Memory-efficient all-pair suffix-prefix overlaps on GPU

Sayan Goswami[0000−0002−7519−524X]

Ahmedabad University, Ahmedabad, India
sayan.goswami@ahduni.edu.in

Abstract. Obtaining overlaps between all pairs from billions of strings is a fundamental computational challenge in de novo whole genome assembly. This paper presents a memory-efficient, massively parallel algorithm that uses GPUs to find overlaps from large sequencing datasets with a very low probability of false positives. Here we use a Rabin-fingerprint-based indexing method which stores the strings with their fingerprints and uses them to generate the fingerprints of suffixes and prefixes. We then sort these fingerprints in hybrid CPU-GPU memory and stream them in GPU to find matches. Experiments show that our implementation can detect a trillion highly probable overlaps within 1.5 billion DNA fragments in just over two hours. Compared to the existing CPU-based approach, it routinely achieves speedups of 5-15x while having a collision rate of 1 in 20 million.

Keywords: Big data · Parallel processing · Memory management · GPU · Genome assembly

1 Introduction

The knowledge of the structure of DNA can be utilized in a vast range of applications such as personalized medicine, epidemiology, evolution, food safety and many more [6]. Deciphering the order of nucleotides in the DNA of a novel organism consists of two steps. First, a biochemical process called *sequencing* clones the DNA, fragments them randomly, and parses fragments into short (100-300 characters long) strings called *reads* or *short-reads*. Next, a computational process called *de novo assembly* is employed to find suffix-prefix overlaps between all pairs of reads and merge overlapping reads to recreate the original sequence. Sequencing projects routinely produce datasets with billions of reads. Assembling such datasets, and more specifically, finding all-pair suffix-prefix overlaps (*APSPO*) between billions of strings, is computationally expensive. On a large dataset, this step could take nearly a day, even on high-end servers.

In recent times, Graphics Processing Units (GPUs) have seen widespread adoption in general-purpose data processing tasks owing to a higher performance-per-dollar[4] and performance-per-watt[3] compared to CPUs. In bioinformatics, several programs that have been ported to GPUs report significant performance

improvements[15,13,14]. GPU-based implementations of hash tables [12] and suffix arrays [1] have also been used in bioinformatics applications. However, for de novo assembly, fewer GPU-based implementations have been reported in the literature, and most of them have only been evaluated on small datasets owing to their limited memory capacities.

In this paper, we present *Empanada*[1], a faster and more memory-efficient GPU-based approach to finding APSPO in large sequencing datasets using an inverted index of $\mathcal{O}(n)$ bits. Our main contributions in this work are as follows. (1) We reduce the index size using a previously unexplored approach of obtaining prefix hashes and avoiding storing them explicitly. (2) We implement this method on GPUs using NVIDIA Thrust, which also enables it to also run in parallel on CPUs. (3) Our implementation can handle datasets much larger than what can be stored in GPU memory and is amenable to out-of-core execution using secondary storage. (4) We evaluate our implementation using several real datasets and observe speedups of two orders of magnitude compared to others. Using our approach, we found overlaps in datasets with about 3 billion strings in a little over 3 hours on a single node where other tools ran out of time/space. (5) Finally, our implementations for sorting and searching in hybrid (host-device) memory may be helpful in other applications where the data fits in the host but not the device.

The rest of the paper is organized as follows. In Section 2, we give some background on de novo assembly and provide a brief exposition of existing works. Next, we explain our methodology in Section 3, followed by a description of our implementation details in Section 4. In Section 5, we evaluate our implementation using real-world datasets of various sizes and compare the execution times with previous works and finally present our conclusions in Section 6.

2 Background & Related Work

A *base* is an element of the set $\Sigma = \{A, C, G, T\}$. A *read* is a string over the alphabet Σ. The *Watson-Crick (WC) Complement*, also called *reverse complement*, of a read p of length l is a read p' of the same length such that $p'[i] = p[l-1-i]'$, where $A' = T$, $T' = A$, $C' = G$ and $G' = C$. An *overlap graph* is a directed weighted graph $\mathcal{G}$ obtained using the APSPO in a set of short-reads R and their WC complements R' such that a vertex corresponds to $s_i \in R \cup R'$ and an edge (u, v, w) exists from u to v iff the w-length suffix of the read corresponding to u is the same as the w-length prefix of the read at v. Naturally, for every edge $e = (u, v, w)$ in $\mathcal{G}$, there must also exist a complementary edge $e' = (v', u', w)$.

Given three vertices p, q, and r and edges $e_i = (p, q, w_i)$, $e_j = (q, r, w_j)$, and $e_k = (p, r, w_k)$ in $\mathcal{G}$, e_k is a *transitive edge* if $w_k = w_i - (|q| - w_j)$. Transitive edges can be removed from an overlap graph without any loss of information. A *string graph* is an overlap graph with all transitive edges removed. In de novo assembly, the string graph is refined and chains in the graph (i.e. vertices with

[1] The code is available at https://github.com/sayangoswami/empanada

degree ≤ 1) are collapsed. The reads corresponding to the collapsed vertices are merged using their edge weights, and the resulting strings are called *contigs*.

The problem of finding APSPO has been studied extensively, and a time-optimal solution [8] using suffix trees has been known since 1992. The high memory overhead of suffix trees has led to approaches using enhanced suffix arrays which are faster and more memory efficient. Edena [10], the first available implementation of a string graph-based de novo assembler, uses a suffix array.

Although suffix arrays are more efficient than suffix trees, their space usage is still prohibitively large. This necessitated the development of compressed datastructures such as the compressed suffix array and other compressed full-text indices like the FM-index. SGA [16] is another string graph assembler that uses the FM-index and is capable of handling reasonably large datasets.

Dinh et al. have proposed a memory-efficient prefix tree to represent exact-match overlap graphs and use it to build the LEAP assembler[2]. Compact prefix trees have also been used by Rachid et al., who use an enhanced B-Tree to find strings with a specified prefix[9]. One of the most popular string graph assemblers, Readjoiner [5], avoids building the entire overlap graph by partitioning the suffixes and prefixes and processing each partition independently, resulting in significant improvements in time and memory usage.

GAMS[11] is the first GPU-based string graph assembler and can run on multiple nodes but is only evaluated on small datasets. LaSAGNA[7] was the first reported GPU-based string-graph builder that could handle large datasets using an inverted index from the fingerprints of suffixes and prefixes. It builds the index out-of-core and stores it on disk, thus reducing the memory usage on GPU. However, this index takes $\mathcal{O}(n \log n)$ bits of space on disk for $2n$ bits of data. Therefore this approach spends most of the execution time transferring the index from disk to GPU memory, resulting in low GPU utilization.

This work presents a memory-efficient, parallel, scalable solution that improves upon the fingerprint-based indexing of LaSAGNA by reducing the size of the index to $\mathcal{O}(n)$ bits. This version of the index stores the fingerprints of entire reads from which the fingerprints of suffixes/prefixes are obtained in $\mathcal{O}(n)$ time.

3 Methodology

In this section we present our approach for a space-efficient inverted index. We assume that reads in a dataset are of the same length l, which is the case for Next Generation Sequencing (NGS) machines such as Illumina. We use Rabin-fingerprint to create hashes of all reads in $\mathcal{R} \cup \mathcal{R}'$ as follows. We choose a large prime q and define an encoding $e : \Sigma \to \mathbb{N}$ which maps every base to $[0, \sigma)$. The fingerprint of a base b is given by $e(b)$. Given the fingerprint $f_{i,m}$ of some m-length substring starting at index i of some read s, we have the following:

$$f_{i,m+1} = (f_{i,m}\sigma + e(s[i+m])) \bmod q \tag{1}$$

$$f_{i+1,m-1} = (f_{i,m} + q - e(s[i])\sigma^{m-1} \bmod q) \bmod q \tag{2}$$

Typically, the rolling hash window expands towards its right and shrinks from left. Using Eq. 1, if we start with the fingerprint $f_{0,1}$ of the leftmost base of a read and successively expand the fingerprint-window, we get the fingerprints of all prefixes and eventually that of the entire read $f_{0,l}$. Similarly, using Eq. 2, if we start from the fingerprint of the entire read and successively shrink the window, we will get the fingerprints of all suffixes.

To store the fingerprints of all prefixes and suffixes as is in LaSAGNA[7], one needs $\mathcal{O}(n \log q) = \mathcal{O}(n \log n)$ bits (q is typically in the order of $\mathcal{O}(r^2)$, where $r = |\mathcal{R} \cup \mathcal{R}'| = n/l$). Instead, our approach only uses $\mathcal{O}(n)$ bits by storing the fingerprints of entire reads and regenerating the fingerprints of prefixes and suffixes from them. Regenerating the suffix fingerprints is straightforward (from Eq. 2). Our contribution is the procedure for regenerating prefix fingerprints from reads and their fingerprints which is as follows.

Given a fingerprint $f_{i,m+1}$ of a read s, our goal is to find $f_{i,m}$. Note that from Eq. 1, we have $\sigma f_{i,m} \equiv (f_{i,m+1} - e(s[i+m])) \pmod q$. Expressed as a linear diophantine equation,

$$\sigma f_{i,m} - qk = f_{i,m+1} - e(s[i+m]) \text{ for some } k \in \mathbb{N} \tag{3}$$

Since σ and q are co-primes, the solutions for $f_{i,m}$ in Eq. 3 belong to exactly one congruence class modulo q, which is

$$f_{i,m} \equiv \sigma^{-1}(f_{i,m+1} - e(s[i+m]) \pmod q$$

where σ^{-1} is the multiplicative inverse of σ in the field $\mathbb{Z}/q$. As q is a prime and σ is non-zero, σ^{-1} exists and is equal to $\sigma^{q-2} (\mathrm{mod} q)$ which can be calculated using binary exponentiation in $\log q$ steps. Moreover, by definition $f_{i,m} < q$ which implies

$$f_{i,m} = (\sigma^{-1}(f_{i,m+1} - e(s[i+m])) \bmod q \tag{4}$$

We can now use the reads and their fingerprints to find overlaps as follows. We iterate through every read $s_i \in \mathcal{R} \cup \mathcal{R}'$ and create a tuple $(f_{0,l}^{(i)}, i)$, where $f_{0,l}^{(i)}$ is the fingerprint of s_i. Each fingerprint $f_{0,l}^{(i)}$ is appended to an array $\mathcal{S}$ and each tuple $(f_{0,l}^{(i)}, i)$ is appended to an array $\mathcal{P}$. We also store all reads in an array such that any base $b_{i,j}$ at position j of a read s_i can be obtained in constant time.

Next, we find suffix-prefix overlaps in descending order of overlap lengths like so. We replace each item $f_{0,l}^{(i)} \in \mathcal{S}$ with $f_{1,l-1}^{(i)}$ using Eq. 2. We also replace each tuple $(f_{0,l}^{(i)}, i) \in \mathcal{P}$ with $(f_{0,l-1}^{(i)}, i)$ using Eq. 4. Finally, we sort the tuples in $\mathcal{P}$ by their fingerprints and perform a binary search of all elements of $\mathcal{S}$ in $\mathcal{P}$. If some $f_{1,l-1}^{(i)} \in \mathcal{S}$ is equal to $f_{0,l}^{(j)} \in \mathcal{P}$ we report with high confidence that read s_i has an $(l-1)$-length overlap with s_j. This process is repeated successively for lengths $l-2, l-3, \cdots, l_{min}$, where l_{min} is the minimum overlap length.

Calculating the fingerprints of reads take $\mathcal{O}(n)$ time, where n is the number of bases. At each overlap-detection round, obtaining the fingerprints for suffixes and prefixes for r reads takes $\mathcal{O}(r)$ time and sorting and searching take $\mathcal{O}(r \log r)$ time. Since there are $\mathcal{O}(l)$ rounds, the total time required to find occurrences

is $\mathcal{O}(n \log r)$. Checking whether the occurrences are true positives and adding the edges in the overlap graph take $\mathcal{O}(el)$ time, where e is the total number of edges. Thus, the total time required is $\mathcal{O}(n \log r + el)$. A greedy approach of converting an overlap graph to a string graph is to only retain the longest overlaps for each read. Here, in each round we could filter out those reads from $\mathcal{S}$ and $\mathcal{P}$ which already have an edge. In that case, $e = \mathcal{O}(r)$ and the total time is $\mathcal{O}(n \log r)$. Storing $\mathcal{S}$ and $\mathcal{P}$ takes $(2r \log q + r \log r)$ bits of space which is $\mathcal{O}(n)$ since $\log r$ and $\log q$ are typically less than l. Storing the original reads requires an additional $2n$ bits, since $|\Sigma| = 4$.

4 GPU Implementation

In this section, we discuss how we adapt our methodology for the PRAM model of computation and implement it on GPU. The entire algorithm is implemented using the NVIDIA Thrust library (included with the CUDA toolkit), which can execute the program on NVIDIA GPUs or CPUs parallelized using OpenMP or Intel's Thread Building Blocks (TBB) with minimal change.

4.1 Overview

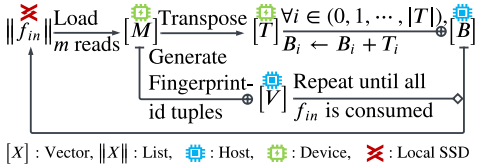


Fig. 1: Generating fingerprint-id tuples

Figure 1 depicts the first stage of building the index - generating fingerprints of reads. The input file f_{in} is processed in chunks of m reads such that the intermediate data fits in device memory. Next, for each read in chunk $\mathcal{M}$, we generate fingerprints, create fingerprint-id tuples, and append them to host vector $\mathcal{V}$. We also generate the fingerprints of reverse complement reads and append the corresponding tuples to $\mathcal{V}$ (not shown in the figure for brevity). The original read s_j is assigned the id $2j$ and its reverse complement $2j + 1$.

Note that getting the fingerprints of suffixes and prefixes of length $k-1$ from those of length k only requires the bases at two positions - $k-1$ for prefixes and $l-k$ for suffixes. Thus, if $\mathcal{R}$ is stored in the host in a column-major order, only the bases used to generate fingerprints can be bulk-transferred to device and removed after the fingerprints are generated. Therefore, the reads are transposed and appended to host vector $\mathcal{B}$ so that $\mathcal{B}_{ij}$ is the i^{th} base of read j. Fingerprints are generated using Thrust's `scan_by_key` function, and the transpose and reverse complements are generated using Thrust's `scatter` function.

Figure 2 illustrates the second stage - regenerating suffix/prefix fingerprints and using them to find overlaps. It takes the vector $\mathcal{V}$ of fingerprint-id pairs generated in the previous stage, sorts the pairs, and removes those with identical fingerprints. The sorting by read-ids ensures that for any read that's removed, its reverse complement is removed as well. The main challenge here is that $\mathcal{V}$ does

not fit in GPU memory, so we partition it in main memory and stream partitions to device memory, where they can be processed independently (discussed in Algorithms 1 and 2). The partitioning involves only an $\mathcal{O}(\log r)$ number of random accesses in the host memory. Most random access operations are offloaded to GPU.

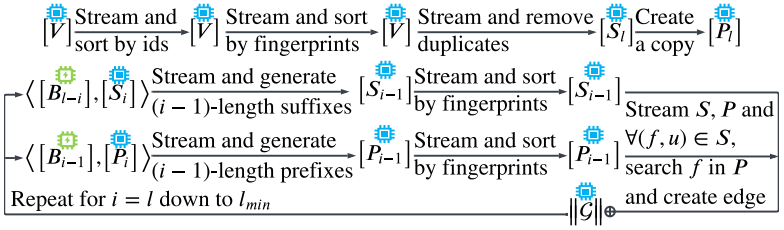


Fig. 2: Finding overlaps

One consequence of partitioning is that both $\mathcal{P}$ and $\mathcal{S}$ must now store tuples of fingerprints and read-ids, instead of the method described in Section 3, where $\mathcal{S}$ only consisted of fingerprints. The fingerprint generation can be done in bulk if both $\mathcal{P}$ and $\mathcal{S}$ are sorted by read ids since the bases in $\mathcal{B}$ are already arranged according to their read ids. After the fingerprints are regenerated as per Eq 2 and Eq 4, both the vectors of suffix- and prefix-pairs are sorted by their fingerprints.

The sorted vectors of suffix and prefix pairs are partitioned again, and the partitions are streamed to GPU to search for overlaps. We use Thrust's vectorized binary search functions to search for each suffix fingerprint in the list of prefix fingerprints. Specifically for each suffix fingerprint f_i, we first search for the lower bound and then for the upper bound of f_i in the array of prefix fingerprints. The difference between the upper and lower bounds gives the number of times f_i occurs in the array of prefix fingerprints. We then filter out the read-ids whose corresponding fingerprints have non-zero occurrences in both vectors, convert them to an edge-list format and write to the output file.

4.2 Sorting and searching in hybrid-memory

The sorting of vectors and searching for keys are performed in hybrid-memory, i.e., both the host and device memories. As stated before, most work is done on the GPU, and the CPU is responsible for partitioning the vectors so that the GPU can process each partition independently. Furthermore, since the vectors can be enormous, we want to perform these operations using a constant amount of auxiliary space in the host memory.

To achieve this, we use a non-contiguous host vector where data is stored in blocks. This lets us allocate and free chunks of data whenever necessary so that the peak memory usage is kept in check. The host-vector consists of a queue of smaller arrays and supports appending new data at the end and reading (and popping) data from the front. When writing to vectors, the data is appended to the last block in the queue. When the last block is full, a new one is allocated and

pushed into the queue. Data is read from the first block in the queue, and the number of items read is tracked. Once a block is consumed, it's popped from the front of the queue, and the memory is freed. In the actual implementation, we use a memory pool of blocks where freed blocks are stored and reused so that the repeated allocations and deallocations do not negatively impact performance.

In addition to sequential reads and writes, these vectors also support random accesses with the `[]` operator. This is useful in partitioning sorted vectors for streaming them into device memory, particularly for the `lower_bound` function. The lower bound of a key in a sorted vector is the smallest index at which the key can be inserted without invalidating the sorted order.

```
1  function get_merge_partition_sizes (A, B, m)
2      p ← list<pair<int, int>>, o_A ← 0, o_B ← 0
3      while o_A < A.n and o_B < B.n
4          n_A ← min(A.n − o_A, m/2), n_B ← min(B.n − o_B, m/2)
5          if n_A < m/2 and n_B < m/2 then:
6              p.push_back((n_A, n_B))
7              break;
8          else:
9              k ← min(A[o_A + n_A − 1], B[o_B + n_B − 1])
10             n_A ← A.lower_bound(o_A, o_A + n_A, k) − o_A
11             n_B ← B.lower_bound(o_B, o_B + n_B, k) − o_B
12             p.push_back((n_A, n_B))
13             o_A ← o_A + n_A, o_B ← o_B + n_B
14     if o_A = A.n or o_B = B.n then:
15         if o_A = A.n and o_B < B.n then p.push_back((0, nB − o_B))
16         else p.push_back((nA − o_A, 0))
17     return p
```

Algorithm 1: Partitioning a non-contiguous host vector

The pseudocode for partitioning a pair of sorted non-contiguous host vectors is shown in Algorithm 1. The partitioning function accepts an argument m which specifies the maximum size of each partition. If then slides an $(m/2)$-sized window over each vector and resizes the windows based on the minimum of the last elements in both (lines 9-11). The partitions resulting from this algorithm splits the key-ranges of both the vectors in a way such that if a key k is present in partition j of one of the vectors A, then it will not be present in any other partition in A and if k is present in vector B, then it will only be present in partition j of B.

The vectors are sorted in device memories as shown in algorithm 2. The vectors are loaded on the GPU in chunks of m and sorted using Thrust's `sort_by_key` function. These sorted chunks are then iteratively merged to obtain a single sorted vector. The merge algorithm uses the partition function in Algorithm 1 to split the vector of keys into independent partitions, loads these partitions into GPU and merges them by keys (fingerprints). In practice, the merging is also performed using radix sort. During the sorting and merging phases, the data

blocks popped from the vectors are freed, thereby keeping the auxiliary memory usage limited to $\mathcal{O}(b)$, where b is the block size of the vector.

```
1  function sort_by_key (K, V)
2      s ← queue<pair<vector, vector>>
3      d_K and d_V are m-length arrays of keys and vals on the device
4      repeat until K is empty
5          s.emplace_back()
6          load upto m items from K and V into d_K and d_V
7          gpu_sort_by_key (d_K, d_V)
8          s.back().first.push_back (d_K, length(d_K))
9          s.back().second.push_back (d_V, length(d_V))
10     while length (s) > 1
11         K_a, V_a ← s.pop_front()
12         K_b, V_b ← s.pop_front()
13         K_c, V_c ← merge_by_key (K_a, V_a, K_b, V_b)
14         s.push_back ((K_c, V_c))
15     K ← s.back().first, V ← s.back().second
16
17 function merge_by_key (K_a, V_a, K_b, V_b)
18     K_c ← vector, V_c ← vector
19     d_K and d_V are m-length arrays of keys and vals on the device
20     P ← get_merge_partition_sizes (K_a, K_b)
21     ∀p ∈ P:
22         n_A ← p.first, n_B ← p.second
23         if n_A and n_B
24             k_A→pop_front (d_K, n_A)
25             k_B→pop_front (d_K + n_A, n_B)
26             v_A→pop_front (d_V, n_A)
27             v_B→pop_front (d_V + n_A, n_B)
28             gpu_sort_by_key (d_K, d_V)
29             k_C→push_back (d_K, n_A + n_B)
30             v_C→push_back (d_V, n_A + n_B)
31         else if n_A
32             pop n_A items from K_a and V_a and push into K_c and V_c
33         else
34             pop n_A items from K_a and V_a and push into K_c and V_c
35     return K_c, V_c
```

Algorithm 2: Sorting in hybrid memory

5 Results

5.1 Datasets and Testbed

We evaluate our implementation (Empanada) on several real-world datasets whose Sequence Read Experiment (SRX) IDs are given in Table 1. All datasets were generated using Illumina genome sequencing machines and were obtained

from the NCBI Sequence Read Archive[2]. They are chosen so that they have similar (or the same) read lengths, and the number of bases roughly doubles in size from one dataset to the next.

Table 1: Datasets used.

SRX ID	Name	l	$\|R\|$	n
10829778	Fruitfly	152	83.6M	12.7 GB
14756245	Salamander	151	165M	24.9 GB
10572708	Butterfly	151	343.7M	51.9 GB
10301361	Starling	151	723.8M	109.3 GB
1382207{4,6}	Pig	151	1355.4M	204.7 GB

$\boldsymbol{l}$: Read-length, $\boldsymbol{|R|}$: Number of reads, not including reverse complements, $\boldsymbol{n} = l \times |R|$: number of bases

All experiments were performed on the Expanse cluster in San Diego Supercomputing Centre[3]. Each GPU node has 384 GB RAM, two 20-core 2.5 GHz processors, four NVIDIA V100, each with 12 GB device memory, and a 1.6TB NVME SSD. Empanada uses a single CPU thread and a single GPU in a shared-node setting. The maximum memory available for Empanada was limited to 90 GB for the first 4 datasets (Fruitfly, Salamander, Building, Starling) and 180 GB for the Pig dataset. Each Compute node has two 64-core 2.5 GHz processors, 256 GB RAM and a 1TB NVME SSD. Compute nodes were used to run SGA and Readjoiner in exclusive mode, with access to all cores and the full available memory. Both types of nodes have access to 12PB Lustre filesystem, but we used SSDs for I/O.

5.2 Execution times

The total execution times of Empanada are reported in Table 2. We use a minimum overlap length of 55% of the read length, which is at the lower end of what is used in most string-graph-based assemblers. As previously mentioned, creating a string graph requires the sparsification of an overlap graph which can be done by removing transitive edges or retaining only the longest overlaps per read. We report the execution times with and without the greedy longest-edge retention approach. The time elapsed for fingerprint generation (t_f) includes that of reading input data. $t_o^{(g)}$ and $t_o^{(f)}$ denote the time required to obtain overlaps with and without the greedy edge retention, respectively. They include the time taken to remove duplicate reads and write the edges on the disk.

For the largest dataset with 2.3 billion unique reads (r_u), each 151 bases long, Empanada finishes in under 4.5 hours ($t^{(f)}$). When it retains only the longest overlaps for each read, Empanada takes under 2.5 hours ($t^{(g)}$). As expected, when only the heaviest edges are retained, the number of edges $E^{(g)}$ grows linearly with r_u, whereas when no edges are discarded, the number of edges $E^{(f)}$ explodes.

We define the collision rate for strings of length j as (1 - the number of unique fingerprints generated $\div$ the total number of unique suffixes/prefixes of length j). The total collision rate ($n_c^{(g)}$) is defined as the sum of collision rates

[2] https://www.ncbi.nlm.nih.gov/sra/
[3] https://www.sdsc.edu/support/user_guides/expanse.html

Table 2: Reads, Edges and Execution times

Data	r	t_f	r_u	$t_o^{(g)}$	$t^{(g)}$	$E^{(g)}$	$t_o^{(f)}$	$t^{(f)}$	$E^{(f)}$
FFY	164M	33s	107M	2m26s	2m59s	83M	5m7s	5m40s	104.75B
SMR	329M	1m39s	261M	10m41s	12m20s	73M	13m15s	14m54s	4.55B
BFY	687M	3m51s	504M	5m34s	9m25s	417M	28m9s	32m0s	26.72B
SLG	1.45B	9m14s	1.37B	1h7m59s	1h17m13s	618M	1h54m50s	2h4m4s	1.67T
PIG	2.71B	17m37s	2.34B	1h56m42s	2h14m19s	1.4B	3h55m58s	4h13m35s	127.79B

$\boldsymbol{r}$: number of error-free reads and reverse complements, $\boldsymbol{t_f}$: time taken to generate fingerprints, $\boldsymbol{r_u}$: number of unique reads (and reverse complements), $\boldsymbol{t_o^{(g)}}$: time taken to find overlaps with greedy edges, $\boldsymbol{t^{(g)}}$: total time with greedy edges, $\boldsymbol{E^{(g)}}$: number of greedy edges created, $\boldsymbol{t_o^{(f)}}$: time taken to find overlaps without any edge-removal, $\boldsymbol{t^{(f)}}$: total time without any edge-removal, $\boldsymbol{E^{(f)}}$: number of edges created.
M = Million, B = Billion, T = Trillion

of all suffixes/prefixes from l_{min} to l, when edges are selected greedily. In all our experiments, the fingerprints were generated using two 29-bit pseudo-Mersenne primes and combined into one 58-bit integer (stored as a 64-bit integer) to avoid integer overflows during multiplications. In this setup, $n_c^{(g)}$ was found to be 1×10^{-8} for Butterfly, 2×10^{-8} for Starling, and 5×10^{-8} for the others. Note that the other tools do not have the issue of false positives, but we claim that the false positive rate and be made arbitrarily close to 0 by randomly matching characters of the *supposedly* overlapping reads.

5.3 Scalability

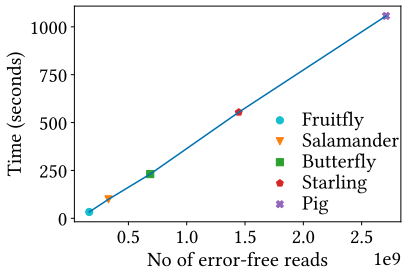


Fig. 3: Fingerprint generation time vs number of reads

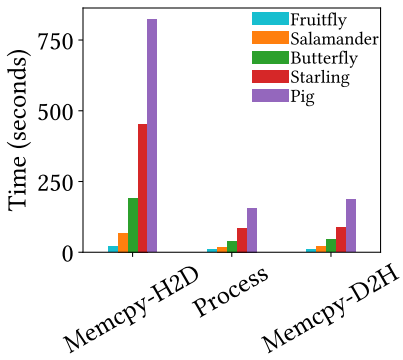
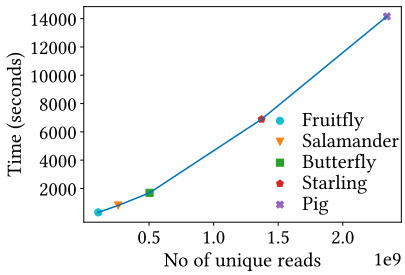


Fig. 4: Time spent of each stage during fingerprint generation

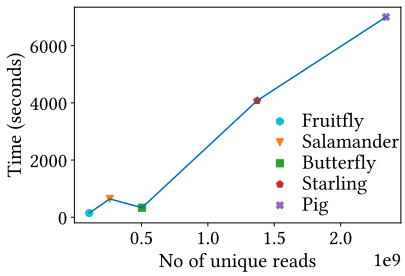
Figure 3 shows the time taken to generate fingerprints w.r.t. the number of reads. It is evident from the graph that this phase is highly-scalable across a wide range of data sizes. This is because this phase

consists of finding reverse complements and transposing bases, both of which are based on the scatter pattern, which is embarrassingly parallel. The fingerprint generation is based on parallel prefix-scan, which has a linearithmic time complexity.

Figure 4 presents the time taken by the different parts of the fingerprint generation phase. The 'process' stage in the figure includes all the computations mentioned before. Most of the execution time is spent reading data from disk to device and copying fingerprint-read tuples and transposed bases from device to host. Both of these stages depend only on the data size and hence scale almost perfectly. This figure demonstrates that the major bottleneck in this phase is I/O. The performance of the data-copying could be improved with GPU-Direct which bypasses host memory when transferring data from disk to GPU. We plan to explore this in future work.



(a) With all edges



(b) With greedy edges

Fig. 5: Overlap detection time vs number of unique reads

Figure 5 plots the time taken to remove duplicates and find overlaps of up to 55% of the read lengths vs the number of unique reads. Figure 5 (a) considers the execution times when all edges are retained. In this scenario, the time spent on overlap detection grows almost linearly with the number of unique reads, with a slight upward trend starting from the Butterfly dataset. This is because, for smaller datasets, the fingerprint-id pairs can be loaded into the GPU and sorted at once, whereas for Starling and Pig, sorting the lists requires multiple passes over the data.

In Figure 5 (b), the execution times are reported for the cases when only the heaviest edges per vertex were retained. In this case, we observe that the overlap detection time does not always grow predictably with an increase in the number of reads. For instance, the Butterfly dataset finishes quicker than Salamander - a dataset half its size.

To investigate the reason behind this, we study the progression of the overlap detection phase for the three smallest datasets in Figure 6. For the three subfigures in Fig. 6, the x-axis depicts the overlap round i, during which the program obtains matching suffix-prefix pairs of length $(i+1)$. Figure 6 (a) shows the number of edges created at each overlap round. We can observe that for the butterfly dataset, about 100 million edges are created during the first overlap round, which is much larger than the two smaller datasets. The vertices from which these edges emerge are removed from

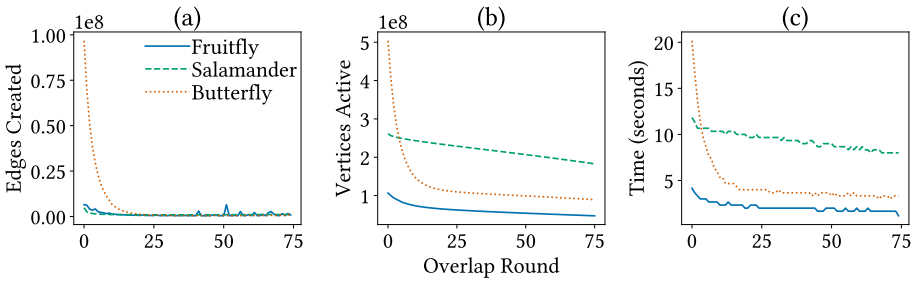


Fig. 6: Edges created vs overlap detection time

the list before the start of the next overlap round. This effect can be seen in Figure 6 (b), which plots the number of active vertices per round. Due to the large number of edges discovered in the first few rounds, the number of active vertices decreases sharply, reducing the search space considerably. Consequently, the search time also decreases within the first few rounds. Eventually, searching in the Butterfly dataset finishes faster than in the smaller Salamander dataset.

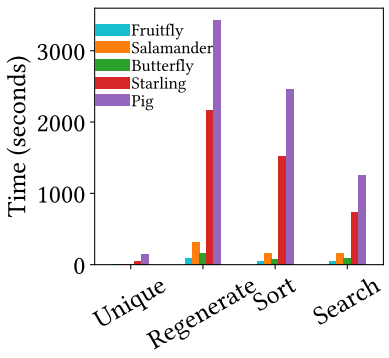


Fig. 7: Time spent of each stage during overlap detection

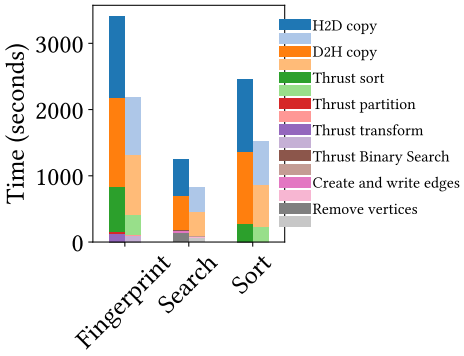


Fig. 8: Time spent for computation and data-copy for Starling and Pig

In Figure 7, we examine the individual stages of the edge-building phase across all overlap lengths and study their contributions towards the execution time. The plot shows that most of the time is spent on regenerating fingerprints and sorting them. These two phases display a quasilinear trend consistent with their respective algorithms having multiple merge passes. Note that regenerating fingerprints also involves sorting, which takes up a considerable portion of its time (discussed later). The search phase also significantly contributes to the execution time but grows more slowly (almost linearly) with increasing data sizes. Again, this is consistent with the algorithm, which comprises creating

vertex pairs and writing them to disk. The stage 'unique' is a one-time removal of duplicates before the start of overlap detection and takes very little time.

In Figure 8, we inspect the components of the three most time-consuming stages of overlap detection - fingerprint regeneration, sorting, and searching in light of their share of execution times for Pig (deeply shaded bars) and Starling (lightly shared bars). To begin with, it is readily apparent that host-to-device (H2D) and device-to-host (D2H) data transfers take up most of the time across all three stages for both datasets. As indicated before, regenerating fingerprints in a batch requires sorting them by their read-ids, which for large datasets requires multiple passes. Indeed, sorting and the associated data transfers are the most time-consuming steps in this stage, as seen in the figure. The actual computation of fingerprints which involves Thrust's partition and transform methods, is almost insignificant in comparison. A similar trend can be observed in the sorting and search stages, where memory transfers dominate the actual computation. We are exploring avenues for alleviating this bottleneck from an algorithmic perspective.

5.4 Comparison of execution times with other tools

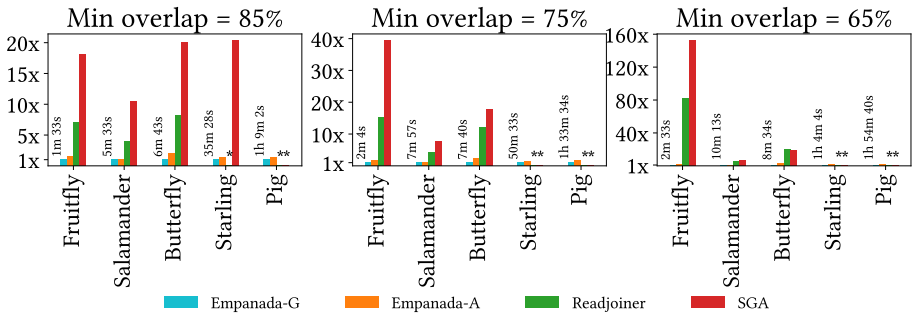


Fig. 9: Comparison of execution times of Empanada with Readjoiner and SGA

Figure 9 compares the performance of Empanada with Readjoiner and SGA, the only string graph assemblers that can handle large short-read datasets. We only use the results of preprocess, index, filter and overlap stages of SGA and the prefilter and overlap phases of Readjoiner. For each dataset, we report the results of 3 experiments where we set the minimum overlaps to 85%, 75%, and 65% of the read lengths. For Empanada, we study the performance both with and without transitive edge removal. Note that Readjoiner could not process the Starling dataset due to some unknown error and the Pig dataset due to insufficient disk space. SGA could not process Pig due to some unknown error and could process Starling within the maximum allowed time (72 hours) only when the minimum overlap was set to 85%. It can be seen that even when Readjoiner is run on 128

cores, Empanada with greedy edge retention (Empanada-G) is at least 4 times faster (in case of Salamander, 85%) and has a median speedup of about 8x. Compared to SGA, Empanada-G has a speedup of 7x at minimum with a median of 18.5x. All of these results are not only significantly better than SGA, but also outperforms our previous tool (LaSAGNA) by a considerable margin. Although we couldn't test LaSAGNA on the Expanse cluster on the large datasets due to a lack of local disk space, for the Fruitfly and Salamander datasets, Empanada finished about thrice as fast as the single-node implementation of LaSAGNA.

6 Conclusions

In this work, we introduce a new memory-efficient, parallel and highly scalable suffix-prefix indexing method using Rabin fingerprints that can find a trillion overlaps in a billion strings in a couple of hours on a single node using a single GPU. Our approach uses a non-contiguous vector to perform an in-place parallel merge sort of data in a two-level memory model, thereby allowing it to process data much larger than GPU memory. Experimental results demonstrate that our implementation outperforms existing approaches by a significant margin and exhibits almost linear scalability on a wide range of data sizes. In the future, we plan to implement a multi-gpu version, and also study the possibility of a heterogeneous cpu-gpu implementation. As an extension to this work, we will also perform more detailed analyses on the assembly accuracy.

Acknowledgements This work used Expanse CPU and GPU nodes and Expanse Projects Storage at San Diego Supercomputing Centre through allocation CIS220052 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296.

References

1. Büren, F., Jünger, D., Kobus, R., Hundt, C., Schmidt, B.: Suffix array construction on multi-gpu systems. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing. pp. 183–194 (2019)
2. Dinh, H., Rajasekaran, S.: A memory-efficient data structure representing exact-match overlap graphs with application for next-generation dna assembly. Bioinformatics **27**(14), 1901–1907 (2011)
3. Dong, T., Dobrev, V., Kolev, T., Rieben, R., Tomov, S., Dongarra, J.: A step towards energy efficient computing: Redesigning a hydrodynamic application on cpu-gpu. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium. pp. 972–981. IEEE (2014)
4. Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S.: Gpu cluster for high performance computing. In: SC'04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing. pp. 47–47. IEEE (2004)
5. Gonnella, G., Kurtz, S.: Readjoiner: a fast and memory efficient string graph-based sequence assembler. BMC bioinformatics **13**(1), 1–19 (2012)

6. Goswami, S., Lee, K., Park, S.J.: Distributed de novo assembler for large-scale long-read datasets. In: 2020 IEEE International Conference on Big Data (Big Data). pp. 1166–1175. IEEE (2020)
7. Goswami, S., Lee, K., Shams, S., Park, S.J.: Gpu-accelerated large-scale genome assembly. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 814–824. IEEE (2018)
8. Gusfield, D., Landau, G.M., Schieber, B.: An efficient algorithm for the all pairs suffix-prefix problem. Information Processing Letters **41**(4), 181–185 (1992)
9. Haj Rachid, M., Malluhi, Q.: A practical and scalable tool to find overlaps between sequences. BioMed research international **2015** (2015)
10. Hernandez, D., François, P., Farinelli, L., Østerås, M., Schrenzel, J.: De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. Genome research **18**(5), 802–809 (2008)
11. Jain, G., Rathore, L., Paul, K.: Gams: Genome assembly on multi-gpu using string graph. In: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS). pp. 348–355. IEEE (2016)
12. Jünger, D., Kobus, R., Müller, A., Hundt, C., Xu, K., Liu, W., Schmidt, B.: General-purpose gpu hashing data structures and their application in accelerated genomics. Journal of Parallel and Distributed Computing **163**, 256–268 (2022)
13. Klus, P., Lam, S., Lyberg, D., Cheung, M.S., Pullan, G., McFarlane, I., Yeo, G.S., Lam, B.Y.: Barracuda-a fast short read sequence aligner using graphics processing units. BMC research notes **5**(1), 1–7 (2012)
14. Liu, C.M., Wong, T., Wu, E., Luo, R., Yiu, S.M., Li, Y., Wang, B., Yu, C., Chu, X., Zhao, K., et al.: Soap3: ultra-fast gpu-based parallel alignment tool for short reads. Bioinformatics **28**(6), 878–879 (2012)
15. Liu, Y., Schmidt, B., Maskell, D.L.: Cushaw: a cuda compatible short read aligner to large genomes based on the burrows–wheeler transform. Bioinformatics **28**(14), 1830–1837 (2012)
16. Simpson, J.T., Durbin, R.: Efficient construction of an assembly string graph using the fm-index. Bioinformatics **26**(12), i367–i373 (2010)