2018 IEEE International Parallel and Distributed Processing Symposium

GPU-Accelerated Large-Scale Genome Assembly

Sayan Goswami, Kisung Lee, Shayan Shams, Seung-Jong Park
Division of Computer Science and Engineering, Center for Computation & Technology
Louisiana State University (LSU), Baton Rouge, LA
sgoswami@cct.lsu.edu, lee@csc.lsu.edu, sshams2@cct.lsu.edu, sjpark@cct.lsu.edu

***Abstract*—Spurred by a widening gap between hardware accelerators and traditional processors, numerous bioinformatics applications have harnessed the computing power of GPUs and reported substantial performance improvements compared to their CPU-based counterparts. However, most of these GPU-based applications only focus on the *read alignment* problem, while the field of *de novo assembly* still relies mostly on CPU-based solutions. This is primarily due to the nature of the assembly workload which is not only compute-intensive but also extremely data-intensive. Such workloads require large memories, making it difficult to adapt them to use GPUs with their limited memory capacities. To the best of our knowledge, no GPU-based assembler reported in the recent literature has attempted to assemble datasets larger than a few tens of gigabytes, whereas real sequence datasets are often several hundreds of gigabytes in size. In this paper, we present a new GPU-accelerated genome assembler called LaSAGNA, which can assemble large-scale sequence datasets using a single GPU by building string graphs from approximate all-pair overlaps. LaSAGNA can also run on multiple GPUs across multiple compute nodes connected by a high-speed network to expedite the assembly process. To utilize the limited memory on GPUs efficiently, LaSAGNA uses a semi-streaming approach that makes at most a logarithmic number of passes over the input data based on the available memory. Moreover, we propose a two-level streaming model, from disk to host memory and from host memory to device memory, to minimize disk I/O. Using LaSAGNA, we can assemble a 400 GB human genome dataset on a single NVIDIA K40 GPU in 17 hours, and in a little over 5 hours on an 8-node cluster of NVIDIA K20s.**

***Keywords*-Genomics, Computational biology, Memory management, Big data, Parallel processing**

I. Introduction

Since the start of the Human Genome Project, genomics has contributed nearly US$1 trillion to the economy [1] and has been widely used in life sciences as well as applied fields such as medicine, forensic science, and virology [2]–[4]. Obtaining an organism's DNA consists of two steps - sequencing and assembly. Sequencing is the process of determining the exact order of nucleotides in the DNA. Since existing technologies cannot read a whole genome in one go, they clone it and extract millions of shorter fragments called *short-reads* (Fig. 1). Assembly is the process of aligning and merging short-reads to reconstruct the original sequence.

Rapid advancements in Next-Generation Sequencing (NGS) technology have drastically reduced the cost of sequencing whole genomes, resulting raw datasets to explode in terms of size and quantity. For instance, a single run of an Illumina HiSeq 4000 sequencer can generate up to 5 billion 150-nucleotide reads at about $0.05 per million bases. Although several tools have been developed to process these high-throughput sequence datasets, assembling a real-world human genome dataset whose size is nearly half a terabyte still requires either a scale-up machine with terabytes of RAM or a scale-out cluster with several dozens of nodes. Even with the pay-as-you-go pricing model of cloud services and availability of various big data frameworks, access to such high-end machines or large-scale clusters is often too expensive for most researchers. Therefore, it has become essential to utilize hardware acceleration to expedite the processing of large genome sequence datasets economically.

Lately, we have observed a growing interest in general-purpose GPUs (Graphics Processing Units) in domains such as bioinformatics, molecular dynamics, computer vision, and most notably in deep learning. We have also noticed a widening gap between the computational capabilities of general-purpose CPUs and GPUs. For example, the recently introduced NVIDIA Tesla V100 theoretically has 15 TFLOP/s of single precision (FP32) performance and delivers 900 GB/sec peak memory bandwidth, which is an order of magnitude higher than the 85 GB/sec found in the latest Intel Xeon Processor (E7-8894 v4).

Several specialized tools have already benefited from the massively parallel processing capabilities of GPUs across a wide range of bioinformatics applications. These include NVIDIA's NVBIO [5], CUSHAW [6], CUSHAW2-GPU [7], BarraCUDA [8], SOAP3 [9], SARUMAN [10], PUNAS [11], and GPU-based tools with optimized BWT (Burrows-Wheeler Transformation) [12], [13] and Smith-Waterman algorithm [14]. However, most of these studies focus on read alignment, whereas sequence assembly is not well-addressed because of the limited GPU device memory.

Although few GPU-based genome assemblers have been proposed, they are designed for small error-free simulated datasets which inherently limits their assembly performance. Specifically, the largest dataset they have assembled is in a few tens of gigabytes, which is much smaller than typical human genome datasets. Given that the high computational capability of GPUs has not been fully utilized by existing genome assembly tools, it is crucial to develop a new GPU-based assembly framework that uses adequate techniques to

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00091

IEEE computer society

Figure 1. Shotgun sequencing

scale up to the size of real-world human genomes without requiring large amounts of expensive computing resources.

Several studies have reported that the most challenging step in genome assembly is building a graph (either a string graph or de Bruijn graph) from the short reads since it requires a lot of computing power and a large amount of memory [15] [16]. To address the fundamental limitation of GPUs in large-scale genome assembly, we present a new GPU-accelerated genome assembler, called LaSAGNA, that can assemble datasets with billions of sequences using a single GPU by building string graphs from approximate all-pair overlaps. LaSAGNA significantly reduces the memory requirement by employing a semi-streaming approach that minimizes the number of disk accesses based on the available memory. It can also run on multiple GPUs across multiple compute nodes to expedite the assembly pipeline. This paper makes several contributions as follows. 1) We develop a new genome assembly framework that can build an approximate overlap graph from a real-world human genome sequence dataset (several hundred GB in size) using a single GPU equipped with only 6 GB device memory. 2) We present a two-level streaming model (from disk to host memory and from host memory to device memory) that effectively utilizes the memory hierarchy to reduce the disk I/O in large-scale genome assembly. 3) We implement a distributed version of LaSAGNA to facilitate faster operation on a cluster of compute nodes. Given that the assembly workload is heavily I/O-intensive, LaSAGNA can benefit from the larger aggregated I/O bandwidth available in distributed computing. 4) We evaluate LaSAGNA using several real-world genome sequence datasets to exhibit its efficiency in various computing environments. Our experimental results show that LaSAGNA can assemble a 400 GB human genome dataset in 17 hours using a single GPU (NVIDIA K40).

The rest of the paper is organized as follows. In Section II, we introduce the preliminaries of the genome assembly problem followed by a discussion of related literature. Next, we describe our proposed approach and its distributed implementation in Section III. Lastly, we show our experimental results in Section IV and conclude this paper in Section V.

II. Background and Related Work

A. Preliminaries

1) De novo assembly: Genome assembly can be performed either by *mapping* or *de novo*. In mapping assembly, reads are aligned with some reference genome of the same or a closely related species. On the other hand, in de novo assembly, the original sequence is obtained by just finding the overlaps between reads. De novo assembly is performed using either a de Bruijn graph-based approach [17] or a string graph-based approach [18]. A de Bruijn graph is created from all unique k-length substrings of sequences as vertices, and their $(k-1)$-length overlaps as edges. This method is prone to collapsing repeated regions of the genome that are larger than k, causing information loss [19].

2) De novo assembly with string graphs: Given a set R of reads and their Watson Crick (WC) complements, a string graph G in its simplest form consists of R as vertices and a set of directed weighted edges E. An edge $e = (r_i, r_j, l)$, where $r_i, r_j \in R$ and $l \in [1, min(|r_i|, |r_j|))$, exists in E if the l-length suffix of r_i matches the l-length prefix of r_j. Naturally, any edge $e = (r_i, r_j, l) \in E$ must also have a complementary edge $e\prime = (r_j\prime, r_i\prime, l)$, where $r_i\prime$ is the WC-complement of r_i.

In theory, there exists a tour in G that corresponds to the original sequence from which R was generated. However, the graph contains a lot of redundant information in the form of transitive edges. Specifically, if a read r_j overlaps with r_k and read r_i overlaps with both r_j and r_k, then the edge (r_i, r_k) can be removed from G without any information loss. Furthermore, a read that is completely contained in another one may also be removed. A heuristic that is often used to prune the graph is a greedy approach, where only one outgoing edge corresponding to the read with the longest overlap is considered for assembly, and the others are ignored [20], [21].

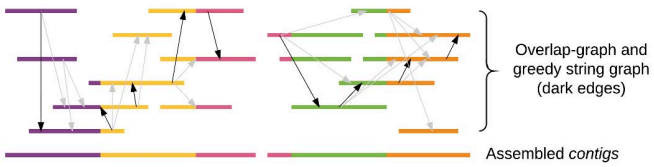

Figure 2. String graph-based assembly

Once the graph is simplified, paths that can be unambiguously traversed are spelled out to obtain subsequences of the original DNA string. These subsequences are known as *contigs*. Fig. 2 shows a string graph created from short reads and the contigs generated by traversing it.

B. Related work

GPU-Euler [22] is the first genome assembler built for GPUs, followed by GAGM [23], which proposes paired de Bruijn graph construction and contig generation using a single GPU. Both assemblers were evaluated using small simulated error-free reads generated from bacterial genomes. Another GPU-based de Bruijn graph construction tool [15] proposes a staged graph construction pipeline and reports its results on a 13 GB human chromosome dataset. GAMS [19] is the first string graph-based assembler that uses GPUs and reports the assembly of a human chromosome using a 16-node GPU cluster. Some works, such as FAssem [24], have

studied the application of FPGAs in assembly, but are also evaluated on small bacterial datasets.

There are many CPU-based assembly tools, and most of them are based on de Bruijn graphs. Since the advent of first-generation assemblers such as Velvet [25] and SOAPdenovo [26], several assemblers, such as Minia [27] and MEGAHIT [28], have been proposed to use succinct data structures (e.g., bloom filters) to store large graphs in memory. Many distributed assemblers have also emerged to expedite the assembly of high-throughput sequencing datasets. Notable among them are Ray [29] and SWAP [30] built on MPI, Lazer [31] which uses ZeroMQ, Contrail [32] and Giga [33] based on Hadoop, and HipMer [34] based on the global-address space model. Among the string graph-based assemblers, SGA [35] is the only one that can process large datasets on a single node using compressed data structures. To the best of our knowledge, CloudBrush [36] is the only distributed string graph-based assembler, but has only been evaluated on small datasets and is still under development.

III. Methodology for GPU-Accelerated Assembly

The most expensive stage in any string graph-based assembler is to find the overlaps between all pairs of reads. In theory, one can generate all suffixes and prefixes from R, and create two lists of key-value pairs, S and P, using suffixes and prefixes respectively as keys and their corresponding read-IDs (unique identifiers of reads) as values. Next, for each key-value pair $< k, r_i >$ in S, one can add an edge $e = (r_i, r_j, |k|)$ in G if $< k, r_j >$ exists in P.

Naturally, this is not practical since storing all suffixes (or prefixes) for an input of size n will require $\mathcal{O}(n^2)$ space, where n is the total number of bases in R. The space can be reduced if the actual suffix (or prefix) is replaced by a tuple $< l, f >$ where l is the length of the suffix (or prefix) and f is its fingerprint. Therefore, for each $<< l_i, f_i >, r_j >$ in S, we can conclude that an edge exists between r_j and r_k with a high probability if P contains $<< l_i, f_i >, r_k >$.

However, even with fingerprints, the space requirement (i.e., $\mathcal{O}(n \log_2 q)$ bits where q is the largest fingerprint value) is often too large to fit in the main memory. Indeed, this approach has been attempted [37] but evaluated only on small simulated low-coverage datasets. This problem intensifies in case of large high-coverage real datasets where the reads have a lot of overlaps and require larger fingerprints to minimize the number of false positive edges.

The situation deteriorates with GPUs since their available device memories are typically an order of magnitude smaller than the main memory of even a mid-range workstation. To address this problem, we present a semi-streaming approach for building the string graph by conceptually dividing the memory hierarchy into a read-only memory, a write-only memory, and a working memory, as depicted in Fig. 3. The

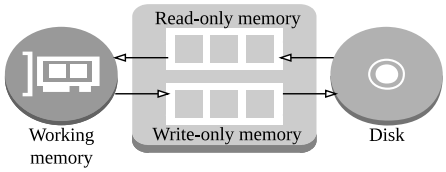

Figure 3. Conceptual view of memory types

read-only memory can only be read sequentially, whereas the write-only memory can only be written sequentially. Note that both the read- and write-only memories reside on disks, but are treated as separate to suggest that a file cannot be read and written at the same time. The working memory consists of a slower host memory and a faster device memory. Unlike the read- and write-only memories, we allow random accesses in the host memory. However, most of the computation is done on the faster device memory, and we minimize the processing done on the host memory. This two-level streaming model (from disk to host memory and from host memory to device memory) efficiently utilizes the memory hierarchy to reduce the amount of disk I/O and enables large-scale sequence assembly.

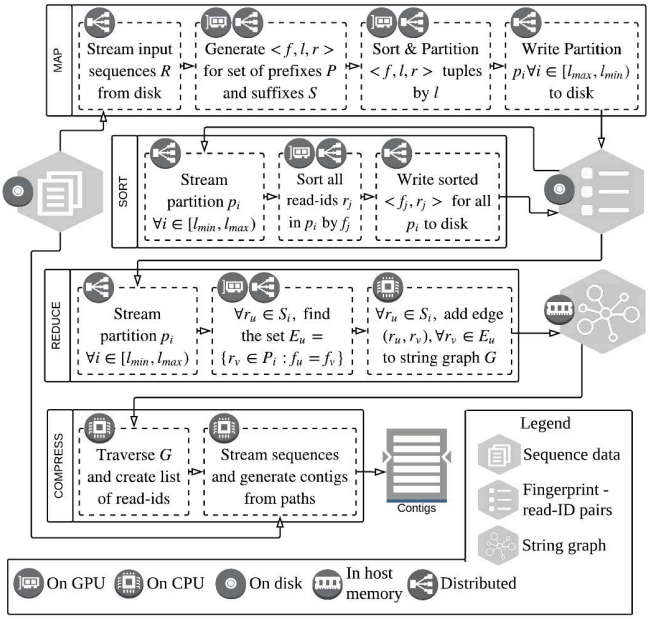

Figure 4. Overview of assembly pipeline

Fig. 4 shows the overview of our assembly pipeline in LaSAGNA consisting of four main phases: map, sort, reduce, and traverse. We describe each phase in detail below.

A. Map: Generate pairs of fingerprints and read-IDs

In this phase, batches of reads are loaded in the GPU where their reverse complements are obtained. Next, for each read and its complement, the fingerprints of all their prefixes and suffixes are generated using the Rabin Karp's

rolling hash. Since short-reads in a sequence dataset are of similar lengths, the fingerprint generation can be easily parallelized by assigning each read to a thread. However, on GPUs, this scheme fails to perform as expected due to excessive memory throttling, a scenario where threads block because of numerous pending memory accesses. Moreover, this scheme does not utilize shared memory.

These issues can be addressed if a block of threads processes each read in parallel. To do so, we express prefix fingerprint generation as a Hillis-Steele scan problem. Here, each batch of reads is processed by a grid of thread blocks where the number of blocks equals the number of reads in the batch, and the number of threads per block equals the read-length. Fig. 5 illustrates the execution of the kernel used to generate fingerprints of all prefixes of a read whose length is 10 with some prime modulo (q) and radix (σ). In practice, the radix is a small prime larger than the alphabet size, and the prime modulo is a large prime number.

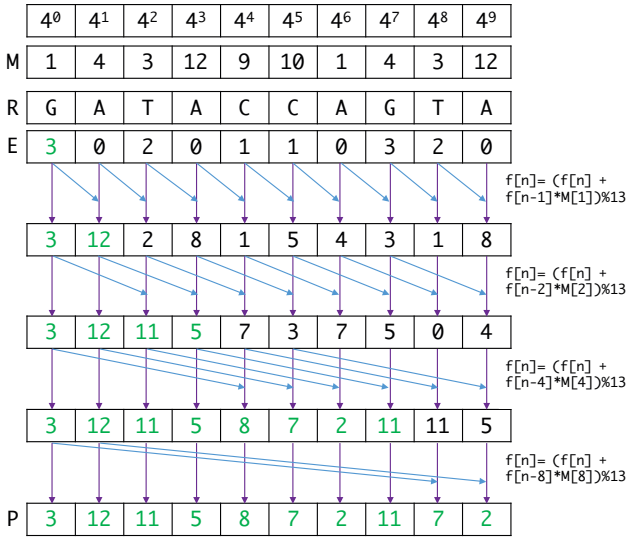

Figure 5. The communication and computing pattern during generation of prefix fingerprints of read GATACCAGTA with radix 4 and prime 13.

Before the kernel is launched, the place values (e.g., σ^0, σ^1) for all positions of the read are computed, and their remainders after division by the prime number q are stored in an array M. This step is done once for the entire program and reused for all reads. In the beginning, each thread in the grid encodes the corresponding base in the read (R=GATACCAGTA) to the radix σ (σ = 4) and places it in its corresponding position in array E. Next, each thread iteratively adds its element to the product of previous element (the current *offset* determines the previous element), and its place value. The *offset* starts at one and is doubled at every step until its length becomes larger than the read itself. At the end of the algorithm, each position i in the output array P will store the fingerprint of the prefix ending at position i (i.e., fingerprint of the prefix of length $i + 1$). For example, in Fig. 5, the fingerprints of G, GA, and GAT are 3, 12, and 11 respectively.

The suffix fingerprints are computed using the place values (M) and prefix fingerprints (P) calculated during the previous step as shown in Fig. 6. As before, the position i will now store the fingerprint of the suffix starting from position i. For instance, in Fig. 6, the fingerprints of *GATACCAGTA* and *ATACCAGTA* are 2 and 5 respectively.

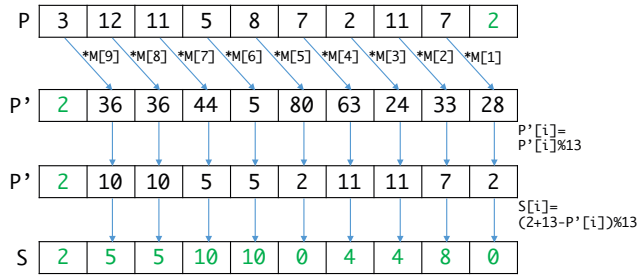

Figure 6. The communication and computing pattern during generation of suffix fingerprints, using the prefix fingerprints of the previous step

This scheme reduces the number of memory transactions and raises the functional unit utilization of GPUs. Moreover, prefixes and suffixes are processed in a single kernel using shared memory, thus improving performance and avoiding scattered writes during suffix fingerprint generation.

Partitioning: To find overlaps between suffixes and prefixes, their lengths (j) and source read-IDs (r_i) are appended to their fingerprints (f_{i_j}) to create a set of 3-tuples. Since we need to find matches between suffixes and prefixes of the same length, it is logical to partition these tuples by their length. Therefore, we sort all (f_{i_j}, r_i) pairs by their corresponding lengths j and count the number of occurrences of each unique j to obtain the size of each partition. In other words, the partitioning process converts a list of (j, f_{i_j}, r_i) tuples to l_{max} lists of (f_{i_j}, r_i) tuples, where l_{max} is the length of the sequences. Out of these lists, the ones corresponding to a length smaller than a user-defined minimum overlap-length (l_{min}) are discarded. Moreover, the partition belonging to l_{max} is also dropped to avoid self-loops in the graph. The remaining ones are written to disk, each into a file corresponding to the partition.

B. Sort: Sort read-IDs by fingerprints

To enable binary searches of suffixes in a list of prefixes, in this phase we sort read-IDs by their fingerprints belonging to each partition corresponding to length $l_i \in [l_{min}, l_{max})$. Since the number of suffixes/prefixes is much larger than the GPU memory, we use an external-memory sorting scheme comprising two phases. In the first phase, chunks of key-value pairs (of size M such that M elements fit in GPU memory) are read from disk, sorted by keys, and written back to disk. In the next phase, these chunks are iteratively merged into a single sorted one.

Note that the size of the chunks being merged will double every iteration, and hence the merging must also be done using external-memory. Algorithm 1 explains our approach to combine two sorted lists on disk, by processing at least $M/2$ and at most M elements at a time. It is adapted from the *k-way merging* scheme and operates under the guiding principle that the input cannot be randomly accessed.

Algorithm 1 External memory merging

Input: A sorted list kvl_A of key-value pairs.
Input: A sorted list kvl_B of key-value pairs.
Input: A count M of key-value pairs that fit in memory
Output: kvl_A and kvl_B are merged into sorted list kvl_C

```
1: procedure MERGE(kvl_A, kvl_B)
2:     repeat
3:         A ← next M/2 key-value pairs from kvl_A
4:         B ← next M/2 key-value pairs from kvl_B
5:         if A ≺ B then kvl_C ← A
6:         else if B ≺ A then kvl_C ← B
7:         else
8:             k ← MIN_KEY(A_{M/2}, B_{M/2})
9:             if k = A_{M/2} then
10:                rank ← UPPER_BOUND(k, B)
11:                RESIZE(B, rank)
12:            else
13:                rank ← UPPER_BOUND(k, A)
14:                RESIZE(A, rank)
15:            end if
16:            kvl_C ← GPU_MERGE(A, B)
17:        end if
18:    until one of the lists is empty
19:    kvl_C ← any remaining elements from A or B
20:    return kvl_C
21: end procedure
```

The merge algorithm consumes two sorted lists of key-value pairs (kvl_A, kvl_B) and emits a single sorted list (kvl_C). We slide windows of size $M/2$ on both lists (lines 3, 4) and read chunks of data from both. If either of the windows reaches the end of the list, the function copies the remaining elements of the other list to the output (line 19) and finishes. Otherwise, when the largest key in one chunk is not larger than the smallest key in the other, all items in the smaller chunk are appended to the output (lines 5, 6). On the other hand, if the chunks cannot be ordered entirely, we resize the windows in a way that the largest key within the current pair of windows is not larger than the smallest key in the subsequent ones. This is done by finding the *upper-bound* of the smaller of the last keys in both chunks, in the chunk containing the larger of those two keys.

The *upper-bound* of a key k in a sorted array A is the index i such that $A[j] \leq k, \forall j < i$ and $A[j] > k, \forall j > i$. This can be easily obtained with a binary search in the host memory (lines 8 to 15). Note that this step is only required if the last elements in both windows are unequal, but this check is omitted from the pseudo-code for brevity. After the windows are *equalized*, the pairs in A and B are copied to device memory where they are merged by keys and appended to the output (line 16).

Algorithm 1 takes $\mathcal{O}(r/m_d)$ iterations where r is the number of keys to be merged, and m_d is the maximum number of elements that fit in the device memory. Since each iteration performs radix sorting in $\mathcal{O}(m_d)$ [38], Algorithm 1 has time complexity $\mathcal{O}(r)$.

Sorting in hybrid-memory: To reduce the number of disk passes during this phase, we use host memory as a buffer between the disk and device memory. This hybrid-memory sort procedure works exactly as before, but in two levels. In the top level, it reads large chunks of size m_h (such that m_h key-value pairs fit in host memory) from disk to host memory, sorts them, and writes them back to disk. Next, these large sorted blocks are streamed from disk and iteratively merged (like Algorithm 1, but in host memory) in batches of $m_h/2$ elements per block, until a single sorted block remains.

Underneath the top level, the sorting and merging of large blocks in host memory are done by streaming them in smaller chunks of m_d to device memory and sorting and merging them on the GPU. This makes up the second level of the scheme. With this optimization, even though the number of merge passes in device remains the same, the number of disk passes can be reduced to $1+\log(n/m_h)$, or by a factor of $\log(m_h/m_d)$, which is typically about 3-4 times.

C. Reduce: Find suffix-prefix matches

This phase consumes two lists S_l and P_l containing tuples of the read-IDs and fingerprints of their l-length suffixes and prefixes respectively, both sorted by fingerprints. For each read-ID in S_l, it searches P_l for read-IDs having the same fingerprint, and adds an edge to the string graph for each matching read-ID. Having sorted lists enables us to stream two windows W_S and W_P from suffixes and prefixes respectively, such that they fit in memory and a fingerprint present in W_S cannot be present in any window except W_P. Therefore, we can process each partition with a single disk pass. We repeat this process for all pairs of S_i and P_i for each $i \in [l_{max}, l_{min})$.

Algorithm 2 shows the pseudo-code of the approach described above. We stream data from the list of suffixes and prefixes into host memory with at most $M/2$ key-value pairs per window (lines 3 and 4). Next, we find the smaller of the largest fingerprints from either window (f) and calculate the *lower-bound* of f in both the windows. The lower-bound of a key k in a sorted array A is the index i such that $A[j] < k, \forall j < i$ and $A[j] \geq k, \forall j > i$. Based on the lower-bound values, we resize both windows (lines 6 and 7) to contain the same range of fingerprints.

Once the windows are resized, we obtain the number of occurrences of each suffix fingerprint in the prefix window. Note that, by the definition of upper- and lower-bounds, if a sorted array A contains p consecutive occurrences of an

Algorithm 2 Overlap detection

Input: A sorted list kvl_{sfx} of suffix-fingerprints & read-IDs
Input: A sorted list kvl_{pfx} of prefix-fingerprints & read-IDs
Input: A count M of key-value pairs that fit in memory
Input: A string graph G
Output: Updated string graph G

```
1: procedure REDUCE(kvl_sfx, kvl_pfx)
2:     repeat
3:         S ← next M/2 key-value pairs from kvl_sfx
4:         P ← next M/2 key-value pairs from kvl_pfx
5:         f ← MIN_KEY(S_{M/2}, P_{M/2})
6:         RESIZE(S, LOWER_BOUND(f, S))
7:         RESIZE(P, LOWER_BOUND(f, P))
8:         L ← GPU_VEC_LOWER_BOUND(S, P)
9:         U ← GPU_VEC_UPPER_BOUND(S, P)
10:        C ← GPU_VEC_DIFFERENCE(U, L)
11:        for r_{s_i} ∈ S do
12:            if c_i ∈ C > 0 then
13:                for j ∈ [L_i, L_i + c_i) do
14:                    G ← (r_{s_i}, r_{p_j}), r_{p_j} ∈ P
15:                end for
16:            end if
17:        end for
18:    until one of the lists is empty
19:    return G
20: end procedure
```

item k starting at index i, the lower-bound of k in A is i, and its upper-bound is $i + p$. On the other hand, if k is not present in A, there must exist an index j such that $A[j] < k < A[j+1]$ and, both the upper- and lower-bounds are j. Thus, the number of occurrences of an element in a sorted array is the difference between its upper- and lower-bound in the array. If the difference is non-zero, the lower-bound yields the position of its first occurrence in the array.

Therefore, we load both the suffix and prefix windows in the GPU and calculate the upper-bound (U) and lower-bound (L) of each suffix fingerprint in the array of prefix fingerprints and the difference (C) between them (lines 8 - 10). We then iterate through C and, for all i such that $c_i > 0, c_i \in C$, we create an edge from the vertex corresponding to the read with suffix $s_i \in S$ to that with prefix $p_j \in P$ for $j \in [L_i, U_i)$.

Our approach of building the graph is greedy, so each vertex (read-ID) will have at most one incoming edge and at most one outgoing edge. We maintain a bit-vector to store the out-degree information of all vertices. Upon receiving a request to add a candidate edge $(u, v, l), l \in [l_{min}, l_{max})$, we check the bit-vector to find out if either the vertex u or $v\prime$ (WC complement of v) has an outgoing edge, and if so, discards the edge. If both vertices have no outgoing edge, we add edges (u, v, l) and $(v\prime, u\prime, l)$ to the graph and update the bit-vector.

Note that the graph is built in the host instead of the GPU since for large datasets it can easily exceed the device memory. For instance, the graph of a human genome with 2.5 billion edges, each containing a 4-byte vertex-ID and a 1-byte overlap length, takes about 12 GB of memory. Besides, due to of the nature of string graphs, adding an edge (u, v) to a graph that is being updated by multiple threads involves acquiring locks for u and $v\prime$. We have observed (using CUDA's native atomics library) that such an approach detrimentally influences the performance when implemented in GPUs.

Like Algorithm 1, Algorithm 2 takes $\mathcal{O}(r/m_d)$ iterations where r is the number of reads, and m_d is the maximum number of elements that can be stored in the device memory. Moreover, the amortized time complexity of each iteration is $\mathcal{O}(m_d \log m_d/p + m_d \times c_{avg})$ where p is the number of processors for the vectorized upper- and lower-bound calculation and c_{avg} is the average number of overlaps per read. Since $\log m_d < p$ in practice (e.g., NVIDIA P100 has 16 GB memory and 3,584 cores), Algorithm 2 has time complexity $\mathcal{O}(r + C)$ where C is the number of overlaps between reads.

D. Compress: Traverse paths and generate contigs

This phase consists of two stages. In the first stage, we traverse the string graph to obtain a set of paths, where each path p_i is represented as a sequence of tuples, and each tuple consists of a read-ID (r_{i_j}) and its overhang-length (l_{i_j}). The overhang-length of a read r_u having an overlap with another read r_v is defined as $(l_u - o_{uv})$ where l_u is the length of r_u and o_{uv} is the overlap between r_u and r_v. Since a vertex in our string graph can have at most one outgoing edge, a read can only have a single overhang-length. Reads having no overlap with others have overhang-lengths equal to their lengths.

Traversal begins with vertices with in-degree 0 and out-degree 1 as seeds. Next, from each seed, we continue to extend the path by appending the read-ID and overhang-length of the current vertex to the sequence of tuples, and stop after we encounter a vertex with no outgoing edge. Since the graph resides in host memory, this stage is performed using multiple threads in the host. However, even for our largest test dataset (i.e., real-world human genome), it takes less than a minute to obtain the paths in host memory, so its effect is insignificant.

In the second stage, we convert read-IDs belonging to the paths in the string graph to their corresponding input sequences to generate contiguous sections (*contigs*) of the original DNA sequence. This is done by laying out the overhang-lengths of paths in order of their read-IDs, streaming the original reads, and placing their overhangs in their appropriate offsets.

Fig. 7 shows the process in greater detail. For any path p_i in the list of paths P, let its length l_i be defined as the number of (r_{i_j}, l_{i_j}) pairs in it. We load P to GPU and, for each $p_i \in P$, we calculate its offset o_i within P using an *exclusive prefix-scan* operation. Next, for all reads r_{i_j} in path p_i, we perform another scan of their overhang-lengths

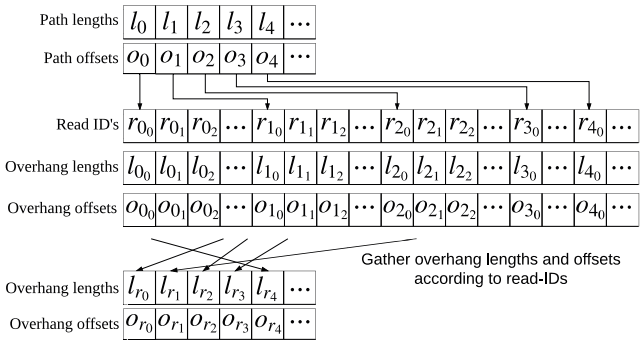

Figure 7. Generating contigs from paths in string graph

l_{i_j} to obtain the size of memory required to store the contig corresponding to p_i, as well as the offset o_{i_j} of each read in the set of all contigs.

Since a vertex has at most one incoming and one outgoing edge, a read can belong to at most one path. Hence, each overhang-offset tuple is copied to the unique location corresponding to its read-ID with a *gather* operation in GPU (i.e., using the array of read-IDs as a stencil). Finally, short-reads are streamed from disk and, for each read r_i, we place the substring $r_i[1 \cdots l_{r_i}]$ into offset o_{r_i} in the memory allocated for contigs. This concludes our contig generation phase.

E. Distributed Graph Building Over Multiple Compute Nodes

Among all phases in the pipeline, sorting takes the longest (more than 50% of the total execution time), closely followed by fingerprint generation (about 25% of the total time). Since fingerprints can be independently generated for each sequence, and each partition is separately sorted, these phases can be distributed across multiple GPUs. However, we have observed that the most prominent bottleneck in the pipeline is the I/O throughput. To mitigate this, we distribute the computation across multiple nodes for exploiting a higher aggregate I/O bandwidth.

The distributed implementation of LaSAGNA assumes the presence of a distributed file system that stores the input sequences and the output graph. Each node also has access to private storage for shuffling and sorting intermediate data. The storage can be located in a network-mounted file system, but must not be shared across nodes since the bulk of the I/O is performed here. In practice, LaSAGNA will benefit from the use of local disks and faster media such as solid-state drives. GASNet [1] active messaging library handles the remote spawning of processes and subsequent communications. Each node runs a process that communicates and synchronizes with peers and calls CUDA functions. One of the peers is chosen as a master and charged with load balancing.

[1]https://gasnet.lbl.gov/

1) Map: In the map phase, each node (including the master itself) requests the master for the address of an input block. It then processes the sequences in the blocks, generates (fingerprint, read-ID) tuples, and writes them into local disk. Like the single-node implementation, the tuples are split into partitions based on the length of suffixes and prefixes, and each partition is maintained in a separate file.

2) Shuffle and sort: Each node works on separate partitions and must aggregate the data assigned to it from other peers before sorting. To do this, each node sends a sequence of active messages to its peers. On reaching the destination, a message reads from the file corresponding to the partition requested and responds with a chunk of data.

3) Reduce: Since edges are added to the graph in descending order of their lengths, and (fingerprint, read-ID) pairs are also partitioned by lengths, a node reducing partition p_i with i-length suffixes/prefixes must wait for the node reducing p_{i+1} to finish. Even with a different partitioning scheme, if multiple nodes greedily update a graph in parallel, it would not be scalable since adding an edge (u, v) needs locks on both u and v, which may reside on different nodes. Nonetheless, some scalability can still be attained using the observation that most of the execution time is spent on reading the suffixes and prefixes from disk and finding the overlaps between them, whereas adding edges is relatively much faster.

We apply this insight to distribute the reduce phase as follows. Each node in charge of a set of partitions processes them in descending order of their corresponding lengths. There exists a bit-vector v that stores the number of outgoing edges of all vertices and initially resides inside the node containing partition $P_{l_{max}-1}$ corresponding to length $l_{max} - 1$. Upon obtaining the set of suffix-prefix overlaps for partition p_i, a node waits until it receives v from the node in charge of p_{i+1}. On arrival of v, the node then uses it to create greedy edges from overlaps and forwards v to the node processing p_{i-1}.

Note that this scheme avoids storing the entire graph in a globally accessible data structure. Instead, it is stored as disjoint sets of edges, and the sets are distributed among the different nodes. On the downside, however, if t_o is the average time spent to find overlaps, and t_g is the average time spent to build the graph (and to send the bit-vector to the next node), the expected total time to finish processing p partitions using an n-node cluster is $t_o \times p/n + t_g \times p$. The scalability of this phase is therefore limited to $n_{max} = t_o/t_g$ nodes, after which the graph building step dominates the computation time.

IV. Evaluation

In this section, we report the experimental results of LaSAGNA for various real-world sequence datasets across multiple computing environments.

A. Datasets

Table I shows the datasets used in the evaluation of LaSAGNA. All datasets used were obtained from Illumina sequencing machines, and the longest sequence length varies from 100 to 150 base pairs. The minimum overlap lengths used to build the string graphs are 63 for H.Chr 14 and H.Genome (lengths of 101 and 100 respectively), 85 for Bumblebee (length of 124), and 111 for Parakeet (length of 150), as suggested by the SGA assembler.

Table I
ILLUMINA DATASETS USED FOR EVALUATION

Dataset	Length	Reads	Bases	Size
H.Chr 14 [39]	101	45,711,162	4,559,613,772	9.2 GB
Bumblebee [39]	124	316,172,570	33,562,702,234	85 GB
Parakeet[2]	150	608,709,922	91,306,488,300	203 GB
H.Genome[3]	100	1,247,518,392	124,751,839,200	398 GB

B. Testbeds and Implementation

To evaluate LaSAGNA, we use three types of computing environments. On *QueenBee II*[4] cluster, we use a node with two NVIDIA Tesla K40 GPUs, two 10-core 2.8 GHz E5-2680v2 Xeon processors, and 128 GB main memory. We use only one GPU for our evaluation. On *SuperMic*[5] cluster, we use a node with one NVIDIA Tesla K20X GPU, two 10-Core 2.8GHz Ivy E5-2680 Xeon processors, and 64 GB main memory. For evaluating the distributed version of LaSAGNA, we use multiple nodes of this type, connected by 56 Gb/s InfiniBand. On *NVIDIA PSG*[6] cluster, we use three types of nodes, each equipped with NVIDIA V100, P100, or P40 GPUs. Each node has two 16-core Haswell E5-2698v3 processors and 256 GB main memory. Even though this cluster provides several high-end GPUs, we could not use them for the entire assembly pipeline because of the limited disk space per account.

LaSAGNA uses 128-bit fingerprints (two 64-bit values generated with different radixes and primes) since we have observed that it yields zero false positive edges across all the datasets that we have tested. LaSAGNA is built primarily with the programming primitives and data structures provided by NVIDIA's Thrust [7] library included with CUDA 7.5 toolkit.

C. Single-GPU Genome Assembly

1) Genome Assembly Times: Table II and Table III show the total assembly times with details of each phase for different datasets on a QueenBee II node (128 GB host memory and one NVIDIA K40 with 12 GB device memory) and on a SuperMic node (64 GB host memory and one NVIDIA K20X with 6 GB device memory) respectively. The results demonstrate that LaSAGNA can finish the entire genome assembly pipeline for a 398 GB human genome dataset within 16.5 hours using a single GPU (NVIDIA K40). To the best of our knowledge, this is the first GPU-based assembler that can assemble a real-world human genome dataset on a single node.

Table II
SINGLE NODE ASSEMBLY TIMES ON 128GB HOST MEMORY AND 12GB DEVICE MEMORY (K40)

	H.Chr 14	Bumblebee	Parakeet	H.Genome
Map	5m 32s	33m 20s	1h 40m 58s	2h 43m 15s
Sort	9m 36s	1h 21m 0s	4h 57m 56s	11h 05m 45s
Reduce	4m 47s	26m 6s	1h 17m 31s	2h 20m 33s
Compress	6s	20s	26s	57s
Load	25s	3m 9s	5m 57s	10m 39s
Total	**20m 26s**	**2h 23m 55s**	**8h 2m 48s**	**16h 21m 09s**

Table III
SINGLE NODE ASSEMBLY TIMES ON 64GB HOST MEMORY AND 6GB DEVICE MEMORY (K20)

	H.Chr 14	Bumblebee	Parakeet	H.Genome
Map	5m 59s	36m 8s	1h 47m 58s	2h 50m 28s
Sort	11m 12s	1h 35m 25s	5h 41m 23s	14h 53m 21s
Reduce	4m 26s	27m 35s	1h 14m 13s	2h 31m 43s
Compress	5s	19s	26s	56s
Load	23s	2m 51s	5m 31s	11m 48s
Total	**22m 5s**	**2h 42m 18s**	**8h 49m 31s**	**20h 28m 16s**

Even with a smaller host and device memory (64 GB and 6 GB respectively), LaSAGNA can assemble the H.Genome dataset in 20.5 hours. Note that the execution times increase only slightly for all other datasets except H.Genome. This is because, for H.Genome, a host with 128 GB memory can sort an entire partition (2.5B keys) in a single disk pass, but one with 64 GB memory needs to merge two sorted lists with an additional disk pass. The run-times of the other phases are similar on both machines because LaSAGNA performs the same amount of I/O.

2) Memory usage: Table IV shows the peak host and device memory usage during the various phases of assembly for different datasets on a QueenBee II node with 128 GB host memory and one NVIDIA K40 with 12 GB device memory. Table V shows the results when we run on a SuperMic node with 64 GB host memory and one NVIDIA K20 with 6 GB device memory. In both cases, the device memory usage is almost identical for all datasets because a fixed amount of device memory is allocated for each phase regardless of the data size, and the device memory assigned is fully utilized except for H.Chr 14 on K40. On the other hand, the host memory usage exhibits wide variations depending on the dataset because we maximize the host memory usage. Note that, during the sorting phase

[2]https://trace.ncbi.nlm.nih.gov/Traces/sra/?study=ERP002324
[3]http://www.ncbi.nlm.nih.gov/sra/?term=SRA000271
[4]http://www.hpc.lsu.edu/resources/hpc/system.php?system=QB2
[5]http://www.hpc.lsu.edu/resources/hpc/system.php?system=SuperMIC
[6]http://psgcluster.nvidia.com/
[7]https://developer.nvidia.com/thrust

Table IV
PEAK MEMORY USAGE (IN GB) ON 128 GB HOST WITH K40

Dataset	Peak Host Memory				Peak Device Memory		
	Map	Sort	Red.	Contig	Map	Sort	Reduce
H.Chr 14	14.48	14.92	16.87	16.78	10.74	6.46	4.89
Bumblebee	14.64	34.40	19.55	22.14	10.74	9.02	4.92
Parakeet	16.82	59.21	28.64	28.39	10.73	9.02	4.92
H.Genome	16.39	103.73	38.11	44.24	10.73	9.02	4.92

Table V
PEAK MEMORY USAGE (IN GB) ON 64 GB HOST WITH K20

Dataset	Peak Host Memory				Peak Device Memory		
	Map	Sort	Red.	Contig	Map	Sort	Reduce
H.Chr 14	7.23	9.71	8.99	9.01	5.41	4.54	2.47
Bumblebee	9.03	30.04	13.34	18.14	5.41	4.54	2.50
Parakeet	8.84	54.20	19.48	22.79	5.40	4.54	2.50
H.Genome	9.18	54.66	31.31	38.95	5.40	4.54	2.50

of H.Genome on 128 GB RAM, the maximum available memory is used to minimize the number of disk passes although LaSAGNA succeeds even on 64 GB RAM. The host memory used during the contig generation phase depends on the size of the graph and contigs.

3) Performance Comparisons: Table VI compares the assembly times (in seconds) of LaSAGNA with those of SGA (version 0.10.15), the representative string graph-based assembler. We choose SGA since, to the best of our knowledge, it is the only string graph-based assembler that can handle large datasets on a single node. We do not include the results of de Bruijn graph-based assemblers because most of them are not designed for processing large datasets on a single machine (i.e., failed with out-of-memory error). The SGA pipeline consists of multiple phases including error correction, but for fair comparisons, we only consider the *preprocess*, *index*, and *overlap* phases. We built the index with the *ropebwt* algorithm, which is faster and takes a smaller amount of memory. Our results demonstrate that LaSAGNA is 1.89x-3.05x faster than SGA while also being more memory-efficient due to a better utilization of the memory hierarchy.

Table VI
COMPARISON BETWEEN SGA AND LASAGNA

Dataset	SGA		LaSAGNA	
	64 GB	128 GB	64 GB	128 GB
H.Chr 14	3081s	3039s	1325s (**2.33**×)	1226s (**2.48**×)
Bumblebee	26360s	23958s	9738s (**2.71**×)	8635s (**2.77**×)
Parakeet	93747s	88229s	31771s (**2.95**×)	28968s (**3.05**×)
H.Genome	**OOM**	111024	73696s	58869s (**1.89**×)

4) Effects of Host and Device Memory Sizes in Sorting: We also study the effects of host and device memory sizes on the sorting phase, the most time-consuming task in our assembly pipeline. For these experiments, we use the data generated from H.Genome, containing about 2.5 billion pairs of 128-bit keys and 32-bit values per partition. We use the term *block-size* to denote the number of key-value pairs in a block. Fig. 8 shows the effects of using different block-sizes in host and device on the average sorting time per partition on an NVIDIA K40 GPU. It shows that increasing the device block-size decreases the execution time since it reduces the number of merge passes on the GPU. More importantly, it shows that the effect of device block-sizes is small compared to that of host block-sizes. The significant performance improvement with a larger host block-size (because of fewer disk passes) indicates that the I/O is the most critical factor in sorting. Note that, since the sorting can be performed in a single disk pass with a host block-size of 2.56 billion, we do not expect any performance improvement beyond this point.

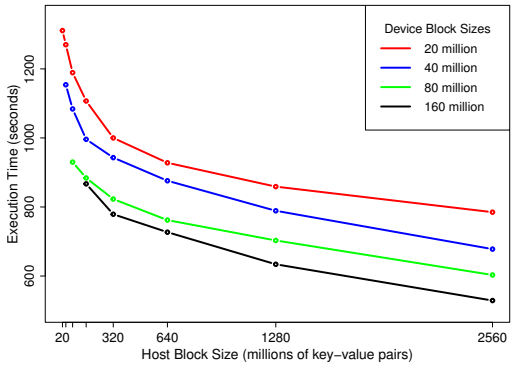

Figure 8. Effects of different host and device block-sizes

5) Effects of Different GPUs: Fig. 9 shows the average sorting times for various host block-sizes with a fixed device block-size of 20 million on different GPUs (K40, P40, P100, and V100). Increasing the host block-size results in a logarithmic decrease in the execution time until it reaches 2.56 billion. V100, NVIDIA's latest high-end GPU, is the fastest since it has much more CUDA cores (5,120) at higher GPU Boost clock rate (1530 MHz), and provides the highest peak memory bandwidth (900 GB/sec). Interestingly, P40 is consistently slower than P100 even though P40 has a larger device memory (24 GB vs. 16 GB) and more cores (3840 vs. 3584). This is because P40 has much smaller memory bandwidth than P100 (346 GB/s vs. 732 GB/s).

Another interesting observation is that, as the host block-size decreases (increasing the number of disk passes), the performances of the different GPUs appear to converge since the sorting phase increasingly becomes more I/O bound. This reinforces the efficacy of our hybrid-memory model in saturating GPU performance during the sorting phase. We also argue that the hybrid-memory model can apply to other types of workloads (e.g., MapReduce-like processing) that require sorting since we can process large-scale datasets for such workloads on a single GPU (or a few GPUs) in a scalable and efficient manner.

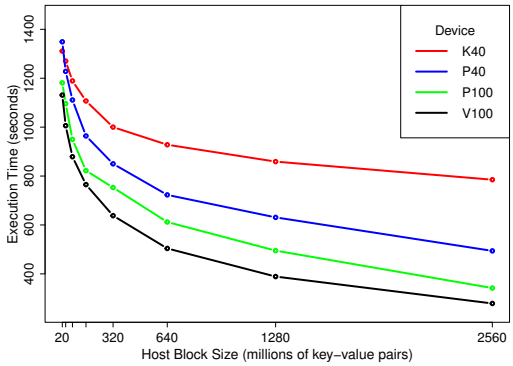

Figure 9. Effects of different GPUs and host block-sizes

D. Distributed Execution Times

To evaluate the scalability of LaSAGNA, we report the execution times of the different phases by varying the number of compute nodes from 1 to 8 on the SuperMic cluster, as shown in Fig. 10. The results show that LaSAGNA can assemble a 398 GB human genome dataset in a little over 5 hours on an 8-node cluster of NVIDIA K20s. LaSAGNA performs better with more nodes because of the distribution of data and computations during the map phase, and the larger aggregated I/O bandwidth during the sort phase.

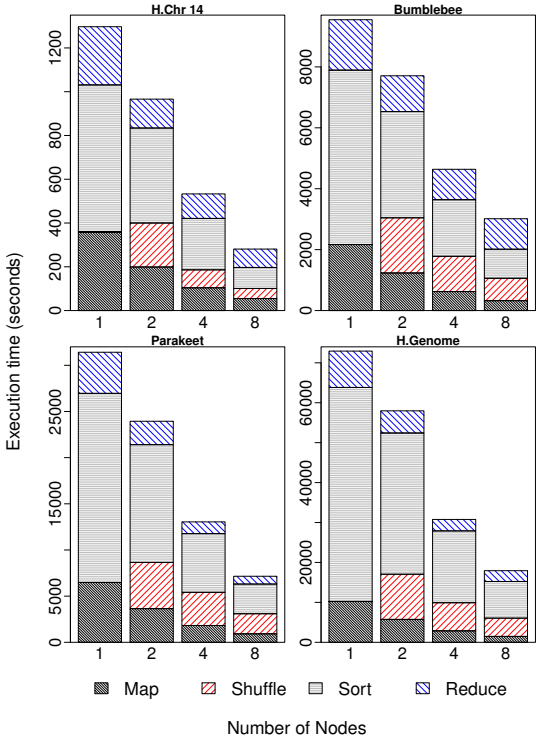

Figure 10. Execution times on different node configurations

However, LaSAGNA does not demonstrate the optimal scalability because scaling out from a single node introduces the additional overhead of an all-to-all data transfer before the sorting phase, as shown in Fig. 10. Moreover, as previously explained, the data dependency among GPUs in the graph-building stage of the reduce phase limits the scalability of the current implementation. To resolve this, are working on partitioning the suffixes/prefixes based on their fingerprints rather than on lengths. We also plan on processing the string graph in parallel using a bulk-synchronous processing model. We leave further optimizations to the shuffle and reduce phases as our future work.

V. Conclusion

In this paper, we have presented LaSAGNA, a new GPU-accelerated genome assembler, that can assemble large-scale sequence datasets on a single GPU by constructing string graphs from approximate overlaps using fingerprints. Our approach uses a two-level semi-streaming model that exploits the speed of GPU device memory as well as the large capacity of host memory. Our experimental results demonstrate that LaSAGNA can assemble a 400 GB human genome dataset in 17 hours using a single GPU (NVIDIA K40). In a distributed setting, LaSAGNA can finish the entire genome assembly pipeline for the dataset in a little over 5 hours using an 8-node cluster. To the best of our knowledge, LaSAGNA is the first GPU-based assembler that can assemble a real human genome dataset on a single node.

Acknowledgment

This work was partially funded by NIH grants (P20GM103458-10, P30GM110760-03, P20GM103424), NSF grants (MRI-1338051, IBSS-L-1620451, SCC-1737557, RAPID-1762600), LA Board of Regents grants (LEQSF(2016-19)-RD-A-08 and ITRS), and IBM faculty awards. The authors would like to thank NVIDIA for its generosity in allowing them to use their compute resources.

References

[1] B. M. Institute, "The Impact of Genomics on the U.S. Economy," 2013.

[2] R. A. Power, J. Parkhill, and T. de Oliveira, "Microbial genome-wide association studies: lessons from human gwas," *Nature Reviews Genetics*, vol. 18, no. 1, pp. 41–50, 2017.

[3] M. Kayser and P. De Knijff, "Improving human forensics through advances in genetics, genomics and molecular biology," *Nature Reviews Genetics*, vol. 12, no. 3, pp. 179–192, 2011.

[4] A. Acevedo, L. Brodsky, and R. Andino, "Mutational and fitness landscapes of an rna virus revealed through population sequencing," *Nature*, vol. 505, no. 7485, pp. 686–690, 2014.

[5] J. Pantaleoni and N. Subtil. (2014) NVBIO A library of reusable components designed by NVIDIA corporation to accelerate bioinformatics applications using CUDA.

[6] Y. Liu, B. Schmidt, and D. L. Maskell, "CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform," *Bioinformatics*, vol. 28, no. 14, pp. 1830–1837, 2012.

[7] Y. Liu and B. Schmidt, "CUSHAW2-GPU: empowering faster gapped short-read alignment using GPU computing," *IEEE Design & Test*, vol. 31, no. 1, pp. 31–39, 2014.

[8] P. Klus, S. Lam, D. Lyberg, M. S. Cheung, G. Pullan, I. McFarlane, G. S. Yeo, and B. Y. Lam, "BarraCUDA-a fast short read sequence aligner using graphics processing units," *BMC research notes*, vol. 5, no. 1, p. 27, 2012.

[9] C.-M. Liu, T.-W. Lam, T. Wong, E. Wu, S.-M. Yiu, Z. Li, R. Luo, B. Wang, C. Yu, X. Chu *et al.*, "SOAP3: GPU-based compressed indexing and ultra-fast parallel alignment of short reads," in *3th Workshop on Massive Data Algorithms*, 2011.

[10] J. Blom, T. Jakobi, D. Doppmeier, S. Jaenicke, J. Kalinowski, J. Stoye, and A. Goesmann, "Exact and complete short-read alignment to microbial genomes using Graphics Processing Unit programming," *Bioinformatics*, vol. 27, no. 10, pp. 1351–1358, 2011.

[11] Y. Chan, K. Xu, H. Lan, W. Liu, Y. Liu, and B. Schmidt, "PUNAS: A Parallel Ungapped-Alignment-Featured Seed Verification Algorithm for Next-Generation Sequencing Read Alignment," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[12] S. Chen and H. Jiang, "An exact matching approach for high throughput sequencing based on bwt and gpus," in *IEEE 14th International Conference on Computational Science and Engineering (CSE)*, 2011.

[13] A. Drozd, N. Maruyama, and S. Matsuoka, "A multi GPU read alignment algorithm with model-based performance optimization," in *International Conference on High Performance Computing for Computational Science*. Springer, 2012, pp. 270–277.

[14] J. Wang, X. Xie, and J. Cong, "Communication Optimization on GPU: A Case Study of Sequence Alignment Algorithms," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[15] M. Lu, Q. Luo, B. Wang, J. Wu, and J. Zhao, "GPU-accelerated bidirected De Bruijn graph construction for genome assembly," in *Asia-Pacific Web Conference*. Springer, 2013, pp. 51–62.

[16] P. Bonizzoni, G. Della Vedova, Y. Pirola, M. Previtali, and R. Rizzi, "FSG: Fast String Graph Construction for De Novo Assembly," *Journal of Computational Biology*, 2017.

[17] P. A. Pevzner, H. Tang, and M. S. Waterman, "An Eulerian path approach to DNA fragment assembly," *Proceedings of the National Academy of Sciences*, vol. 98, no. 17, pp. 9748–9753, 2001.

[18] E. W. Myers, "The fragment assembly string graph," *Bioinformatics*, vol. 21, no. suppl 2, pp. ii79–ii85, 2005.

[19] G. Jain, L. Rathore, and K. Paul, "GAMS: Genome Assembly on Multi-GPU Using String Graph," in *IEEE 18th International Conference on High Performance Computing and Communications (HPCC)*, 2016.

[20] M. de la Bastide and W. R. McCombie, "Assembling genomic DNA sequences with PHRAP," *Current Protocols in Bioinformatics*, 2007.

[21] 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, pp. 2942–2944, 2007.

[22] S. F. Mahmood and H. Rangwala, "Gpu-euler: Sequence assembly using gpgpu," in *IEEE 13th International Conference on High Performance Computing and Communications (HPCC)*, 2011, pp. 153–160.

[23] A. Jain, A. Garg, and K. Paul, "GAGM: Genome assembly on GPU using mate pairs," in *20th International Conference on High Performance Computing (HiPC)*. IEEE, 2013, pp. 176–185.

[24] B. S. C. Varma, K. Paul, M. Balakrishnan, and D. Lavenier, "Fassem: Fpga based acceleration of de novo genome assembly," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. IEEE, 2013, pp. 173–176.

[25] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de Bruijn graphs," *Genome research*, vol. 18, 2008.

[26] R. Luo, B. Liu, Y. Xie, Z. Li, W. Huang, J. Yuan, G. He, Y. Chen, Q. Pan, Y. Liu *et al.*, "SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler," *Gigascience*, vol. 1, no. 1, p. 18, 2012.

[27] R. Chikhi and G. Rizk, "Space-efficient and exact de Bruijn graph representation based on a Bloom filter," *Algorithms for Molecular Biology*, vol. 8, no. 1, p. 22, 2013.

[28] D. Li, C.-M. Liu, R. Luo, K. Sadakane, and T.-W. Lam, "Megahit: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph," *Bioinformatics*, vol. 31, no. 10, pp. 1674–1676, 2015.

[29] S. Boisvert, F. Laviolette, and J. Corbeil, "Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies," *Journal of computational biology*, vol. 17, no. 11, pp. 1519–1533, 2010.

[30] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji, "SWAP-Assembler: scalable and efficient genome assembly towards thousands of cores," in *BMC bioinformatics*, vol. 15, no. Suppl 9. BioMed Central Ltd, 2014.

[31] S. Goswami, A. K. Das, R. Platania, K. Lee, and S.-J. Park, "Lazer: Distributed memory-efficient assembly of large-scale genomes," *IEEE International Conference on Big Data (Big Data)*, pp. 1171–1181, 2016.

[32] M. C. Schatz, D. Sommer, D. Kelley, and M. Pop, "De novo assembly of large genomes using cloud computing," in *Proceedings of the Cold Spring Harbor Biology of Genomes Conference*, 2010.

[33] P. Koppa, A. Das, S. Goswami, R. Platania, and S. Park, "Giga: Giraph-based genome assembler for gigabase scale genomes," in *Proceedings of the 8th International Conference on Bioinformatics and Computational Biology (BICOB 2016)*, 2016, pp. 55–63.

[34] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, R. Egan, L. Oliker, D. Rokhsar, and K. Yelick, "HipMer: an extreme-scale de novo genome assembler," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–11.

[35] J. T. Simpson and R. Durbin, "Efficient de novo assembly of large genomes using compressed data structures," *Genome research*, vol. 22, no. 3, pp. 549–556, 2012.

[36] Y.-J. Chang, C.-C. Chen, C.-L. Chen, and J.-M. Ho, "A de novo next generation genomic sequence assembler based on string graph and MapReduce cloud computing framework," *BMC genomics*, vol. 13, no. 7, p. S28, 2012.

[37] I. Ben-Bassat and B. Chor, "String graph construction using incremental hashing," *Bioinformatics*, vol. 30, no. 24, pp. 3515–3523, 2014.

[38] D. G. Merrill and A. S. Grimshaw, "Revisiting sorting for gpgpu stream architectures," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 545–546.

[39] S. L. Salzberg, A. M. Phillippy, A. Zimin, D. Puiu, T. Magoc, S. Koren, T. J. Treangen, M. C. Schatz, A. L. Delcher, M. Roberts *et al.*, "GAGE: A critical evaluation of genome assemblies and assembly algorithms," *Genome research*, vol. 22, no. 3, pp. 557–567, 2012.