Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

GPU-Euler: Sequence Assembly using GPGPU

Syed F. Mahmood
smahmoo4@gmu.edu

Huzefa Rangwala
rangwala@cs.gmu.edu

Technical Report GMU-CS-TR-2011-1

Abstract

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 showed 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.

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). 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 [20], graph oriented approach that find Eulerian tours [27], and use bi-directed string graph representation [24]. ABySS [35] 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. [17, 19] 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 [28]. Our method was motivated by the current advances in multi-core technologies and the use of graphic processor units (GPUs) in several computing applications. In this paper, 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 was 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 250 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 [7]. Our experimental evaluation showed the promise of using GPUs for genome assembly tasks. In terms of run-time performance, GPU-Euler outperformed EulerSR. In our current implementation, we do not perform any graph simplification as well as error correction that are integral for producing accurate contigs.

The rest of the paper is organized as follows. Section 2 provides a thorough review of different sequence assemblers. Sections 3 and 4 provide the background on the parallel computing models and the implementation details of GPU-Euler. Section 5 provides the experimental results. Section 6 describes several directions for future work and provides concluding remarks.

2 Literature Review

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 [36], where the genome is randomly sheared into many small subsequences or pieces referred to as reads. The problem of combining these sequence reads to reconstruct the source genome is called the "sequence assembly" problem. Sanger, 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 [29] and Myers et. al. [23] provide a detailed review of the computational challenges involved with sequence assembly, along with a study of the widely used approaches. *De novo* assembly algorithms stitch together short fragment reads and put them in order to get long contiguous DNA fragments called contigs, which are further extended to get supercontigs 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, (ii) overlap-layout-consensus (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 [37], SHARCGS [10], QSRA [5] and VCAKE [20] 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 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 [25], Arachne [4] and EDENA [13].

The Eulerian based de novo methods have always been widely used, and were inspired by the sequencing-by-hybridization approach [16, 28]. These algorithms represent each read by its set of k-mers (smaller subsequences) and construct a de-Bruijn graph. A *de-Bruijn* graph is a directed graph where vertices are k-mers, and there exists an edge between two vertices if there is an overlapping subsequence 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 from the de-Bruijn graph, nodes and edges 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 [24]. 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 [7], Velvet [38], SHORTY [14], ALLPATHS [6] and ABYSS [35] 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.

ABySS [35] 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. [17, 19] 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 propose an implementation of the Eulerian-based sequence assembler, that will utilize graphic processor units (GPUs) to produce sequence assemblies from short read NGS data.

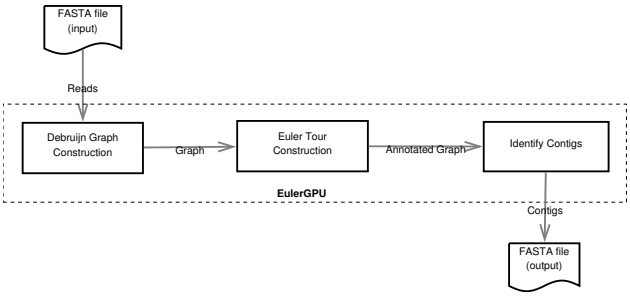

Figure 1: GPU-Euler Work Flow.

3 Background

3.1 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. For the past few years, there have been several efforts, to tap the potential of GPU's parallel computation capability so that they 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 runtime 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 for 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). CUDA threads are scheduled to run on these SPs. The actual number of threads scheduled to run concurrently depends on the resources available to SMs. 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 accessibility to threads or to blocks.

3.2 Parallel Graph Algorithms

Designing efficient graph algorithms for parallel 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 [8, 9, 11, 15].

3.2.1 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 architecture. Makki [22] 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. [3] 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, we opted for Awerbuch's approach 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. We implement a GPU-based parallel Eulerian tour algorithm by following the above approach. We specifically use a parallel connected component algorithm as a step during 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 [2]. In our implementation, we opted for Kruskal's Spanning tree algorithm as implemented in BOOST graph library [34].

Finding Connected Components For a given graph, finding connected components involves partitioning the

graph such that the vertices in the partitioned subgraph (subset) are connected to each other through some path and there does not exist any path between vertices of different subgraphs.

Shiloach-Vishkin [33] 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 updated. Update steps corresponds 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.

4 Methods

The current implementation of our algorithm works in three steps as outlined in Figure 1. In the first step, *k*-mers are extracted from the input sequence reads (in a fasta file) and the de-Bruijn graph is constructed. In the second step, the de-Bruijn graph is processed on the GPU to find the euler tour across the graph. In the final step, the contigs are identified and sent to the output. At this moment no error correction is performed during the assembly.

4.1 de-Bruijn Graph Data Structure

An important aspect of our implementation is the data structure that was used to represent the de-Bruijn graph. We had two considerations while implementing this data structure. Firstly, the data structure should be able to express better locality, which is a driving force for developing GPU specific application. This would yield to a layout where those attributes that are going to be accessed by the same CUDA kernel will be contiguous, resulting in an efficient memory access pattern. Secondly, the representation should be efficient in terms of the memory used, due to the the capacity of memory available on GPUs. This would also reduce the time spent transferring data from the host memory to the GPU memory.

Our de-Bruijn graph is represented as a list vertices V, a list of edges E, and two lists of pointers, EP and LP, which store information about the leaving and entering edges for each of the vertices, respectively. The entries within EP and LP point to locations within the edge list E. Also, since the number of entering (incoming) and leaving (outgoing) edges per vertex is unknown, both EP and LP are maintained as dynamic lists, with allocation information per vertex computed during run time.

4.2 de-Bruijn Graph Construction

The input sequence reads are represented as a string over four nucleotides or alphabets (A, C, G, T). As such, each base can be represented using two bits and similarly a *k*-mer can be represented as a sequence of $2 * k$ bits. (A *k*-mer is a subsequence of a longer DNA sequence of length *k*). With the 2-bit encoding per character, each *k*-mer is mapped onto a unique integer value. This encoding allows easy computation of *k*-mer neighbors by simple bitwise operations (Shift, OR). A neighbor for a *k*-mer x, is one that shares either the prefix or suffix of length $k - 1$ with x.

For the de-Bruijn graph construction, we used a scheme where a hash table [1] is constructed with a (key,value) pair defined using the *k*-mer's encoded representation as key, and its index within an array as value. The steps for the GPU-based de-Bruijn graph construction are shown in Figure 2) Initially, the input file containing the reads is processed to extract $(k + 1)$-mers and *k*-mers. Each *k*-mer is then assigned an index value based on the hashing function, which can serve as the location to store various attributes for the given *k*-mer in associative array representation.

Each *k*-mer represents a vertex in the de-Bruijn graph, while each *l*-mer, where $(l = k + 1)$, defines an edge between two *k*-mers representing an overlap of $k - 1$ nucleotides (or characters). Algorithm 1 uses three CUDA kernel launches. After the initial processing, a CUDA kernel is invoked to process each of the *l*-mers (i.e., count the edges), and update the in-degree and out-degrees for the different *k*-mer vertices. As seen in Algorithm 1, we maintain two count lists *ECount* and *LCount* to calculate the number of entering and leaving edges per vertex. Using the CUDPP Library [12, 31, 32], we perform a parallel prefix scan on the GPUs to compute the memory required per vertex to store the information regarding the entering and leaving edges. As such, we record the offset locations for both types of edges, per vertex. This allows us to maintain the dynamic *LP* and *EP* data structures.

After allocating the required memory, a second CUDA kernel is invoked to set the vertices with the edge list offset and count information. During the final step, a

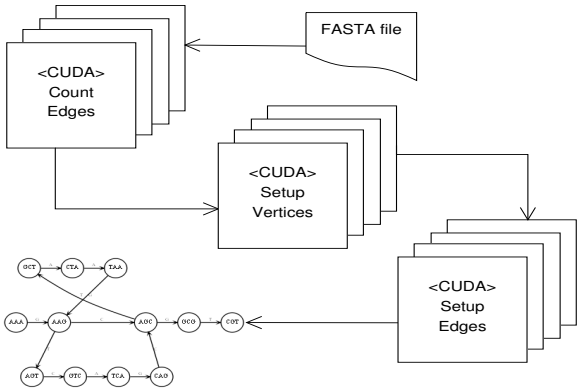

Figure 2: de-Bruijn Graph Construction.

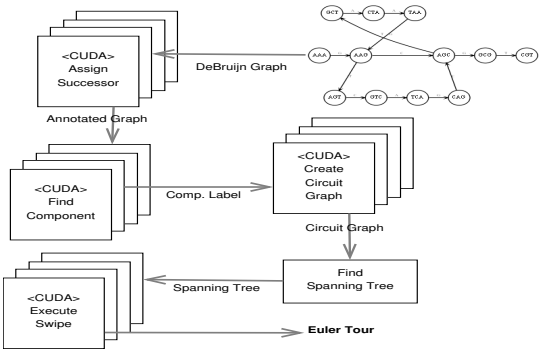

Figure 3: Parallel Euler Tour.

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.

4.3 Euler Tour Construction

The Euler tour construction step is a modified implementation of the algorithm proposed by Vishkin et. al. [3] for the CRCW PRAM model. These modifications were necessary, since CUDA follows the SPMD (Single Program Multiple Data) paradigm without inherent instruction level synchronization. On the other hand, the CRCW PRAM model expects that all processors synchronize after executing each instruction. CRCW PRAM also allows for concurrent read/write to the same memory location. CUDA does not support concurrent writes. CUDA offers two kinds of synchronizations [26]; (i) Across a thread block, all threads in a single block can be synchronized using an API command. However, threads across the blocks aren't affected by this method. (ii) Across all threads, there is automatic synchronization at the launch and completion of CUDA kernels.

These two synchronization mechanism can be used while working on CUDA platform, to accomplish a closer simulation of PRAM model. To handle concurrent writes, we can use atomic APIs which serialize all the concurrent accesses to a memory location. Our implemented algorithm defines concurrent writes as one where arbitrary threads (processors) can win and CUDA atomic APIs can be used to mimic this behavior. 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 sets are available on CUDA devices with compute capability 1.1 and later. An alternate option is to use shared memory and break ties using duels.

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 shows major components of the Euler tour construction and are discussed below.

4.3.1 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.

4.3.2 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.

4.3.3 Connected Components

Our implementation is based on the algorithm proposed by Uzi Viskin et. al. [33] 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

Table 1: Genome size and number of simulated reads for different read lengths.

Genome	Length	36 bp	50 bp	250 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

Table 2: Profiling of GPU-Euler.

Phase	Computation
I/O and k-mer Extraction	CPU
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

involving updating of root vertices performed iteratively. (See Section 3.2.1).

4.3.4 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.

4.3.5 Spanning Tree

This step is implemented serially on the CPU using the boost graph library [34], 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.

4.3.6 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, resulting in a connected Eulerian tour. Contigs are generated by identifying the source vertices and following the successor edge information pertaining to each edge.

4.4 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 [3] 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.

5 Experimental Results

5.1 Datasets

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 [28]. 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 [30]. 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 1. Since, we were focused on illustrating the performance of GPU-Euler in terms of run times we simulated only error free reads.

5.2 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 [7]. We ran

Table 3: Run Time Performance for CJ Genome using GPU-Euler (250 bp reads).

k	I/O	**Hash Table**	**de-Bruijn**	Euler Tour				Output	GPU	CPU	Total
				Component	Spanning Tree	**Swipe**	**Other**				
16	61.646	0.129	0.487	4.364	0.123	0.004	0.240	1.501	5.224	65.497	70.722
18	57.705	0.137	0.513	4.371	0.050	0.004	0.156	1.521	5.181	61.476	66.657
22	60.541	0.136	0.513	4.412	0.035	0.004	0.272	1.594	5.339	64.45	69.790
24	62.273	0.136	0.504	4.412	0.033	0.004	0.301	1.643	5.358	66.221	71.579
26	63.682	0.137	0.503	4.423	0.030	0.004	0.321	1.687	5.388	67.740	73.128
28	65.494	0.137	0.514	4.431	0.026	0.004	0.327	1.734	5.413	69.588	75.002
30	67.595	0.136	0.513	4.484	0.025	0.004	0.324	1.768	5.462	71.651	77.114
32	68.965	0.138	0.504	4.387	0.027	0.004	0.312	1.820	5.344	73.072	78.417

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

Table 4: Run Time Performance for NM Genome using GPU-Euler (250 bp reads).

k	I/O	**Hash Table**	**de-Bruijn**	Euler Tour				Output	GPU	CPU	Total
				Component	Spanning Tree	**Swipe**	**Other**				
16	75.948	0.173	0.723	5.798	0.264	0.030	0.593	2.003	7.319	80.858	88.178
18	76.615	0.179	0.759	5.839	0.209	0.028	0.518	2.057	7.326	81.658	88.985
20	79.029	0.181	0.758	5.732	0.181	0.026	0.499	2.108	7.199	84.139	91.339
22	82.235	0.182	0.748	5.753	0.176	0.024	0.473	2.161	7.183	87.428	94.612
24	84.787	0.181	0.763	5.739	0.164	0.023	0.456	2.263	7.163	90.046	97.210
26	87.398	0.183	0.773	5.734	0.158	0.021	0.439	2.281	7.152	92.767	99.920
28	89.956	0.182	0.781	5.755	0.138	0.021	0.430	2.310	7.171	95.323	102.495
30	92.503	0.184	0.793	5.779	0.135	0.019	0.419	2.380	7.195	97.958	105.154
32	95.303	0.184	0.778	5.735	0.124	0.019	0.361	2.479	7.079	100.835	107.915

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

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 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 2 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 [21] 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.

5.2.1 System 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 CUDA compute capiblity 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.

5.3 Run-Time Performance

Tables 3, 4 and 5 show 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, NM and LL genomes, respectively. We performed experiments across reads of length, 36, 50 and 250 bps but show results in these tables for 250 bps. We report the run-times for the different phases of the algorithm across the GPU and CPU as shown in Table 2. 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. We use an unordered set data structure from the Boost library [34] to maintain the list of occurring k-mers and their frequency. This is a dynamic structure and needs to expand, if the number of entries in the set (i.e., k-mers) increase. In the I/O phase we also compress and encode the k-mers using the $base_4$ representation. Thus, the execution time during the I/O phase performed on the CPU dominates the overall performance of our current algorithm. In the future, we intend to extend our GPU-implementation to off-load more work from the CPU.

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

Table 5: Run Time Performance for LL Genome using GPU-Euler (250 bp reads).

k	I/O	**Hash Table**	**de-Bruijn**	Euler Tour				Output	GPU	CPU	Total
				Component	Spanning Tree	**Swipe**	**Other**				
16	84.858	0.200	0.900	6.241	0.186	0.008	0.352	2.224	7.702	90.225	97.928
18	84.397	0.208	0.985	6.278	0.124	0.007	0.302	2.225	7.782	89.877	97.660
20	85.231	0.209	1.001	6.278	0.111	0.006	0.348	2.262	7.845	90.765	98.611
22	88.208	0.208	1.000	6.447	0.096	0.006	0.358	2.384	8.022	93.853	101.876
24	90.868	0.209	0.995	6.315	0.090	0.006	0.366	2.407	7.892	96.547	104.440
26	93.315	0.208	1.016	6.307	0.089	0.006	0.361	2.482	7.900	99.144	107.045
28	95.276	0.210	1.003	6.316	0.083	0.006	0.369	2.530	7.905	101.153	109.059
30	98.125	0.209	0.980	6.341	0.081	0.006	0.352	2.610	7.891	104.079	111.971
32	100.088	0.210	1.008	6.297	0.079	0.006	0.356	2.662	7.879	106.123	114.003

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

Table 6: Comparative Performance for GPU-Euler on the CJ genome (Contigs >= 100 bp).

Assembler	*k*	Time (s)	N50	N	Mean	Max	TB	WA
	Read Length = 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	42.456	8480	720	4491.206	40255	3233668	83.23
	Read Length = 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	49.166	47766	307	10527.147	158836	3231834	91.01
	Read Length = 250 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	78.417	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.

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 1). 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 5.339, 7.199 and 7.869 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 4 we show a plot of the GPU run times across these three genome benchmarks for the different read lengths.

5.4 CUDA Configuration

Within the CUDA specification, we can specify different groupings of threads for the problem instances. CUDA threads are grouped into blocks and blocks are grouped as grids. This arrangement provides different portion of on-chip memory to be shared among a block of threads, which can lead to overall improvement in the execution time depending upon the problem. We also performed experiments using different kernel launch configuration i.e., assigning 4 threads per block to 512 threads per block for our datasets. We observed that the overall GPU run time exhibited very slight variation across the different configurations. This was primarily because of the nature of the problem ported on the GPUs. In this work, we report results with the kernels having 512 threads per block.

5.5 Comparative Performance

We compared our assembler with EulerSR [7], a widely used Eulerian-based assembler developed for short reads. Experiments for EulerSR were performed on a single CPU core of our workstation. Tables 6, 7 and 8 show the contig length statistic, contig accuracy results and the run-time for 36, 50 and 250 bps across the CJ, NM and LL genome benchmarks, respectively. We report these performance results for GPU-Euler 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*". We highlight in bold those entries that show the best performance across the two assemblers.

Across the three genome benchmarks, we notice that

Table 7: Comparative Performance for GPU-Euler on the NM genome (Contigs >= 100 bp).

Assembler	k	Time (s)	N50	N	Mean	Max	TB	WA
	Read Length = 36 bp							
EulerSR*	20	145.105	2063	23	482.826	3830	11105	96.77
	22	144.716	1083	19	510.000	2852	9690	96.84
EulerSR	21	279.277	4538	894	2307.614	21195	2063007	98.26
	22	257.953	3742	1011	2028.121	16374	2050430	98.45
GPU-Euler	22	55.933	3909	2275	1804.153	17635	4104447	84.38
	Read Length = 50 bp							
EulerSR*	27	128.601	1020	32	399.594	5949	12787	91.30
	23	136.295	688	54	321.759	6283	17375	97.60
EulerSR	25	222.046	6808	601	3370.797	25857	2025849	98.19
	23	225.512	6574	594	3395.434	23693	2016888	99.32
GPU-Euler	23	67.948	6596	1964	2121.377	25383	4166385	81.37
	Read Length = 250 bp							
EulerSR*	16	198.608	481	2245	367.697	5227	825479	73.69
	31	-	-	-	-	-	-	-
EulerSR	27	162.595	31614	346	9365.142	79346	3240339	47.48
	31	-	-	-	-	-	-	-
GPU-Euler	31	106.305	7226	1610	2715.053	30965	4371235	79.97

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.

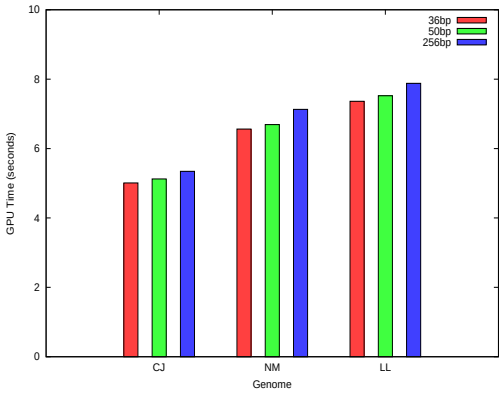

Figure 4: GPU Time comparison across different genomes.

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 250 base pairs, we notice that the run time for GPU-Euler 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.

6 Conclusion and Future Work

In this work, 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 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.

The hash table implementation can be improved for more efficient creation and access time. We can also investigate the possible use of shared memory and texture memory especially for hash lookup. At this moment , k-mer extraction and read encoding is done on CPU, which can be moved to GPU for additional improvements. Besides the run-time performance, we intend to improve the contig lengths and accuracy by incorporating graph simplification strategies as well as error correction capabilities. From a benchmarking perspective, we would like to compare GPU-Euler to EulerSR using multi-cores, as well as other distributed-memory parallel assemblers developed, ABySS [35] and YGA [18].

A limitation of our approach is scalability in terms of the genome sizes. One of the limitations of using GPUs, is the amount of available memory and the high memory requirement for genome assembly and analysis tasks. This can potentially be avoided by a pipelining solution that would break the computation into different parts and inter-leave the operations that would be performed

Table 8: Comparative Performance for GPU-Euler on the LL genome (Contigs >= 100 bp).

Assembler	k	Time (s)	N50	N	Mean	Max	TB	WA
	Read Length = 36 bp							
EulerSR*	19	158.887	5358	7	1380.143	5358	9661	99.73
	21	152.476	5222	3	3045.000	5222	9135	99.6
EulerSR	20	276.670	9180	619	3862.969	31070	2391178	99.62
	21	270.720	9154	631	3753.751	33986	2368617	97.76
GPU-Euler	21	62.359	7234	1500	3061.075	37629	4591612	80.48
	Read Length = 50 bp							
EulerSR*	24	139.019	5381	18	590.000	5381	10620	99.08
	26	133.875	5379	10	988.700	5379	9887	99.17
EulerSR	25	216.472	30133	317	7264.991	83247	2303002	97.36
	26	216.281	27804	318	7168.132	83249	2279466	99.56
GPU-Euler	26	71.991	23998	877	5259.621	95876	4612688	84.76
	Read Length = 250 bp							
EulerSR*	17	189.402	479	2319	377.916	3719	876388	76.6
	32	-	-	-	-	-	-	-
EulerSR	27	167.085	74484	193	11967.648	198384	2309756	97.29
	32	-	-	-	-	-	-	-
GPU-Euler	32	113.254	13789	1008	4699.620	47788	4737217	94.58

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.

on the GPU and CPU. Such a strategy will slow the overall process but would allow assembly for large human genomes.

References

[1] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the GPU. *ACM Transactions on Graphics*, 28(5):1, 2009.

[2] Awerbuch and Shiloach. New connectivity and MSF algorithms for Shuffle-Exchange network and PRAM. *IEEE Transactions on Computers*, C-36(10):1258–1263, 1987.

[3] 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*, pages 249–257, Not Known, 1984.

[4] S. Batzoglou. ARACHNE: a Whole-Genome shotgun assembler. *Genome Research*, 12(1):177–189, 2002.

[5] D. W Bryant, W. K Wong, and T. C Mockler. QSRA – a quality-value guided de novo short read assembler. *BMC bioinformatics*, 10(1):69, 2009.

[6] 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*, 18(5):810, 2008.

[7] M. J. Chaisson and P. A. Pevzner. Short read fragment assembly of bacterial genomes. *Genome Research*, 18(2):324–330, 2008.

[8] F. Y Chin, J. Lam, I. Chen, et al. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):665, 1982.

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

[10] 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*, 17(11):1697–1706, 2007.

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

[12] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with cuda. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007.

[13] 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*, 18(5):802–809, 2008.

[14] Mohammad Hossain, Navid Azimi, and Steven Skiena. Crystallizing short-read assemblies around seeds. *BMC Bioinformatics*, 10(Suppl 1):S16, 2009.

[15] 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.

[16] Ramana M. Idury and Michael S. Waterman. A new algorithm for dna sequence assembly. *Journal of Computational Biology*, 2:291–306, 1995.

[17] B. Jackson, P. Schnable, and S. Aluru. Parallel short sequence assembly of transcriptomes. *BMC bioinformatics*, 10(Suppl 1):S14, 2009.

[18] B. G. Jackson, M. Regennitter, X. Yang, P. S. Schnable, and S. Aluru. Parallel de novo assembly of large genomes from high-throughput short reads. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, 2010.

[19] Benjamin G. Jackson and Srinivas Aluru. Parallel construction of bidirected string graphs for genome assembly. In *2008 37th International Conference on Parallel Processing*, pages 346–353, Portland, Oregon, USA, 2008.

[20] 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*, 23(21):2942, 2007.

[21] 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*, 5(2):R12, 2004.

[22] SAM Makki. A distributed algorithm for constructing an Eulerian tour. In *Performance, Computing, and Communications Conference, 1997. IPCCC 1997., IEEE International*, pages 94–100. IEEE, 1997.

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

[24] E. W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl_2):ii79–ii85, 2005.

[25] 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.J Reinert, K. A Remington, et al. A whole-genome assembly of drosophila. *Science*, 287(5461):2196, 2000.

[26] Nvidia. Cuda toolkit reference manual.

[27] 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*, 98(17):9748, 2001.

[28] 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*, pages 256 –267, 2001.

[29] M. Pop. Genome assembly reborn: recent computational challenges. *Briefings in Bioinformatics*, 10(4):354–366, 2009.

[30] D. C Richter, F. Ott, A. F Auch, R. Schmid, and D. H Huson. MetaSim- a sequencing simulator for genomics and metagenomics. *PLoS One*, 3(10):3373, 2008.

[31] S. Sengupta, M. Harris, and M. Garland. Efficient parallel scan algorithms for GPUs. Technical report, Citeseer, 2008.

[32] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Graphics Hardware 2007*, pages 97–106. ACM, August 2007.

[33] Y Shiloach and U Vishkin. An (log) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[34] J. Siek, L.Q. Lee, A. Lumsdaine, L.Q. Lee, L.S. 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*, volume 243, pages 112–121, 2002.

[35] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J.M. Jones, and I. Birol. ABySS: a parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009.

[36] J. Craig Venter, Hamilton O. Smith, and Leroy Hood. A new strategy for genome sequencing. *Nature*, 381(6581):364–366, May 1996.

[37] R. L Warren, G. G Sutton, S. J.M Jones, and R. A Holt. Assembling millions of short DNA sequences using SSAKE. *Bioinformatics*, 23(4):500, 2007.

[38] D. R. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18(5):821–829, 2008.

Algorithm 1 de-Bruijn Graph Construction on CUDA-based GPUs.

Input: L: l -mer list, T (): k-mer hash function
Output: V : Vertex List, E : Edge List , LP : Leaving Edge Pointers List, EP : Entering Edge Pointers List

/ Temporary Lists */*
LCount : Leaving Edge Count List
LOffset : Leaving Edge Offset List
ECount : Entering Edge Count List
EOffset : Entering Edge Offset List
/ Kernel 1 for Counting Edges */*

1: **for all** $thread_i : L_i$ in L **do**
2: $p \leftarrow$ PREFIX (L_i)
3: $s \leftarrow$ SUFFIX (L_i)
4: $pindex \leftarrow T(p)$ /* T is a lookup hash function */
5: $sindex \leftarrow T(s)$
6: ECount $[sindex] + +$
7: LCount $[pindex] + +$
8: **Synchronize**

/ CUDPP Library Prefix-Scan */*
9: $EOffset \leftarrow PREFIX - SCAN(ECount)$
10: $LOffset \leftarrow PREFIX - SCAN(LCount)$

/ Kernel 2 for Vertices Setup */*
11: **for all** $thread_i : (key, index) \leftarrow (T_i) \in T$ **do**
12: V [$index$].vid $\leftarrow$ key
13: V [$index$].EOffset $\leftarrow$ EOffset $[index]$
14: V [$index$].LOffset $\leftarrow$ LOffset $[index]$
15: **Synchronize**

/ Kernel 3 for Edge Setup */*
16: **for all** $thread_i : L_i$ in L **do**
17: $p \leftarrow$ PREFIX (L_i)
18: $s \leftarrow$ SUFFIX (L_i)
19: $pindex \leftarrow T(p)$ /* T is a lookup hash function */
20: $sindex \leftarrow T(s)$
21: E [i].source $\leftarrow pindex$
22: E [i].sink $\leftarrow sindex$
23: Update LP [V[$pindex$].LOffset] with i
24: Update EP [V[$sindex$].EOffset] with i