affects the overall performance by repeated access to the global memory. With a bucket of

Table 4.5: Run-time Profiling for GPU based Error Correction on CJ
(in seconds)

36 base pairs

t	Encoding	**Mutation**	Accumulate	Substitution	Position	Other	Total
12	0.378	**185.763**	1.633	0.042	0.105	6.785	194.706
14	0.378	**220.683**	1.514	0.041	0.105	8.911	231.632
16	0.378	**221.383**	1.386	0.041	0.105	11.636	234.929
18	0.385	**215.316**	1.258	0.043	0.105	12.119	229.225
20	0.382	**216.109**	1.131	0.041	0.105	11.922	229.69
22	0.378	**210.865**	1.006	0.041	0.105	11.26	223.654
24	0.393	**200.521**	0.878	0.041	0.105	10.463	212.401
26	0.377	**178.142**	0.749	0.041	0.105	9.563	188.977
28	0.378	**187.011**	0.625	0.041	0.105	8.522	196.682
30	0.378	**179.258**	0.501	0.041	0.105	7.137	187.421
32	0.377	**168.836**	0.372	0.042	0.105	5.581	175.314

50 base pairs

t	Encoding	**Mutation**	Accumulate	Substitution	Position	Other	Total
12	0.373	**331.242**	2.768	0.041	0.100	7.432	341.957
14	0.376	**473.586**	2.651	0.04	0.100	14.154	490.908
16	0.374	**452.676**	2.512	0.039	0.100	36.848	492.55
18	0.372	**673.473**	2.404	0.041	0.100	48.209	724.599
20	0.375	**847.962**	2.289	0.042	0.100	52.309	903.078
22	0.373	**863.035**	2.15	0.043	0.101	53.936	919.639
24	0.374	**852.272**	2.008	0.043	0.100	54.131	908.929
26	0.373	**854.301**	1.877	0.04	0.100	53.124	909.816
28	0.374	**759.304**	1.722	0.041	0.100	50.922	812.463
30	0.373	**656.773**	1.569	0.04	0.100	47.472	706.327
32	0.374	**708.347**	1.436	0.04	0.100	42.629	752.927

512 items, at most 9 steps would be required to fetch a value for given tuple determining if it belongs to strong tuple set or weak tuple set.

When the read length changes from 36 base pairs to 50 base pairs, increasing the number of mutations increases by 2 times at tuple size 12 to 5 times at tuple size 32. However the generation time only increased by 2 times. In most of the cases, execution time for mutation generation time exhibited linear increase with respect to number of mutations. At the extremes (12 and 32), different behaviour is observed. With small tuple size, mutation step performed closer to the lower bound of the binary search during hash value retrieval, since it smaller tuple sizes leads to large number of duplicate entries. While, on the other hand, large tuple size, requested to look up keys from a large set of possible values, mostly return

Table 4.6: Error Correction on GPU-Euler for 36bp CJ reads

l	GPU Euler			GPU Euler + EC			EulerSR		
	N50	Contigs	Avg. Len.	N50	Contigs	Avg. Len.	N50	Contigs	Avg. Len.
20	-	-	-	55	139815	45.986	6917	602	2776.38
22	35	609178	33.452	55	140945	47.635	2543	1131	1421.788
24	36	611384	35.152	56	142126	49.234	907	2553	605.568
26	36	615267	36.713	56	144113	50.655	351	4554	293.988
28	36	627501	37.979	56	148945	51.533	-	-	-
30	36	663076	38.623	52	164901	50.691	-	-	-
32	36	747500	38.305	45	219018	46.39	-	-	-

without any value. Thus it spent most of its time closer to the upper bound of the binary search.

4.5.3 Sequence Assembly with Error Correction

We also analyzed the affect of error correction on sequence assembly by comparing the N50 score of the assembly on error containing reads before and after the error correction. The sequence assembly does not currently employs a graph simplification step to reduce the complexity of graph and incorporate hints from the read set. We also compared the output from GPU Euler with result from EulerSR to assess the performance of error correction on sequence assembly. Table 4.6 summarizes the results of sequence assembly from GPU Euler, GPU Euler with error correction and EulerSR for simulated reads of length 36 from CJ. GPU Euler was first executed on the reads without any modification and than again using the output from the error correction phase. We selected the corrected reads obtained from error correction with tuple size 20 as it has highest percentage of error free reads.

Although the N50 score did improve when error correction is performed, the overall N50 score and mean length of the contigs remain very low compared to output from EulerSR. However, EulerSR was not able to produce output for tuple size beyond 26. GPU Euler does not incorporate graph transformation steps, which can provide significant hints to guide the assembly. Furthermore, even with the corrected reads, the read set still contains some erroneous reads which adds to the incorrect paths in the de-Bruijn graph.

4.6 Conclusion

In this chapter we presented a GPU based error correction scheme that was able to improve the number of error free reads from 65% to 93% for 36 bp long reads while for 50 bp long reads, it managed to improve the number from 25% to approximately 67%.

Chapter 5: Conclusion and Future Work

In this thesis a GPU based sequence assembly method is presented and evaluated for performance and accuracy. The assembly algorithm is based on Eulerian path sequence assembly where an euler path is constructed on tuple from read modeled as a de-Bruijn graph. The GPU implementation uses a modified parallel euler tour algorithm and runs on CUDA devices and was evaluated by using a set of error free reads. The result proved to be highly motivating in terms of run-time performance as well as assembly outcome. Using EulerSR as a benchmark to compare the assembly outcome, GPU Euler assembly produced contigs of comparable size and quality. GPU Euler surpassed EulerSR in terms of execution and contig length statistics while leaving room for improvement regarding accuracy of the assembled contigs.

A gpu based error correction method is also proposed to augment the sequence assembler. The error correction procedure used Spectral Alignment as a basis for correcting error in the reads. Error correction performance was evaluated on different aspects using simulated reads of 36 and 50 base pairs. Error correction was able to improve the number of error free reads from 65% to 93% for reads 36 base pair long, while for 50 base pair long with initially 25% error reads were improved to have 67% error free reads. The number of correct decsion taken by the procedure varied with the tuple length and reached its maximum at tuple length 20 for 36 base long reads and 26-28 for 50 bases long reads. The error correction routine faced challenges for the execution time performance due to inefficient hash table implementation.

The contribution of this thesis for sequence assembly are presented in the paper title "GPU-Euler : Sequence Assembly using GPGPU" [10] which was published by IEEE for HPCC'11.

5.1 Future Work

In future, there are primarily two direction that can provide considerable improvements as discussed below

5.1.1 Correctness

GPU Euler provides considerable opportunity to improve upon the correctness and accuracy. Parallel Graph Transformation is one key which can significantly improve the assembly accuracy for both error free and error-containing reads, as it incorporated more information from the read set into the graph. Not only it adds valuable information to retrieve a path closer to the correct assembly, it also reduces the graph complexity by shrinking the edges and splitting vertices with multiple in-degree and out-degree.

There is also some room for improvement in error correction procedure. Error correction routines can be enhanced to consider more than one substitution while computing the mutations. Furthermore, it should be able to compensate for the insertion and deletion of the bases as well. Allowing multiple substitution can reduce the reads that gets discarded as they appear to have more than one error and error correction routine can not compute all the combination with 2 or more substitution.

5.1.2 Performance

An efficient GPU based hash table implementation will be able to facilitate various task in the assembly pipeline. It will prove an essential improvement on the performance of GPU Euler. A hash table that can support efficient concurrent read from multiple threads will improve the runtime statistics quite significantly. Error correction procedure will benefit most from the improved hash table implementation.

As of now, GPU Euler does not utilize shared memory and texture memory to their fullest extent. Moving look-ups to texture memory and splitting problem so that it can use shared memory , will provide good improvements on the execution time.

There is also growing need to split the graph problem into smaller chunks of work item,

so that larger genomes can be assembled using GPU Euler. This would require re-iterating the graph problem such that sub-problem can be easily isolated from a large graph.

There is a rapid development going on for CUDA architecture and tool kit, which exposes several new feature in each revisions. Important feature like atomic operation on shared memory and memory allocation inside a kernel can provide a new view to look upon the solution and would definitely lead to important performance enhancements.

Appendix A: Compute Unified Device Architecture

nVidia CUDA is a framework for exploiting nVidia based graphics card for general purpose computing. It brings an opportunity to harness the massive parallelism present in the design of the graphics card. This appendix presents a summary for the architecture and programming model of CUDA.

A.1 Hardware Architecture

Compute Unified Device Architecture (CUDA) is the culmination of efforts spanning over decades in hardware and software design. GPU hardware is primarily targeted for high throughput graphic operation. Figure A.1 present a schematic representation of CUDA supported device architecture. Important aspects of CUDA hardware design are discussed below.

A.1.1 Streaming MultiProcessors

CUDA devices have several streaming multiprocessors (SM) and each streaming multiprocessor has several ALU which are known as scalar processors (SP) or CUDA cores. The number of streaming multiprocessors and scalar processors depends on the architecture generation of the CUDA devices. Earlier generation CUDA devices had 8 scalars processors while the latest generation offers upto 32 scalar processors. Each block in CUDA kernel invocation is executed by a single SM which assigns each thread in the block to one SP. Streaming Multiprocessor have very fast on-chip memory known as *shared memory* which is shared among all its SPs. Apart from shared memory each SM also has a register file and local memory. The register file and local memory is split among all the executing threads, therefore it is important keep the CUDA kernel as small as possible, otherwise it may fail to launch due to unavailability of the resources.

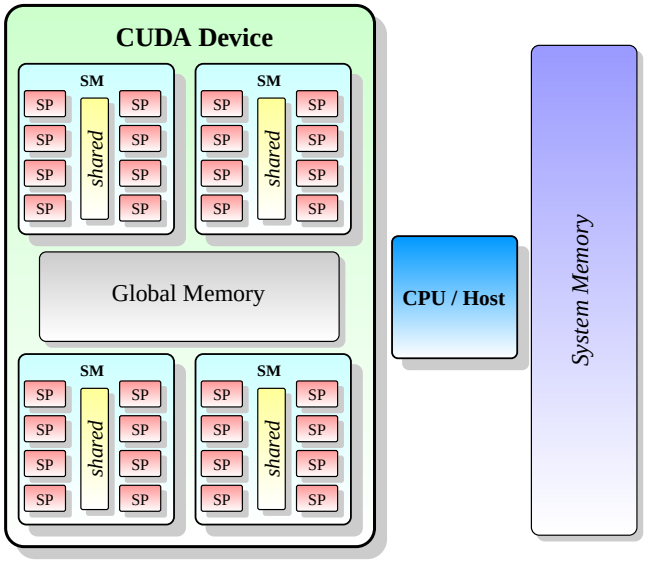

Figure A.1: CUDA Hardware Architecture

A.1.2 Parallel Thread Execution Environment (PTX)

CUDA instruction set architecture (ISA) is known as Parallel Thread Execution environment (PTX) which exposes the underlying GPU as data-parallel computing device. The major advantage of PTX is that it offers a standard representation of the instruction for all CUDA devices. The PTX instructions are then converted to the native GPU specific instructions, allowing applications to take advantage of the newer hardware without rewriting the application from scratch.

PTX offers the underlying parallelism of the CUDA devices as Cooperative Thread Arrays(CTA). Each CTA is represented by a CUDA thread-block. CTAs can be arranged as into a grid depending upon the problem at hand. Threads in a CTA can communicate with each other through shared memory and block level synchronization operations. Threads in different CTAs can only communicate through global memory. A CTA can be arranged as 1D , 2D or 3D array of threads. Similarly CTAs can be arranged into 1D, 2D or 3D grids of blocks. These arrangements provide a way to decompose the problem into sub-problem and exploit it for concurrent solution.

A.1.3 Memory Hierarchy

Memory Hierarchy is one of the most important aspect while designing solution for any given problem. CUDA devices offer different types, each with its own benefits and short comings. The hierarchy allows to opt for trade off between size and latency. As the memory latency decrease, the size of available memory shrinks as well.

Global Memory

Global Memory is off-chip memory with the highest latency. It is accessible from all the threads in every block and supports atomic instructions. This global memory is also accessible to the Host (CPU) which uploads data required by the computation from System Memory to GPU global memory. Writes to an address in global memory are visible to all threads after kernel level synchronizations.

Shared Memory

Shared Memory is on-chip memory with approximately 100 times lower latency than the global memory. It is present on each SM and can only be shared within a block of threads. Writes to shared memory location are visible after thread level synchronization.

Register File

Register file is an on-chip very low latency memory. It is a set of 32 bit register which is split among the threads of a block. The actual number of registers available on each CUDA devices depends on architecture generation. Register are local to each thread are used to hold thread specific computations.

Constant

Constant provide a fast on-chip read-only memory, which can be accessed from all the threads.

Texture Memory

Texture Memory is a fast cached memory that can be used to implement fast look-up

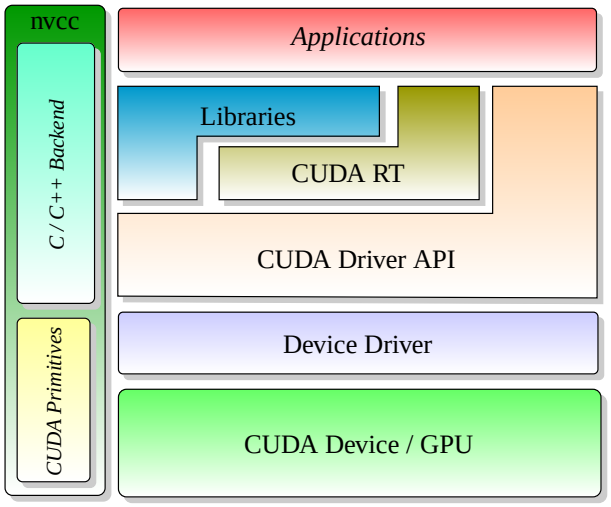

Figure A.2: CUDA Software Stack

tables.

A.2 Software Stack

CUDA applications require a supporting software stack in order to exploit the computing capabilities of the underlying GPU. Figure A.2 presents a conceptual view of the CUDA software stack and how different components of the CUDA eco-system interact each other. There two main components of CUDA software stack as discussed below:

A.2.1 CUDA driver

CUDA device drivers are special drivers for CUDA enabled GPUs that exposes computing capabilities to the applications executed on a CUDA capable system. nVidia maintains two set of drivers for their GPUs. The standard driver which only provide GPU specific interface without exposing CUDA capabilities and CUDA enabled drivers which enables CUDA functionality and provides access to the GPU as a computing device. CUDA enabled drivers exposes a low-level API to program the CUDA devices at a discrete level. However, most applications rarely use CUDA driver directly and rely on CUDA run-time library

A.2.2 CUDA run-time library

CUDA run-time library is set of APIs built over CUDA driver to provide a more coarse-grained functions.

A.3 GPU Application Development

A.3.1 Compiler Extension

CUDA provides an extended C/C++ front-end compiler to build applications for CUDA. CUDA has introduced few new keywords and construct in order to reflect the PTX at higher level. Interestingly the language additions are kept to minimum, so as decrease the learning curve. The new keywords include (__sync, __shared__ , __global__, __device__ ..etc)

A.3.2 Launch Configuration

Kernel launch configurations are central to CUDA application development as it defines the problem decomposition. It describes the arrangement of threads into blocks and blocks into grid. Each thread executes the same kernel code, and interacts with other other in the same block. A grouping at the block level is not the same as the grouping at grid level, since threads grouped in a block share a low latency memory while thread across blocks only share global GPU memory. The first generation of CUDA devices support .

Consider an example of matrix addition of order $m \times n$ on GPU. The solution requires $m \times n$ addition operation which can be performed in parallel given enough parallel threads leading to $O(1)$ run-time with $m \times n$ speedup. Considering this parallelism, each addition operation can be mapped to a single thread. There can be multiple ways to specify the launch configuration. A simple configuration would be to have 1 block of $m \times n$ threads. Alternatively, we can launch $m \times n$ blocks with only one thread. Since GPUs have limited resource to of

Deciding for a prefect launch configuration can become tricky. In order to decide which launch configuration would yield higher throughput, depends on factors such as data sharing

between decomposed sub-problems and the underlying hardware.

A.3.3 Programming Techniques

This section presents some programming techniques which are frequently employed during CUDA based GPU application development.

Branch Avoidance

Conditional branch operations have always been considered as the most celebrated nemesis not only for the optimizing compilers but for the underlying hardware as well. Therefore CUDA devices also get benefit from straight line code while code with conditional impose serious performance issues. For straight line code all threads of block perform same operation, improving the instruction throughput. In case of conditional execution, threads which are not on the branching path performs a no-op and wait for the branching code to converge to the main execution path. For case with more than one branches, threads are grouped by their evaluation of the condition expression, and then executed separately.

Therefore it is often desired to write code without using conditional statements. Such code pattern may allow extra values to be computed by the block , but masks their writes to only valid values. For instance, a block of m threads computes n values and if $n < m$, it implies at least $m - n$ will be not be required to participate in the computation. Instead of guarding the situation with conditions, it is usually more feasible to compute values for $m - n$ threads and mask them during the write by writing at some sink location.

Data Representation

CUDA devices can take advantage of the memory access pattern through coalesced memory access if the requested addresses refer to consecutive locations. In order to exploit this feature, all data objects must be aligned to word boundary. Another technique which is used frequently for avoid non-coalesced access, is that instead of representing a list of objects as an array of object structure(*Array of Structure*), each attribute can be stored in

a separate arrays containing values for single attributes from all the objects, transform the array of structure into an structure of arrays. This provide additional benefits of loading only the required attributes into the global memory.

Appendix B: Source Code

B.1 de-Bruijn Graph Construction

Listing B.1: Count Edges

```
__global__ void
debruijnCount(
              KEY_PTR lmerKeys , /* lmer keys    */
              VALUE_PTR lmerValues , /* lmer frequency */
              unsigned int lmerCount , /* total lmers */
              KEY_PTR TK , /* Keys' pointer for Hash table*/
              VALUE_PTR TV , /* Value pointer for Hash table*/
              unsigned int * bucketSize , /* bucketSize: size of each bucket */
              unsigned int bucketCount , /* total buckets */
              unsigned int * lcount , /* leaving edge count array : OUT */
              unsigned int * ecount , /* entering edge count array: OUT */
              KEY_T validBitMask /* bit mask for K length encoded bits*/
              )
{
  unsigned int tid =
                     (blockDim.x * blockDim.y * gridDim.x * blockIdx.y)
                   + (blockDim.x * blockDim.y * blockIdx.x)
                   + (blockDim.x * threadIdx.y)
                   + threadIdx.x;

  if (tid < lmerCount)
    {
      KEY_T lmer = lmerKeys[tid];
      VALUE_T lmerValue = lmerValues[tid];
      KEY_T prefix = (lmer & (validBitMask << 2)) >> 2;
```

```
26        KEY_T suffix = (lmer & validBitMask);
27
28        KEY_T lomask =  3;                // 2 bit Mask
29        VALUE_T prefixIndex =
30                                getHashValue(
31                                             prefix,
32                                             TK,
33                                             TV,
34                                             bucketSize,
35                                             bucketCount);
36        VALUE_T suffixIndex =
37                                getHashValue(
38                                             suffix,
39                                             TK,
40                                             TV,
41                                             bucketSize,
42                                             bucketCount);
43        KEY_T transitionTo   = (lmer & lomask);
44        KEY_T transitionFrom = ((lmer >> __popcll(validBitMask)) & lomask);
45
46        lcount[(prefixIndex << 2) + transitionTo]   = lmerValue;
47        ecount[(suffixIndex << 2) + transitionFrom] = lmerValue;
48     }
49  }
```

Listing B.2: Setup Vertices

```
1  __global__ void
2  setupVertices(
3                KEY_PTR kmerKeys ,
4                unsigned int kmerCount ,
5                KEY_PTR TK ,
6                VALUE_PTR TV ,
7                unsigned int * bucketSeed ,
8                unsigned int bucketCount ,
9                EulerVertex * ev ,
10               unsigned int * lcount ,
11               unsigned int * loffset ,
12               unsigned int * ecount ,
13               unsigned int * eoffset)
14 {
15
16   unsigned int tid =
17                       (blockDim.x * blockDim.y * gridDim.x * blockIdx.y)
18                       + (blockDim.x * blockDim.y * blockIdx.x)
19                       + (blockDim.x * threadIdx.y)
20                       + threadIdx.x;
21
22   if (tid < kmerCount)
23     {
24
25       KEY_T key = kmerKeys[tid];
26
27       VALUE_T index =
28                        getHashValue(
29                                     key,
```

```
30                                      TK,
31                                      TV,
32                                      bucketSeed,
33                                      bucketCount);
34
35        ev[index].vid = key;
36
37        //set up leaving edge information
38        ev[index].lp      =
39                                 loffset[(index << 2)];
40        ev[index].lcount  =
41                                 lcount[(index << 2)]
42                                + lcount[(index << 2) + 1]
43                                + lcount[(index << 2) + 2]
44                                + lcount[(index << 2) + 3];
45
46        //set up entering edge information
47        ev[index].ep      =
48                                 eoffset[(index << 2)];
49        ev[index].ecount  =
50                                 ecount[(index << 2)]
51                                + ecount[(index << 2) + 1]
52                                + ecount[(index << 2) + 2]
53                                + ecount[(index << 2) + 3];
54     }
55  }
```

Listing B.3: Setup Edges

```
__global__ void
setupEdges(
           KEY_PTR lmerKeys ,
           VALUE_PTR lmerValues ,
           unsigned int * lmerOffsets ,
           const unsigned int lmerCount ,
           KEY_PTR TK ,
           VALUE_PTR TV ,
           unsigned int * bucketSize ,
           const unsigned int bucketCount ,
           unsigned int * l ,
           unsigned int * e ,
           EulerEdge * ee ,
           unsigned int * loffsets ,
           unsigned int * eoffsets ,
           const KEY_T validBitMask)
{
  unsigned int tid =
                     (blockDim.x * blockDim.y * gridDim.x * blockIdx.y)
                   + (blockDim.x * blockDim.y * blockIdx.x)
                   + (blockDim.x * threadIdx.y)
                   + threadIdx.x;
  if (tid < lmerCount)
    {
      KEY_T lmer = lmerKeys[tid];
      VALUE_T lmerValue = lmerValues[tid];
      KEY_T prefix = (lmer & (validBitMask << 2)) >> 2;
      KEY_T suffix = (lmer & validBitMask);
      KEY_T lomask = 3;
```

```
30
31        VALUE_T prefixIndex =
32                              getHashValue(
33                                           prefix,
34                                           TK,
35                                           TV,
36                                           bucketSize,
37                                           bucketCount);
38        VALUE_T suffixIndex =
39                              getHashValue(
40                                           suffix,
41                                           TK,
42                                           TV,
43                                           bucketSize,
44                                           bucketCount);
45
46        KEY_T transitionTo = (lmer & lomask);
47        KEY_T transitionFrom = ((lmer >> __popcll(validBitMask)) & lomask);
48
49        unsigned int loffset = loffsets[(prefixIndex << 2) + transitionTo];
50        unsigned int eoffset = eoffsets[(suffixIndex << 2) + transitionFrom];
51
52        unsigned int lmerOffset = lmerOffsets[tid];
53
54        for (unsigned int i = 0; i < lmerValue; i++)
55          {
56
57            ee[lmerOffset].eid = lmerOffset;
58            ee[lmerOffset].v1  = prefixIndex;
59            ee[lmerOffset].v2  = suffixIndex;
60            ee[lmerOffset].s   = lmerValues[lmerCount - 1]
```

```
61                              + lmerOffsets[lmerCount - 1];
62
63            l[loffset] = lmerOffset;
64            e[eoffset] = lmerOffset;
65            loffset++;
66            eoffset++;
67            lmerOffset++;
68          }
69      }
70  }
```

B.2 Euler Tour

Listing B.4: Assign Successor

```
1  __global__ void
2  assignSuccessor(
3                  EulerVertex * ev ,
4                  unsigned int * l ,
5                  unsigned int * e ,
6                  unsigned vcount ,
7                  EulerEdge * ee ,
8                  unsigned int ecount)
9  {
10   unsigned int tid =
11                      (blockDim.x * blockDim.y * gridDim.x * blockIdx.y)
12                    + (blockDim.x * blockDim.y * blockIdx.x)
13                    + (blockDim.x * threadIdx.y) + threadIdx.x;
14   unsigned int eidx = 0;
15   if (tid < vcount)
16     {
```

```
17        while (eidx < ev[tid].ecount && eidx < ev[tid].lcount)
18          {
19            ee[e[ev[tid].ep + eidx]].s = l[ev[tid].lp + eidx];
20            eidx++;
21          }
22      }
23  }
```

Listing B.5: Construct Edge Graph

```
1   __global__ void
2   constructSuccessorGraph_Step1(
3                                 EulerEdge* e ,
4                                 Vertex * v ,
5                                 unsigned int ecount)
6   {
7     unsigned int tid =
8                          (blockDim.x * blockDim.y * gridDim.x * blockIdx.y)
9                        + (blockDim.x * blockDim.y * blockIdx.x)
10                       + (blockDim.x * threadIdx.y)
11                       + threadIdx.x;
12    if (tid < ecount)
13      {
14        v[tid].n1  = ecount;
15        v[tid].n2  = ecount;
16        v[tid].vid = e[tid].eid;
17        v[tid].n1  = e[tid].s;
18      }
19  }
20
21  __global__ void
22  constructSuccessorGraph_Step2(
```

```
23                                EulerEdge* e ,
24                                Vertex * v ,
25                                unsigned int ecount)
26 {
27   unsigned int tid =
28                      (blockDim.x * blockDim.y * gridDim.x * blockIdx.y)
29                    + (blockDim.x * blockDim.y * blockIdx.x)
30                    + (blockDim.x * threadIdx.y)
31                    + threadIdx.x;
32   if (tid < ecount)
33     {
34       if (v[tid].n1 < ecount)
35         {
36           v[v[tid].n1].n2 = v[tid].vid;
37         }
38     }
39 }
```

Listing B.6: Construct Circuit Graph

```
1  __global__ void
2  calculateCircuitGraphVertexData(
3                                  unsigned int * D ,
4                                  unsigned int * C ,
5                                  unsigned int ecount)
6  {
7
8    unsigned int tid =
9                      (blockDim.x * blockDim.y * gridDim.x * blockIdx.y)
10                   + (blockDim.x * blockDim.y * blockIdx.x)
11                   + (blockDim.x * threadIdx.y) + threadIdx.x;
12   if (tid < ecount)
```

```
13      {
14        unsigned int c = D[tid];
15        atomicExch( C + c, 1);
16      }
17  }
18
19  __global__ void
20  constructCircuitGraphVertex(
21                              unsigned int * C ,
22                              unsigned int * offset ,
23                              unsigned int ecount ,
24                              unsigned int * cv ,
25                              unsigned int cvCount)
26  {
27    unsigned int tid =
28                        (blockDim.x * blockDim.y * gridDim.x * blockIdx.y)
29                      + (blockDim.x * blockDim.y * blockIdx.x)
30                      + (blockDim.x * threadIdx.y)
31                      + threadIdx.x;
32    if (tid < ecount)
33      {
34        if (C[tid] != 0)
35          {
36            cv[offset[tid]] = tid;
37          }
38      }
39  }
40
41  __global__ void
42  calculateCircuitGraphEdgeData(
43                                EulerVertex* v ,
```

```
44                                unsigned int * e ,
45                                unsigned vCount ,
46                                unsigned int * D ,
47                                unsigned int * map ,
48                                unsigned int ecount ,
49                                unsigned int * cedgeCount)
50 {
51
52   unsigned int tid =
53                       (blockDim.x * blockDim.y * gridDim.x * blockIdx.y)
54                     + (blockDim.x * blockDim.y * blockIdx.x)
55                     + (blockDim.x * threadIdx.y)
56                     + threadIdx.x;
57   unsigned int index = 0;
58   unsigned int maxIndex = 0;
59   index = 0;
60   maxIndex = 0;
61   if (tid < vCount && v[tid].ecount > 0)
62     {
63       index = v[tid].ep;
64       maxIndex = index + v[tid].ecount - 1;
65       while (index < maxIndex)
66         {
67           unsigned int c1 = map[D[e[index]]];
68           unsigned int c2 = map[D[e[index + 1]]];
69           if (c1 != c2)
70             {
71               unsigned int c = min( c1, c2);
72               atomicInc( cedgeCount + c, ecount);
73             }
74
```

```
75            index++;
76          }
77      }
78
79  }
80  __global__ void
81  assignCircuitGraphEdgeData(
82                              EulerVertex* v ,
83                              unsigned int * e ,
84                              unsigned vCount ,
85                              unsigned int * D ,
86                              unsigned int * map ,
87                              unsigned int ecount ,
88                              unsigned int * cedgeOffset ,
89                              unsigned int * cedgeCount ,
90                              unsigned int cvCount ,
91                              CircuitEdge * cedge ,
92                              unsigned int cecount)
93  {
94
95    unsigned int tid =
96                         (blockDim.x * blockDim.y * gridDim.x * blockIdx.y)
97                       + (blockDim.x * blockDim.y * blockIdx.x)
98                       + (blockDim.x * threadIdx.y)
99                       + threadIdx.x;
100   unsigned int index = 0;
101   unsigned int maxIndex = 0;
102   if (tid < vCount && v[tid].ecount > 0)
103     {
104       index = v[tid].ep;
105       maxIndex = index + v[tid].ecount - 1;
```

```
106       while (index < maxIndex)
107         {
108           unsigned int c1 = map[D[e[index]]];
109           unsigned int c2 = map[D[e[index + 1]]];
110           if (c1 != c2)
111             {
112               unsigned int c = min( c1, c2);
113               unsigned int t = max( c1, c2);
114               unsigned int i = atomicDec( cedgeCount + c, ecount);
115               i = i - 1;
116               cedge[cedgeOffset[c] + i].c1 = c;
117               cedge[cedgeOffset[c] + i].c2 = t;
118               cedge[cedgeOffset[c] + i].e1 = e[index];
119               cedge[cedgeOffset[c] + i].e2 = e[index + 1];
120             }
121           index++;
122         }
123     }
124 }
```

Listing B.7: Execute Swipe

```
1  __global__ void
2  executeSwipe(
3               EulerVertex * ev ,
4               unsigned int * e ,
5               unsigned int vcount ,
6               EulerEdge * ee ,
7               unsigned int * mark ,
8               unsigned int ecount)
9  {
10   unsigned int tid =
```

```
11                      (blockDim.x * blockDim.y * gridDim.x * blockIdx.y)
12                    + (blockDim.x * blockDim.y * blockIdx.x)
13                    + (blockDim.x * threadIdx.y)
14                    + threadIdx.x;
15   unsigned int t;
16   unsigned int index = 0;
17   unsigned int maxIndex;
18   unsigned int s;
19   if (tid < vcount)
20     {
21       index = ev[tid].ep;
22       maxIndex = index + ev[tid].ecount - 1;
23       while (index < maxIndex){
24           if (mark[ee[e[index]].eid] == 1)
25             {
26               t = index;
27               s = ee[e[index]].s;
28               while (mark[ee[e[index]].eid] == 1 && index < maxIndex)
29                 {
30                   ee[e[index]].s = ee[e[index + 1]].s;
31                   index = index + 1;
32                 }
33               if (t != index){
34                   ee[e[index]].s = s;
35                 }
36             }
37           index++;
38         }
39     }
40 }
```

B.3 Error Correction

Listing B.8: Compute Mutation

```
1  __global__ void
2  calculateMutationScores(
3                          char * in_read ,
4                          char * mutationMask ,
5                          unsigned int readLength ,
6                          unsigned int lmerLength ,
7                          unsigned int M ,
8                          KEY_PTR TK ,
9                          VALUE_PTR TV ,
10                         unsigned int * bucketSize ,
11                         unsigned int bucketCount ,
12
13                         unsigned int* mutation , /*out param*/
14                         unsigned int * buffer)   /*out param*/
15 {
16   extern __shared__  char read[];        /* shared variable for read data*/
17   char * mask =  read + readLength + 31; /* shared variable for mask */
18
19   VALUE_T statusF;
20   VALUE_T statusR;
21   volatile KEY_T lmer = 0;
22
23   /*convert to base4*/
24   read[threadIdx.x] =  base4[
25                            in_read[blockIdx.x * readLength + threadIdx.x]
26                           & 0x07
27                            ];
28   /*apply mutation mask*/
```

```
29    mask[threadIdx.x] = mutationMask[
30                                (threadIdx.x + readLength - blockIdx.y)
31                              % readLength
32                                      ];
33
34    __syncthreads();
35
36    for (unsigned int k = 0; k < 4; k++)
37      {
38        lmer = 0;
39  #pragma unroll 4                //loop unroll
40        for (unsigned int i = 0; i < 8; i++)
41          {
42
43            lmer = (lmer << 8) |
44                ((KEY_T)(
45                        shifter[mutator[mask[threadIdx.x + i * 4]
46                                        | read[threadIdx.x + i * 4]][k]][3]
47                      | shifter[mutator[mask[threadIdx.x + i * 4 + 1]
48                                        | read[threadIdx.x + i * 4 + 1]][k]][2]
49                      | shifter[mutator[mask[threadIdx.x + i * 4 + 2]
50                                        | read[threadIdx.x + i * 4 + 2]][k]][1]
51                      | mutator[mask[threadIdx.x + i * 4 + 3]
52                                        | read[threadIdx.x + i * 4 + 3]][k]));
53          }
54
55        lmer = (lmer >> ((32 - lmerLength) << 1)) & lmerMask[lmerLength - 1];
56        statusF =
57                  getHashValue(
58                                lmer,
59                                TK,
```

```
60                              TV,
61                              bucketSize,
62                              bucketCount);
63        statusF = ((statusF == MAX_INT) ? 0 : ((statusF >= M)) ? 1 : 0);
64
65        /*compute reverse*/
66        lmer = 0;
67  #pragma unroll 4        //loop unroll
68        for (int i = 7; i >= 0; i--)
69          {
70            lmer = (lmer << 8) |
71                ((KEY_T)
72                   shifter[reverse[mutator[mask[threadIdx.x + i * 4 + 3]
73                                   | read[threadIdx.x + i * 4 + 3]][k]]][3]
74                 | shifter[reverse[mutator[mask[threadIdx.x + i * 4 + 2]
75                                   | read[threadIdx.x + i * 4 + 2]][k]]][2]
76                 | shifter[reverse[mutator[mask[threadIdx.x + i * 4 + 1]
77                                   | read[threadIdx.x + i * 4 + 1]][k]]][1]
78                 | reverse[mutator[mask[threadIdx.x + i * 4]
79                                   | read[threadIdx.x + i * 4]][k]]);
80          }
81
82        lmer = (lmer) & lmerMask[lmerLength - 1];
83
84        statusR =
85                  getHashValue(
86                              lmer,
87                              TK,
88                              TV,
89                              bucketSize,
90                              bucketCount);
```

```
91        statusR = ((statusR == MAX_INT) ? 0 : ((statusR >= M)) ? 1 : 0);
92
93        //increment solidCount
94        mutation[blockIdx.x * (readLength * NA_COUNT * readLength)
95                 + blockIdx.y * readLength * NA_COUNT
96                 + k * readLength
97                 + threadIdx.x]
98                              =      (unsigned int) (statusF + statusR);
99
100     }
101
102 }
```

Listing B.9: Accumulate Mutation Score

```
1  __global__ void
2  accumulate(
3              unsigned int * mutation ,
4              unsigned int readLength ,
5              unsigned int l ,
6              /*out*/
7              unsigned int * mutationScore)
8  {
9    unsigned int sum = 0;
10   for (int i = 0; i < readLength - l + 1; i++)
11     {
12       sum =   sum
13             + mutation[  blockIdx.y * (readLength * NA_COUNT * readLength)
14                        + blockIdx.x * readLength
15                        + threadIdx.x * readLength * NA_COUNT
16                        + i];
17     }
```

```
18    mutationScore[  blockIdx.y * (NA_COUNT * readLength)
19                  + blockIdx.x
20                  + threadIdx.x * NA_COUNT] =  sum;
21  }
```

Listing B.10: Select Best Substitution

```
1   __global__ void
2   bestMutation2(
3                 unsigned int * mutationScore ,
4                 unsigned int readLength ,
5                 unsigned int l ,
6                 /*out*/
7                 unsigned int * mutationStep)
8   {
9     unsigned int bestIdx = 0;
10    unsigned int bestScore = mutationScore[
11                                           blockIdx.x * readLength * NA_COUNT
12                                         + threadIdx.x * NA_COUNT];
13    unsigned int newScore = 0;
14    for (int i = 1; i < NA_COUNT; i++)
15      {
16        newScore =
17                    mutationScore[   blockIdx.x * readLength * NA_COUNT
18                                   + threadIdx.x * NA_COUNT
19                                   + i];
20        bestIdx   = bestScore < newScore ? i : bestIdx;
21        bestScore = bestScore < newScore ? newScore : bestScore;
22      }
23    mutationStep[  blockIdx.x * readLength
24                         + threadIdx.x] = (bestIdx << 16 | bestScore);
25  }
```

Listing B.11: Select Best Position

```
__global__ void
bestFinalMutation(
                  unsigned int * mutationStep ,
                  unsigned int readLength ,
                  unsigned int l ,
                  /*out*/
                  unsigned int * bestMutationPos ,
                  unsigned int * bestMutationIdx)
{
  unsigned int newScoreValue = mutationStep[blockIdx.x * readLength];
  unsigned int bestScore = (newScoreValue & 0x0000FFFF);
  unsigned int bestIdx = newScoreValue >> 16;
  unsigned int bestPos = 0;
  unsigned int newScore;
  for (int i = 1; i < readLength; i++)
    {
      newScoreValue = mutationStep[blockIdx.x * readLength + i];
      newScore = (newScoreValue & 0x0000FFFF);
      bestIdx = bestScore < newScore ? (newScoreValue >> 16) : bestIdx;
      bestPos = bestScore < newScore ? i : bestPos;
      bestScore = bestScore < newScore ? newScore : bestScore;
    }
  bestMutationPos[blockIdx.x] =
                                  (bestScore > (
                                                (readLength - l + 1)
                                              + (readLength - l + 1) >> 1))
                                     ? bestPos : readLength;
  bestMutationIdx[blockIdx.x] = bestIdx;
}
```

Appendix C: GPU Euler Command Line Reference

Synopsis

eulercuda ¡Required Arguments¿ [Error Correction Params] [Assembly Params] [Optional Args]

Description

blabla bla bla bal

Options

General Arguments

-h | --help

display this help

Required Arguments

<-i | --input-file> <file>

Input file in FASTA format

<-o | --output-file > <file>

Output file (generated in FASTA format)

<-r --read-length> <n>

Read Length for input set of reads

Error Correction Parameters

<-e | --error-correction>

Enable error correction

<-t | --tuple> <n>

tuple size for error correction

<-m | --max-iterations> <n>(=1)

Error correction iterations

Assembly Parameters

<-a | --assemble>

Enable Assembly

<-l | --lmer> <n>(=16)

tuple size for debruijn graph

<-b | --block-size> <n>(=512)

block size for CUDA execution

<-c | --coverage> <n>(=20)

Read Coverage

Optional Arguments

<-v | --verbose> <n>

verbose level (0 = off, 9 = full)

Bibliography

Bibliography

[1] W. R. Jeck, J. A. Reinhardt, D. A. Baltrus, M. T. Hickenbotham, V. Magrini, E. R. Mardis, J. L. Dangl, and C. D. Jones, "Extending assembly of short DNA sequences to handle error," *Bioinformatics*, vol. 23, no. 21, p. 2942, 2007.

[2] P. A. Pevzner, H. Tang, and M. S. Waterman, "An eulerian path approach to DNA fragment assembly," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 98, no. 17, p. 9748, 2001.

[3] E. W. Myers, "The fragment assembly string graph," *Bioinformatics*, vol. 21, no. suppl_2, pp. ii79–ii85, 2005.

[4] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol, "ABySS: a parallel assembler for short read sequence data," *Genome Research*, vol. 19, no. 6, pp. 1117–1123, 2009.

[5] B. Jackson, P. Schnable, and S. Aluru, "Parallel short sequence assembly of transcriptomes," *BMC bioinformatics*, vol. 10, no. Suppl 1, p. S14, 2009.

[6] B. G. Jackson and S. Aluru, "Parallel construction of bidirected string graphs for genome assembly," in *2008 37th International Conference on Parallel Processing*, Portland, Oregon, USA, 2008, pp. 346–353.

[7] P. A. Pevzner, H. Tang, and M. S. Waterman, "A new approach to fragment assembly in DNA sequencing," in *Proceedings of the fifth annual international conference on Computational biology*, 2001, pp. 256 –267.

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

[9] I. Birol, S. D. Jackman, C. B. Nielsen, J. Q. Qian, R. Varhol, G. Stazyk, R. D. Morin, Y. Zhao, M. Hirst, J. E. Schein *et al.*, "De novo transcriptome assembly with ABySS," *Bioinformatics*, vol. 25, no. 21, p. 2872, 2009.

[10] S. F. Mahmood and H. Rangwala, "Gpu-euler: Sequence assembly using gpgpu," in *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications*, ser. HPCC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 153–160.

[11] M. Zvelebil, *Understanding bioinformatics*. Garland Science, 2007.

[12] J. C. Venter, H. O. Smith, and L. Hood, "A new strategy for genome sequencing," *Nature*, vol. 381, no. 6581, pp. 364–366, May 1996.

[13] F. Sanger, S. Nicklen, and A. Coulson, "Dna sequencing with chain-terminating inhibitors," *Proceedings of the National Academy of Sciences*, vol. 74, no. 12, pp. 5463–5467, 1977.

[14] M. Pop, "Genome assembly reborn: recent computational challenges," *Briefings in Bioinformatics*, vol. 10, no. 4, pp. 354–366, 2009.

[15] P. Medvedev, K. Georgiou, G. Myers, and M. Brudno, "Computability of models for sequence assembly," *Algorithms in Bioinformatics*, pp. 289–301, 2007.

[16] T. Gingeras, J. Milazzo, D. Sciaky, and R. Roberts, "Computer programs for the assembly of dna sequences," *Nucleic Acids Research*, vol. 7, no. 2, pp. 529–543, 1979.

[17] D. Hernandez, P. Francois, L. Farinelli, M. Osteras, and J. Schrenzel, "De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer," *Genome Research*, vol. 18, no. 5, pp. 802–809, 2008.

[18] R. L. Warren, G. G. Sutton, S. J. Jones, and R. A. Holt, "Assembling millions of short DNA sequences using SSAKE," *Bioinformatics*, vol. 23, no. 4, p. 500, 2007.

[19] J. C. Dohm, C. Lottaz, T. Borodina, and H. Himmelbauer, "SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing," *Genome Research*, vol. 17, no. 11, pp. 1697–1706, 2007.

[20] D. W. Bryant, W. K. Wong, and T. C. Mockler, "QSRA – a quality-value guided de novo short read assembler," *BMC bioinformatics*, vol. 10, no. 1, p. 69, 2009.

[21] E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, M. J. Flanigan, S. A. Kravitz, C. M. Mobarry, K. H. Reinert, K. A. Remington *et al.*, "A whole-genome assembly of drosophila," *Science*, vol. 287, no. 5461, p. 2196, 2000.

[22] S. Batzoglou, "ARACHNE: a Whole-Genome shotgun assembler," *Genome Research*, vol. 12, no. 1, pp. 177–189, 2002.

[23] R. M. Idury and M. S. Waterman, "A new algorithm for dna sequence assembly," *Journal of Computational Biology*, vol. 2, pp. 291–306, 1995.

[24] D. R. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de bruijn graphs," *Genome Research*, vol. 18, no. 5, pp. 821–829, 2008.

[25] M. Hossain, N. Azimi, and S. Skiena, "Crystallizing short-read assemblies around seeds," *BMC Bioinformatics*, vol. 10, no. Suppl 1, p. S16, 2009.

[26] J. Butler, I. MacCallum, M. Kleber, I. A. Shlyakhter, M. K. Belmonte, E. S. Lander, C. Nusbaum, and D. B. Jaffe, "ALLPATHS: de novo assembly of whole-genome shotgun microreads," *Genome Research*, vol. 18, no. 5, p. 810, 2008.

[27] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. 21, no. 9, pp. 948–960, Sep. 1972.

[28] J. Greiner, "A comparison of parallel algorithms for connected components," in *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures - SPAA '94*, Cape May, New Jersey, United States, 1994, pp. 16–25.

[29] G. Cong and D. A. Bader, "Techniques for designing efficient parallel graph algorithms for SMPs and multicore processors," *Lecture Notes in Computer Science*, vol. 4742, p. 137, 2007.

[30] F. Y. Chin, J. Lam, I. Chen *et al.*, "Efficient parallel algorithms for some graph problems," *Communications of the ACM*, vol. 25, no. 9, p. 665, 1982.

[31] T. Hsu, V. Ramachandran, and N. Dean, *Implementation of parallel graph algorithms on a massively parallel SIMD computer with virtual processing.* University of Texas at Austin, Dept. of Computer Sciences, 1993.

[32] S. Makki, "A distributed algorithm for constructing an Eulerian tour," in *Performance, Computing, and Communications Conference, 1997. IPCCC 1997., IEEE International.* IEEE, 1997, pp. 94–100.

[33] B. Awerbuch, A. Israeli, and Y. Shiloach, "Finding euler circuits in logarithmic parallel time," in *Proceedings of the sixteenth annual ACM symposium on Theory of computing - STOC '84*, Not Known, 1984, pp. 249–257.

[34] Awerbuch and Shiloach, "New connectivity and MSF algorithms for Shuffle-Exchange network and PRAM," *IEEE Transactions on Computers*, vol. C-36, no. 10, pp. 1258–1263, 1987.

[35] J. Siek, L. Lee, A. Lumsdaine, L. Lee, L. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, M. Heroux *et al.*, "The boost graph library: User guide and reference manual," in *Proceedings of the*, vol. 243, 2002, pp. 112–121.

[36] D. Johnson, "Connected components inO(log3/2n) parallel time for the CREW PRAM," *Journal of Computer and System Sciences*, vol. 54, no. 2, pp. 227–242, 1997.

[37] Y. Shiloach and U. Vishkin, "An (log) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57–67, 1982.

[38] D. Bader and G. Cong, "A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs)(extended abstract)," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, vol. 1, Santa Fe, NM, USA, 2004, pp. 38–47.

[39] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the GPU," *ACM Transactions on Graphics*, vol. 28, no. 5, p. 1, 2009.

[40] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with cuda," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, Aug. 2007.

[41] S. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithms for GPUs," Citeseer, Tech. Rep., 2008.

[42] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," in *Graphics Hardware 2007.* ACM, Aug. 2007, pp. 97–106.

[43] Nvidia, "Cuda toolkit reference manual."

[44] D. C. Richter, F. Ott, A. F. Auch, R. Schmid, and D. H. Huson, "MetaSim- a sequencing simulator for genomics and metagenomics," *PLoS One*, vol. 3, no. 10, p. 3373, 2008.

[45] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg, "Versatile and open software for comparing large genomes," *Genome biology*, vol. 5, no. 2, p. R12, 2004.

[46] W. Huang, L. Li, J. R. Myers, and G. T. Marth, "Art: a next-generation sequencing read simulator," *Bioinformatics*, vol. 28, no. 4, pp. 593–594, 2012.

[47] D. Blankenberg, G. Von Kuster, N. Coraor, G. Ananda, R. Lazarus, M. Mangan, A. Nekrutenko, and J. Taylor, "Galaxy: a web-based genome analysis tool for experimentalists." *Current protocols in molecular biology / edited by Frederick M. Ausubel ... [et al.]*, vol. Chapter 19, Jan. 2010.

[48] J. Goecks, A. Nekrutenko, J. Taylor, and Galaxy Team, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences." *Genome biology*, vol. 11, no. 8, pp. R86+, Aug. 2010.

[49] B. Giardine, C. Riemer, R. C. Hardison, R. Burhans, L. Elnitski, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, J. Taylor, W. Miller, W. J. Kent, and A. Nekrutenko, "Galaxy: a platform for interactive large-scale genome analysis." *Genome research*, vol. 15, no. 10, pp. 1451–1455, Oct. 2005.

Curriculum Vitae

Syed Faraz Mahmood received his Bachelor of Science in Computer Science in 2004 from FAST - National University of Computer and Emerging Sciences, Karachi,Pakistan. He started his carrer as a Software Engineer at Kalsoft (Pvt.) Limited, Karachi. he worked at Sharesoft Solutions, Dubai, UAE for three years before joining GMU as a graduate student. He worked for 18 months as Graduate Reasearch Assistant with Dr. Huzefa Rangwala at Data Mining Lab, GMU. Nowadays, he works at Nodal Exchange, Vienna, VA as an Application Developer.