Rapid GPU-Based Pangenome Graph Layout

Jiajie Li
Cornell University
Ithaca, NY, USA
jl4257@cornell.edu

Jan-Niklas Schmelzle
Technical University of Munich
Munich, Germany
ge75qew@tum.de

Yixiao Du
Cornell University
Ithaca, NY, USA
yd383@cornell.edu

Simon Heumos
University of Tübingen
Tübingen, Germany
simon.heumos@qbic.uni-tuebingen.de

Andrea Guarracino
UTHSC
Memphis, TN, USA
aguarra1@uthsc.edu

Giulia Guidi
Cornell University
Ithaca, NY, USA
gg434@cornell.edu

Pjotr Prins
UTHSC
Memphis, TN, USA
jprins@uthsc.edu

Erik Garrison
UTHSC
Memphis, TN, USA
egarris5@uthsc.edu

Zhiru Zhang
Cornell University
Ithaca, NY, USA
zhiruz@cornell.edu

***Abstract*—Computational Pangenomics is an emerging field that studies genetic variation using a graph structure encompassing multiple genomes. Visualizing pangenome graphs is vital for understanding genome diversity. Yet, handling large graphs can be challenging due to the high computational demands of the graph layout process.**

In this work, we conduct a thorough performance characterization of a state-of-the-art pangenome graph layout algorithm, revealing significant data-level parallelism, which makes GPUs a promising option for compute acceleration. However, irregular data access and the algorithm's memory-bound nature present significant hurdles. To overcome these challenges, we develop a solution implementing three key optimizations: a cache-friendly data layout, coalesced random states, and warp merging. Additionally, we propose a quantitative metric for scalable evaluation of pangenome layout quality.

Evaluated on 24 human whole-chromosome pangenomes, our GPU-based solution achieves a 57.3x speedup over the state-of-the-art multithreaded CPU baseline without layout quality loss, reducing execution time from hours to minutes.

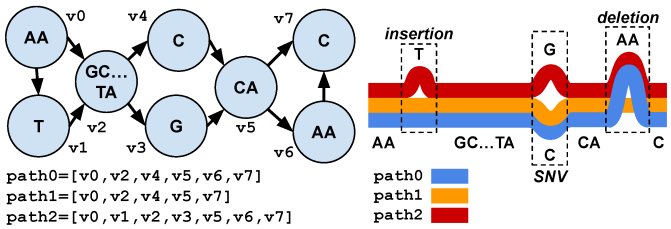
***Index Terms*—Pangenomics, Bioinformatics, Graph layout, GPU acceleration**

I. Introduction

Low-cost genome sequencing [1], [2] has made it possible to collect extensive genetic data for specific species, providing opportunities for deeper exploration. Pangenomics [3] is an emerging field of genomics that aims to understand the complete picture of the genetic variation of a species by studying multiple genomes [4], [5]. Graphical pangenomics models a pangenome as a graph. This graph-based approach complements traditional reference-based genomics by revealing overlooked genetic variation when a single reference genome is used [6]. In particular, the recent release of the first draft of the human pangenome reference [7] represents a major milestone. This achievement represents a significant advance in human genetics, echoing the first release of the human genome sequence in 2001 [8].

Pangenomes [9] can model the entire genomic variation of a given population [10]. A variant refers to the differences between different genome sequences and can provide biological insights, such as disease susceptibility [11], [12], genome functionality identification [13], and evolutionary studies [14].

A prevalent pangenomic model to represent these differences is the variation graph [15]. As illustrated in Fig. 1, the variation graph captures both genomic sequences and variations amongst them. The data structure of a variation graph, formed by merging identical segments from multiple genomes into a single node, is depicted in Fig. 1a. Its visualization, as seen in Fig. 1b, reveals variants including insertions, deletions, and single nucleotide variants (SNVs). In general, visualization is an effective way to reveal structural differences between genomes and gain insights [3].



(a) Variation graph data structure. (b) Visualization.

Fig. 1: **A variation graph and its visualization example** — the three genomes are depicted in different colors; the path of interconnected nodes represents the original genome.

The visualization of a pangenome combines layout and rendering, with the layout component being fundamental to the graph visualization quality. In particular, the layout of a variation graph is crucial to variant discovery in large pangenomes. Fig. 2 shows the layout of the HLA-DRB1 gene [16], which encodes an immune system protein associated with reduced severity of COVID-19 disease [17]. Genome researchers can easily identify the location and structure of variants with an optimized, planar 2D layout of the variation graph, aiding in the study of pangenomes.

However, general graph layout frameworks are not well-suited to effectively lay out pangenome graphs. This limitation primarily arises from the unique biological significance associated with the nodes and paths within a pangenome. Currently, only specialized tools [18]–[22] for pangenome graphs offer effective support. Yet, pangenome graph layout, especially for

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

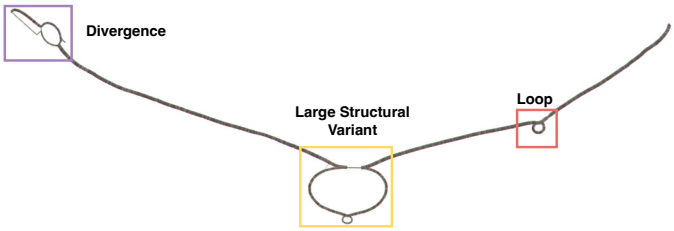


Fig. 2: **Layout of the HLA-DRB1 gene** — three distinct variant types are shown in the bounding boxes.

large human pangenome graphs, remains an extremely time-consuming process. The current state-of-the-art approach *odgi-layout* [20] requires hour-scale time to generate the layout of the variation graph for a single human chromosome with a 32-core server-class Intel Xeon CPU. In addition, the layout process often requires multiple rounds of parameter tuning to achieve an optimal layout, resulting in a bottleneck in the pangenome analysis pipeline.

This work aims to accelerate the computation of pangenome graph layouts, a crucial step in pangenomics. We show that the pangenome layout algorithm exhibits a substantial degree of data-level parallelism, albeit underutilized in the current state-of-the-art CPU implementation [20], impeding progress in pangenome research. With significant data parallelism available, GPU acceleration holds promise for this application. However, challenges arise due to the irregular data access pattern and memory-bound nature of the algorithm.

In this paper, we present a novel solution to pangenome graph layout computation, by leveraging the computational power of modern GPUs and optimizing the data access pattern. Our approach not only accelerates layout computation but also improves the overall efficiency and scalability of pangenomics analyses. Our main contributions are as follows:

- To our knowledge, we present the first GPU-based solution to accelerate pangenome graph layout, which enables minute-scale layout for the entire chromosome dataset. Our implementation achieves an average speedup of 57.3× compared to an optimized, state-of-the-art CPU implementation. We will open-source our software in a format that facilitates easy integration into the pangenomic analysis pipeline.
- To identify the performance bottleneck, we perform a detailed workload characterization of the pangenome graph layout algorithm. Our analyses indicate that this workload has a highly irregular data access pattern and is memory-bound. Thus, a naïve approach is inadequate for fully exploiting GPU's computational capabilities.
- We introduce three key optimizations to improve GPU performance: (1) optimizing the data layout for improved cache efficiency, (2) enabling coalesced memory accesses by coalescing random states, and (3) reducing warp divergence through warp merging.
- We propose a quantitative metric called sampled path stress to assess the quality of GPU-generated layouts in a scalable manner. Through a case study, we demonstrate the potential to explore performance-quality trade-offs using this metric, leading to additional speedup.

II. Background

This section introduces the background of pangenomics, its variation graph representation, and its graph layout algorithm.

A. Variation Graph

Graph-based pangenomics aims to study genome variation within a population of samples. The variation graph serves as the primary model to describe graph-based pangenomes.

A variation graph $G = (P, V, E)$ is a directed graph composed of a set of *paths* P, *nodes* V and *edges* E, as shown in Fig. 1a. Each *node* represents a nucleotide sequence, each *edge* represents the connection of an ordered pair of nodes, and each *path* describes a walk over nodes.

The path consists of interconnected nodes and represents the original genome, e.g., path 2 in Fig. 1a embodies a genome sequence of `AATGC...TAGCAAAC`. While most nodes are shared across all paths, variants exist in the form of unique nodes. These variants are revealed by visualizing the variation graph, as shown in Fig. 1b. For instance, the `T` insertion in path 2 serves as a variant and is the primary discovery focus.

Variation graphs representing biological sequences typically exhibit a linear structure, as opposed to the more commonly encountered planar graphs. This characteristic stems from the linear nature of the genome sequences they represent, where the majority of segments are identical due to sequence homology. Consequently, variation graphs display a notably low average node degree and density. As an example, the average node degree of human pangenome graphs released by the HPRC [7] is 1.4, and the average density is 3.5×10^{-7}. These graph properties, along with the genome-specific path information, make variation graphs particularly unique, opening opportunities for ad-hoc algorithmic optimizations.

B. Pangenome Graph Layout

The aim of a pangenome graph layout is to organize nodes and edges in order to highlight the genetic variation present in the genomes represented in the graph. This enables the large-scale study of the diversity and evolution embodied in tens or hundreds of genomes. For example, the layout structure of a pangenome graph representing the 5 acrocentric human chromosomes of the HPRC pangenome revealed heterologous recombination in the human pangenome [13].

Existing general graph layout frameworks [23], [24] struggle to reveal the structural variants of pangenome graphs. We illustrate this by using Gephi [23] to lay out the HLA-DRB1 gene with algorithms including Fruchterman-Reingold [25], ForceAtlas2 [26] and Yifan Hu [27]. These algorithms, while creating 2D structures, fail to uncover the underlying structural variants. This is due to their design for calculating distances between all nodes, whereas pangenome graphs only consider nodes on the same path meaningful.

Given that both the biological meanings of nodes and paths must be factored into the layout process, only specialized

tools for pangenome graphs prove effective. Among these, the current state-of-the-art approach is *odgi-layout*, which is part of the comprehensive pangenome analysis framework ODGI [28]. By adapting Zheng et al.'s work [24] to the pangenomic field, *odgi-layout* utilizes a path-guided stochastic gradient descent (Path-Guided SGD) algorithm to minimize stress, a proxy metric quantifying the difference between reference and layout distances. With its multi-threaded CPU implementation, *odgi-layout* stands as the only tool capable of handling whole-chromosome graphs with millions of nodes.

However, more efficient graph layout solutions are needed to rapidly compute layouts of increasingly large and complex pangenomes. Indeed, *odgi-layout* demands hours on a 32-core Intel Xeon CPU to generate a pangenome graph layout for just one human chromosome. Specifically, computing the layout of the chromosome 1 (Chr.1) pangenome — the largest chromosomal pangenome released by HPRC — alone exceeds 2.5 hours. Completing the layouts for all 24 chromosomal pangenome graphs from HPRC sums up to a significant 28 hours. Notably, running the layout computation once takes up nearly a third of the entire pangenomics analysis pipeline [29]'s duration. Given that multiple runs are often performed for optimization, the layout computation becomes an even more pronounced bottleneck.

This performance issue impedes the study of large and/or complex pangenome graphs because of the prolonged layout generation times. Importantly, a fast layout solution would facilitate interactive visualization, allowing on-the-fly exploration of specific loci, genomic regions, entire chromosomes, or even whole genomes. This would further pave the way for the development of next-generation pangenome browsers, unlocking the study of population-scale genetic variability. This motivates us to pursue substantial acceleration in pangenome graph layout generation.

C. Path-Guided SGD Algorithm

Alg. 1 presents the pseudocode for the path-guided SGD algorithm used in *odgi-layout*. This algorithm iteratively selects one pair of *nodes* (n_i, n_j) from the same path p (lines 7-11). For each of these nodes, represented by a line segment in the layout, a *visualization point* is selected (lines 12, 13). This yields a pair of *visualization points* $(\mathbf{v_i}, \mathbf{v_j})$, each corresponding to an endpoint of the respective node's line segment. This pair of *visualization points* forms a loss function (known as stress) with its reference distance d_{ref} and current layout distance $||\mathbf{v_i} - \mathbf{v_j}||$ (line 14). Then the coordinates are updated based on the gradient (line 15).

The update process is illustrated in Fig. 3, where both nodes are moved against the direction of the gradient [24].

III. Workload Characterization

In this section, we describe the workload characterization of the multi-threaded CPU implementation of *odgi-layout* on a 32-core Intel Xeon Gold 6246R 3.4GHz CPU. For detailed profiling, we use Linux Perf [30] and Intel VTune profiler [31].

Algorithm 1 Path-Guided Pangenome Graph Layout

Input: pangenome graph $G = (P, V, E)$, SGD schedule S, total iteration count N_{iters}
Output: a 2D layout $\mathbf{L}$ consisting of line segments. $\mathbf{L}[n]$ returns an array of 2 vectors pointing to its endpoints given $n \in V$.

1: $N_{steps} \leftarrow 10 \times \sum_{p \in P} |p|$ ▷ $|p|$: # of nodes in path p
2: **for** $iter \leftarrow 0$ to N_{iters} **do**
3: $\eta \leftarrow S[iter]$ ▷ learning rate
4: **for** $step \leftarrow 0$ to N_{steps} **do in parallel**
5: $p \leftarrow$ RandomSelect(P, $prob \propto |p|$)
6: $cooling \leftarrow (iter \geq N_{iters}/2)$ or FlipCoin()
7: **if** $cooling$ **then**
8: $n_i, n_j \leftarrow$ RandomSelect(p, Powerlaw)
9: **else**
10: $n_i, n_j \leftarrow$ RandomSelect(p, Uniform)
11: **end if**
12: $\mathbf{v_i} \leftarrow$ FlipCoin() ? $\mathbf{L}[n_i]$.start : $\mathbf{L}[n_i]$.end
13: $\mathbf{v_j} \leftarrow$ FlipCoin() ? $\mathbf{L}[n_j]$.start : $\mathbf{L}[n_j]$.end
14: $stress_{ij} \leftarrow ((||\mathbf{v_i} - \mathbf{v_j}|| - d_{ref})/d_{ref})^2$ ▷ loss
15: $(\mathbf{v_i}, \mathbf{v_j}) \leftarrow (\mathbf{v_i}, \mathbf{v_j}) - \eta \nabla stress_{ij}$ ▷ update
16: **end for**
17: **end for**

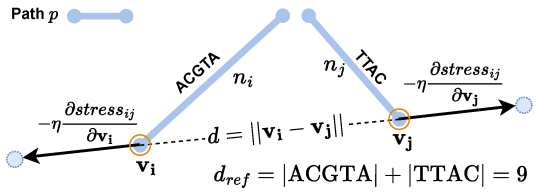


Fig. 3: **Layout update within one step** — n_i and n_j are two nodes representing the nucleotide sequences "ACGTA" and "TTAC", respectively.

We evaluate the layout computation on three representative pangenomes of varying sizes, as detailed in Table I.

TABLE I: **Properties of representative pangenomes** — # Nuc. is the number of nucleotides.

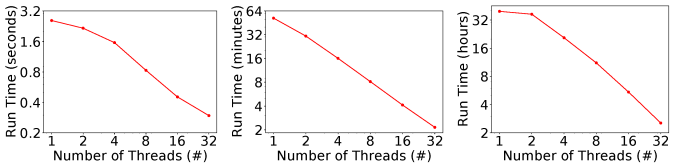
Pangenome	# Nuc.	# Nodes	# Edges	# Paths
HLA-DRB1	2.2×10^4	5.0×10^3	6.8×10^3	12
MHC	5.9×10^6	2.3×10^5	3.2×10^5	99
Chr.1	1.1×10^9	1.1×10^7	1.5×10^7	2,262

Our analysis highlights three key observations: (1) the algorithm exhibits high data-level parallelism; (2) it is highly memory-bound; (3) randomness is critical to the layout quality. In the following, we delve deeper into each of them.

A. Data-level Parallelism

The multi-threaded CPU implementation of *odgi-layout* runs the inner loop in parallel and updates the layout asynchronously in a Hogwild! [32] manner. This means that the

`for` loop at line 4 in Alg. 1 has high data-level parallelism for a graph with a large number of nodes. While the intrinsic race condition between parallel threads could introduce errors, the layout quality is barely affected since pangenome graphs are so sparse that the probability of multiple threads updating the same nodes simultaneously is low.



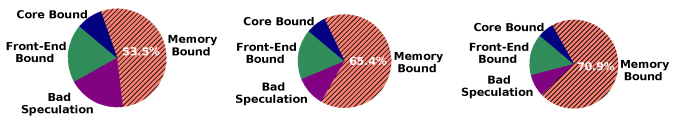
(a) HLA-DRB1. (b) MHC. (c) Chr.1.

Fig. 4: **Scaling of *odgi-layout*.**

Fig. 4 reveals a linear scaling pattern of *odgi-layout* with CPU threads. However, the CPU cannot fully take advantage of the high degree of data-level parallelism that exists in the inner loop, particularly for larger graphs. For instance, the pangenome graph of the human chromosome 1 (Chr.1) requires six billion node pair updates per iteration, making it less ideal for a CPU with a limited number of threads.

B. Memory-Bound

We use the top-down approach proposed in [33] to identify the performance bottleneck. Fig. 5 displays the results of the bottleneck analysis. It is apparent that *odgi-layout* uses a significant portion of the microarchitecture's pipeline slots for memory operations on all three graphs, demonstrating its memory-bound nature. We then profile the memory stall and cache performance of *odgi-layout*. As illustrated in Table II, workload performance is bottlenecked by a high percentage of memory stall cycles and a significant miss rate of last-level cache (LLC) loads. As a result, the memory operations dominate the run time of the layout process.



(a) HLA-DRB1. (b) MHC. (c) Chr.1.

Fig. 5: **Microarchitecture bottleneck analysis with VTune.**

TABLE II: **Memory stall and cache performance of *odgi-layout* profiled by Perf.**

Pangenome	HLA-DRB1	MHC	Chr.1
Run Time (h:mm:ss)	0:00:00.4	0:01:47	2:32:38
Memory Stall Cycle Percentage	67.67%	78.07%	77.38%
LLC-load Miss Rate	75.09%	77.84%	89.88%

We observe that the random memory accesses to $\mathbf{L}$ (lines 12, 13) and obtaining d_{ref} outweigh the computational part (lines 14, 15). Given the massive size of these data structures, e.g., the graph of Chr.1 is composed of 11.1M nodes, the scope for data reuse is severely limited due to random memory access. This leads to the unusually high LLC load miss rate.

Additionally, the repeated use of pseudo-random number generator (PRNG) (lines 5, 6, 8, 10, 12, 13) increases memory traffic. *odgi-layout* uses Xoshiro256+ [34], a PRNG utilizing linear-feedback shift registers (LFSR). LFSR-based PRNG is known for its low computational requirements, which adds to the memory-bound nature of the layout process.

C. Randomness & Layout Quality

Randomness is essential for fast convergence and high-quality layout generation in this path-guided SGD algorithm. This is consistent with the discussion in the paper by Zheng et al. [24], from which the path-guided SGD algorithm was adopted. Random path and node pair selections (lines 5, 8, 10) are performed in each step to ensure the layout quality, as a naïve iteration could cause the algorithm to get stuck in local minima due to biases. Fig. 6 shows a non-converged layout created by forcing all selected pairs of nodes to be 10 hops away. This node pair selection scheme significantly reduces randomness in node selection and does not converge within the same number of iterations. In contrast, the optimized layout of the same gene shown in Fig. 2 clearly reveals the variants, which are the primary targets of pangenome graph layout.

Fig. 6: **Layout of poor quality** — the yellow box captures the "Large Structural Variant" region in Fig. 2.

IV. Pangenome Graph Layout in PyTorch

As previously discussed, CPUs cannot fully exploit the substantial data-level parallelism within the pangenome graph layout algorithm. Since the algorithm relies on SGD-based optimization, adopting PyTorch [35], a deep learning framework optimized for gradient computation, is an attractive option for implementing the layout algorithm on GPUs.

In this section, we introduce a PyTorch-based implementation of the algorithm and assess its performance on the MHC pangenome graph with an NVIDIA RTX A6000 GPU. We employ NVIDIA Nsight Systems [36] for detailed profiling. Our analysis not only reveals the limitations of a basic PyTorch implementation but also underscores the challenges in achieving effective GPU acceleration of the pangenome layout.

A. Implementation and Performance Analysis

We utilize PyTorch to solve the layout optimization problem following the neural network training procedure — Each data instance is a node pair (n_i, n_j), with its ground-truth label as d_{ref}. The layout coordinates $\mathbf{L}$ act as the adjustable weights that are updated in each step based on the gradient of the stress function. We process a batch of node pairs simultaneously to leverage the data-level parallelism of the algorithm.

The performance of the PyTorch implementation on the MHC pangenome is shown in Table III, where we measure the GPU run time and compare it to the 32-thread CPU baseline, which completes in 107 seconds. The run time decreases as the batch size increases, up to a batch size of 1 million. Beyond this point, there is no further linear scaling.

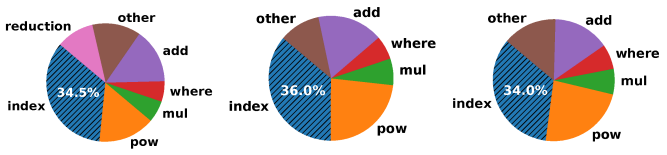
We also assess the layout quality to understand the impact of larger batch sizes by visual inspection. As mentioned in Sec. III-A, excessive asynchronous updates by too many threads could reduce the effectiveness of these updates, resulting in layout quality degradation. This is reflected in the increasing node stress with larger batch sizes. By visual inspection, the design with a batch size of 10M has some layout quality degradation, and the design with a batch size of 100M does not converge to a valid layout.

TABLE III: **Performance of the PyTorch implementation** — the speedup is compared to the 32-thread CPU baseline.

Batch Size	Run Time (s)	Speedup	Quality
10K	702.2	0.2x	Good
100K	67.3	1.6x	Good
1M	15.6	**6.8x**	Good
10M	14.3	7.5x	Satisfying
100M	11.8	9.1x	Poor

The PyTorch implementation achieves a 6.8× speedup over the CPU baseline on MHC. However, this approach does not fully exploit the potential of the GPU due to the lack of tailored optimizations for the memory-bound nature of the application and the GPU architecture.

Fig. 7 presents the breakdown of kernel time for the PyTorch implementation when using different batch sizes, demonstrating that the indexing kernel consumes the most time. Taking into account the profiling results in Sec. III-B, it is evident that memory operations are the primary time-consuming operations on both the CPU and GPU. Given the algorithm's inherent randomness leading to a random memory access pattern, combined with this memory operation dominance, an effective data layout is crucial to enhance performance on both hardware platforms. However, neither the CPU baseline nor the current PyTorch implementation have a customized data layout.



(a) Batch size = 100K. (b) Batch size = 1M. (c) Batch size = 10M.

Fig. 7: **Kernel time breakdown of the PyTorch implementation, profiled by NVIDIA Nsight Systems** — only kernels accounting for over 2% of total GPU time are included. The shaded `index` is the memory operation.

Another challenge arises from PyTorch's tensor-based programming model. The implementation groups multiple node pairs into long tensors for computation and memory operations. Due to the large number of node pairs, multiple batches are needed per iteration, resulting in numerous CUDA kernel launches, as shown in Table IV. This leads to significant overhead in kernel launches and unnecessary implicit synchronization between kernels, which is not needed for this specific application that permits asynchronous Hogwild! style updates.

TABLE IV: **CUDA kernel launching overhead.**

Batch Size	100K	1M	10M
CUDA kernels launched (#)	6,562,860	651,480	64,080
Time percentage of CUDA API	76.4%	20.2%	2.1%

Furthermore, using PyTorch, a high-level framework, makes it challenging to implement low-level, customized optimizations tailored to the GPU architecture. The highly-optimized kernels that PyTorch relies on come from its backend libraries. These are fixed and not tailored for our specific workload, which means that issues like conditional branching (lines 7, 9 in Alg. 1) and uncoalesced memory access can still significantly impair GPU performance.

B. Challenges to Efficient GPU Offloading

By characterizing the pangenome graph layout workload and implementing a basic PyTorch implementation, we have identified several challenges that must be addressed in order to fully leverage the power of GPUs.

- Numerous CUDA kernels launched by PyTorch lead to a notable overhead due to redundant memory operations and synchronization. This is addressed in Sec. V-A.
- The application is memory-bound on both CPUs and GPUs. Dominant memory operations and irregular access patterns necessitate an effective data layout to minimize memory traffic. This is addressed in Sec. V-B1.
- The conditional branching and uncoalesced memory access can degrade GPU performance. This is addressed in Sec. V-B2 and Sec. V-B3.

V. Optimized GPU Implementation

In this section, we describe our GPU design with customized optimizations to address the challenges highlighted in Sec. IV-B. First, we introduce a base CUDA kernel for pangenome graph layout to exploit the high degree of data-level parallelism. Then, we detail three optimization techniques: a *cache-friendly data layout* for the pangenome graph, *coalesced random states*, and *warp merging*.

A. CUDA Kernel for Pangenome Graph Layout

Our base CUDA kernel design for pangenome graph layout is shown in Fig. 8. Each GPU thread runs the update steps (lines 4-16 in Alg. 1) in parallel. Within a single CUDA kernel launch, all GPU threads collectively contribute to completing the N_{steps} steps required per iteration.

The memory-bound nature of the algorithm would lead to frequent memory stalls. When a warp is stalled, the GPU warp scheduler attempts to switch to another available warp

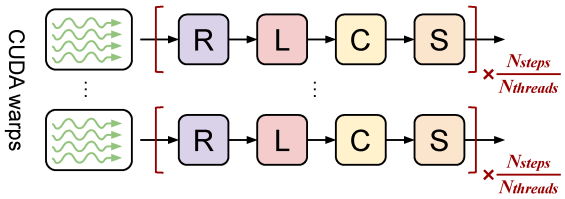


Fig. 8: **CUDA kernel execution** — one update step includes pseudo random number generation (R), node pair loading (L), computing the updated value (C), and storing the result (S).

to hide memory latency. The abundant data-level parallelism in our design ensures the amount of available warps, thereby improving streaming multiprocessors (SM) utilization.

In our method, a single CUDA kernel is launched per iteration, with inter-block synchronization occurring only after all steps in an iteration are completed. Therefore, with the default setting of N_{iters} of 30, a total of 31 CUDA kernels are launched, including one additional kernel launch for initialization. This achieves implicit kernel fusion compared to our preliminary PyTorch implementation, which greatly reduces the overhead due to the numerous CUDA kernels launched, as discussed in Table IV.

Here, we also build a lean data structure specifically for the pangenome graph layout application. As a part of the comprehensive pangenome analysis framework ODGI, the current SOTA *odgi-layout* uses the ODGI data structure. Therefore, the data structure includes numerous fields, some of which are not relevant to *odgi-layout*, resulting in a suboptimal data structure for pangenome graph layout.

The lean data structure in our CUDA kernel retains only the data fields used in the pangenome graph layout process. For instance, the ODGI data structure represents the nucleotide sequence as a string, and invoking the `.size()` method returns the size; our lean data structure directly stores the sequence length since the content of the string is not used in the pangenome graph layout. Note that this lean data structure can be easily transformed from the ODGI data structure, leading to an easy integration into the ODGI framework.

Since *odgi-layout* and the external libraries used [21] were developed for the multi-core CPU, the data structures are heavily dependent on the use of dynamic containers such as vectors. The GPU provides limited support for dynamic data constructs, so we manually implement the necessary data structures and functions in our CUDA kernel.

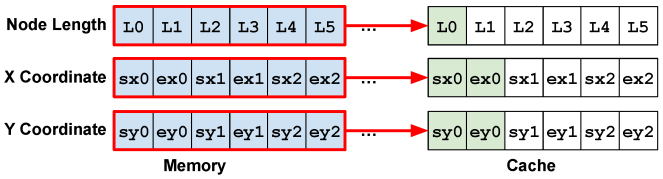
B. Kernel Optimizations

To address the memory-bound nature of the application, we introduce three kernel optimization methods: a *cache-friendly data layout* for pangenome graphs to improve cache locality, *coalesced random states* to enable coalesced memory accesses, and *warp merging* to reduce warp divergence.

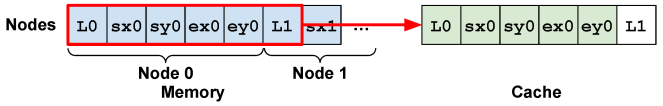
1) Cache-friendly Data Layout: Our data structure for the pangenome graph layout includes node data and path data. The node data includes the sequence length of each node and the coordinates of the start and end points of the visualization, while the path data consists of the node ID, path ID, position, and orientation of each node as it traverses the paths.

ODGI maintains its core data structure for pangenome graph and develops auxiliary structures for the tools built upon it. For instance, the x and y coordinates, which are used exclusively in *odgi-layout*, are organized into two arrays separate from the primary graph data structure. The data structure of our base CUDA kernel follows this design, resulting in a struct-of-arrays (SoA) layout. This has a negative impact on cache performance for the pangenome graph layout workload. To solve this problem, we propose a cache-friendly data layout by repacking data to match the memory access pattern of Alg. 1.

Fig. 9 compares the proposed cache-friendly data layout with the original one in terms of access to node data during an update step. When using the original data layout, updating a single node requires three separate memory accesses for three different arrays. This is illustrated in Fig. 9a. Although neighboring node data is cached, there is a high chance of eviction due to the random selection of node pairs.



(a) Original data layout. Every node incurs three memory accesses; the majority of cached data are not used due to randomness.



(b) Cache-friendly data layout. One memory access for one node.

Fig. 9: **Cache-friendly data layout** — `Li` is the length of node `i`; `sxi`, `syi`, `exi`, `eyi` are the x and y coordinates of the start and end points of the line segment for node `i`.

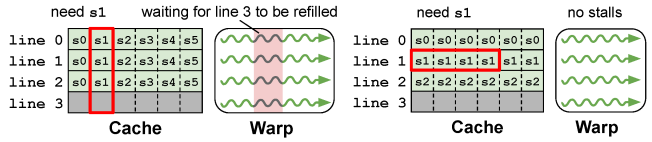
In contrast, we use an array-of-structs (AoS) layout for node-related data, ensuring a cache-friendly design. Only one memory access is necessary for each node's data retrieval, as shown in Fig. 9b. Since memory accesses to the start and end point coordinates are contiguous (lines 12, 13 in Alg. 1), this packing scheme improves spatial locality, thus removing traffic to higher-level caches and DRAM. The same principle applies to the path data which is not discussed in detail here.

2) Coalesced Random States: Pseudo random number generator (PRNG) is heavily used in the algorithm. The CUDA cuRAND library [37] utilizes the xorshift PRNG [38], a type of LFSR with low computational requirements.

To maintain layout quality, we map a set of random states to each SM, enabling each thread within a block to have its own random state. This ensures that threads generate uncorrelated random numbers, eliminating potential biases. However, this approach leads to a large number of memory accesses to the random states with concurrent running threads, which becomes the primary bottleneck for PRNG. As the GPU cache is

shared by multiple warps running asynchronously, one warp's pangenome graph data may displace another warp's random states in the cache, increasing the risk of eviction.

The cuRAND implementation represents each random state by a structure consisting of six 32-bit fields. This object-oriented design forms an AoS data layout, with each thread having its own random state. However, this data layout results in uncoalesced memory access to the random state, since the same field in different threads is not in contiguous memory. Uncoalesced memory access to any random state requires much more frequent cache refills if some cache lines (e.g., cache line 3 in Fig. 10a) are evicted. This pattern amplifies global memory accesses and causes memory stalls.



(a) Original random states. (b) Coalesced random states.

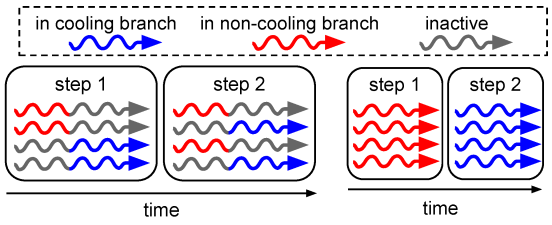
Fig. 10: **Coalesced random states** — in (a), a refill is required for any evicted cache line; in (b), a refill only happens when the warp accesses exactly the evicted cache line.

To solve this problem, we introduce a *coalesced random states* method by transforming the AoS data layout into the SoA data layout. As shown in Fig. 10b, this switch facilitates coalesced memory accesses to random states within a warp, storing the same field from multiple threads within the same cache line. In this way, a cache is only refilled from global memory when a warp requires an evicted cache line.

3) Warp Merging: The conditional branching (lines 7, 9 in Alg. 1) is crucial for generating a high quality pangenome graph layout. The non-cooling branch uniformly selects node pairs to create the coarse-grained layout, while the cooling branch selects node pairs at closer proximity with a power law distribution to refine the layout. However, this conditional branching structure leads to warp divergence. Since all 32 threads within a warp execute the same instruction, divergence forces some threads to idle, degrading GPU performance.

To solve this problem, we introduce the warp merging method. As indicated in Fig. 11, all threads within a warp select the same branch in an update step, keeping the threads constantly active. This method is achieved by using a control thread within each warp to randomly select the branch. The selection is then stored in shared memory, accessible to all threads within the same warp.

While warp merging causes threads within a single warp to select the same branch, resulting in reduced intra-warp randomness, the presence of multiple concurrently running warps on various SMs ensures different branches are chosen across warps. Consequently, the overall distribution of threads taking each branch remains consistent with the original algorithm, thereby preserving layout quality.



(a) No warp merging. (b) With warp merging.

Fig. 11: **Warp merging** — in (a), conditional branches cause warp divergence, leading to suboptimal thread utilization; in (b), all threads within a warp are active by selecting the same branch in an update step.

VI. A Quantitative Metric for Pangenome Layouts

In our GPU implementation, we leverage a notably higher degree of data-level parallelism in comparison to the CPU baseline. As detailed in Section III-A and examined through experiments in Section IV-A, excessive parallelism may challenge the sparsity assumption underlying the Hogwild! asynchronous update, potentially compromising layout quality. Visual inspection, while useful, is subjective and not scalable since it relies on human evaluation of the results. Consequently, there is a crucial need to quantify the quality of the GPU-generated layouts.

In this section, we incorporate the stress metric, widely used in general graph layouts, into the pangenome graph to propose the *path stress* with a GPU implementation, and then further apply sampling to solve the scalability issue.

A. Path Stress

Prior studies [39], [40] have introduced various quantitative metrics to evaluate the aesthetic quality of general graph layouts, including stress, the number of edge crossings, the uniformity of edge lengths. However, each metric focuses on a single aspect, while some criteria contradict each other [41]. So far, there is no agreement on the most effective metric [42], and the metric selection highly depends on which features of the graph you want to highlight in each use case [40], [43].

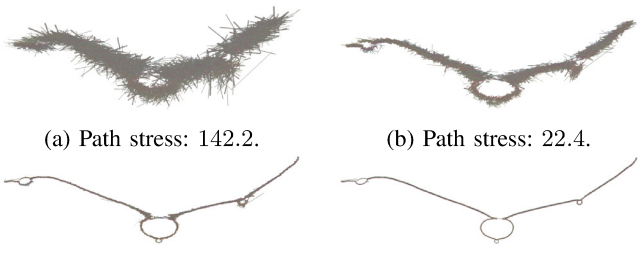
Therefore, since the pangenome graph layout algorithm is based on the popular energy-based algorithms by minimizing stress [24], [44], [45], we incorporate stress (line 14 in Alg. 1) with the unique path property of the pangenome graph, forming the *path stress*, defined in Equation 1.

$$path_stress = \frac{\sum_{p \in P} \sum_{n_i, n_j \in p} stress(n_i, n_j)}{N_{total_node_pairs}} \quad (1)$$

Here $stress(n_i, n_j)$ is the average stress of all four combinations of the start and end points of node n_i and n_j. The path stress is calculated by averaging the stress across all node pairs on all paths. The key distinction between path stress and the standard stress used for general graphs is that path stress only considers node pairs on the same path. This aligns with

the layout algorithm, as d_{ref} only considers distances within the same path.

We implement the path stress with a CUDA kernel to speedup the computation by mapping a pair of nodes to each GPU thread, and then aggregating partial results with a reduction tree. Fig. 12 shows how path stress can differentiate between pangenome graph layouts of varying qualities. The layout with a lower path stress is considered more legible and aesthetically sound, thereby more effectively revealing the structural information of the pangenome graph.



(a) Path stress: 142.2. (b) Path stress: 22.4.

(c) Path stress: 1.3. (d) Path stress: 0.07.

Fig. 12: **Layouts of HLA-DRB1 of different qualities.**

B. A Scalable Metric: Sampled Path Stress

Although path stress can effectively present layout quality, it has a quadratic computational complexity in terms of nodes. This poses a significant challenge on scalability even with the compute power of GPUs. As shown in Table V, it would require 194 GPU hours with an NVIDIA RTX A6000 GPU to compute the path stress of a human Chr.1 pangenome graph layout, which is impractical. Therefore, there is a need for a metric scalable to chromosomal pangenome graphs.

TABLE V: **Run time of metrics computation.**

Pangenome	# of Nodes	RT of Path Stress	RT of Sampled Path Stress
HLA-DRB1	5.0×10^3	1.6 sec	0.3 sec
MHC	2.3×10^5	53.0 min	6.5 sec
Chr.1	1.1×10^7	(Est.) 194.0 hour	5.5 min

We propose a more scalable metric, *sampled path stress*, which estimates overall path stress by randomly sampling a total of n pairs of visualization nodes $(\mathbf{v_i}, \mathbf{v_j})$, whose corresponding nodes are on the same path. Equation 2 defines sampled path stress, where $\mathcal{S}$ stands for the set of sampled nodes. By default, we sample $n = 100|p|$ node pairs in each path, where $|p|$ is the number of nodes in path p; each node is expected to be sampled 100 times within its path.

$$sampled_path_stress = \frac{\sum_{p \in P} \sum_{(\mathbf{v_i}, \mathbf{v_j}) \in \mathcal{S}} stress_{ij}(\mathbf{v_i}, \mathbf{v_j})}{n} \quad (2)$$

Sampled path stress, which is the mean of the sample (noted as μ), would converge to a normal distribution based on the central limit theorem [46], [47]. Therefore, we also compute the 95% confidence interval to validate the sampling coverage. This is computed by another pass through the sampled stress terms to get the standard deviation σ, and derived from $CI_{95\%} = \mu \pm 1.96\sigma/\sqrt{n}$.

Applying sampling makes the metric computation linear in complexity, allowing the metric computation for chromosomal pangenome graphs with millions of nodes to be done in minutes, as shown in Table V.

To check the correctness of sampled path stress, we compare it against path stress with 1824 small-sized pangenome graph layouts, where path stress computation is feasible. Fig. 13 demonstrates that sampled path stress closely approximates path stress with a correlation of 0.995. We also verify that sampled path stress remains consistent with different random seeds for a given layout.

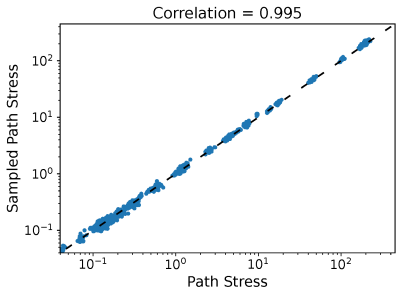


Fig. 13: **Linear correlation** — sampled path stress closely approximates the entire path stress.

Thus, we adopt sampled path stress as the scalable quantitative metric for evaluating layout quality.

VII. Evaluation

A. Experiment Setup

We utilize a 32-core Intel Xeon Gold 6246R CPU@3.4GHz, an NVIDIA RTX A6000 GPU, and an NVIDIA A100 GPU for hardware setup, with GCC 10.2.1 for compilation.

For overall performance analysis, our GPU design is tested on both an NVIDIA RTX A6000 with CUDA 11.7 and an NVIDIA A100 with CUDA 12.2. The ablation study is conducted only on the NVIDIA RTX A6000, utilizing NVIDIA Nsight Compute [48] 2022.2 and Linux Perf [30] as profiling tools. We perform another case study to explore the performance-quality trade-off with the RTX A6000. The multi-threaded CPU baseline is *odgi-layout* [20].

We use the human pangenome reference dataset released by the HPRC [7], composed of 24 chromosomal pangenome graphs, from Chr.1 to Chr.22, Chr.X, and Chr.Y. As detailed in Table VI, these graphs contain millions of nodes and are characterized by their notably low node degree and density.

TABLE VI: **Properties of the human pangenome graphs.**

	# Nuc.	# Nodes	# Edges	# Paths	$\overline{deg}$	Density
Min	8.8×10^7	3.2×10^5	307	4.4×10^4	1.4	1.3×10^{-7}
Max	1.1×10^9	1.1×10^7	3,029	5.0×10^5	1.4	4.4×10^{-6}
Mean	3.0×10^8	4.0×10^6	1,295	2.3×10^5	1.4	3.5×10^{-7}

TABLE VII: **Run time and speedup** — the run time format is in h:mm:ss.

Pan.	CPU	A6000	Speedup	A100	Speedup	Pan.	CPU	A6000	Speedup	A100	Speedup	Pan.	CPU	A6000	Speedup	A100	Speedup
Chr.1	2:32:38	0:04:59	30.6x	0:02:42	56.5x	Chr.9	1:16:49	0:02:53	26.6x	0:00:55	83.8x	Chr.17	1:03:45	0:02:01	31.7x	0:01:07	57.1x
Chr.2	1:17:03	0:03:33	21.7x	0:01:01	75.8x	Chr.10	0:48:34	0:02:22	20.6x	0:00:44	66.2x	Chr.18	0:50:29	0:01:50	27.6x	0:01:08	44.6x
Chr.3	1:28:41	0:03:27	25.7x	0:01:31	58.5x	Chr.11	0:56:25	0:02:07	26.7x	0:00:37	91.5x	Chr.19	0:40:23	0:01:29	27.3x	0:00:27	89.8x
Chr.4	1:47:32	0:03:40	29.3x	0:02:06	51.2x	Chr.12	0:44:05	0:02:07	20.9x	0:00:49	54.0x	Chr.20	0:51:34	0:01:30	34.3x	0:01:01	50.7x
Chr.5	1:41:09	0:03:19	30.5x	0:01:07	90.6x	Chr.13	1:03:32	0:02:22	26.8x	0:00:53	71.9x	Chr.21	0:44:18	0:01:26	30.9x	0:00:38	69.9x
Chr.6	1:13:55	0:02:49	26.3x	0:01:27	51.0x	Chr.14	0:51:21	0:02:04	24.9x	0:00:46	67.0x	Chr.22	0:39:59	0:01:37	24.8x	0:00:30	80.0x
Chr.7	1:16:46	0:03:00	25.6x	0:01:34	49.0x	Chr.15	1:11:33	0:02:52	25.0x	0:01:16	56.5x	Chr.X	1:04:06	0:01:49	35.4x	0:00:49	78.4x
Chr.8	1:17:27	0:02:57	26.3x	0:01:41	46.0x	Chr.16	2:19:47	0:04:56	28.3x	0:12:58	10.8x	Chr.Y	0:01:55	0:00:03	36.9x	0:00:04	28.7x
Geometric Mean			**27.7x**		**57.3x**												

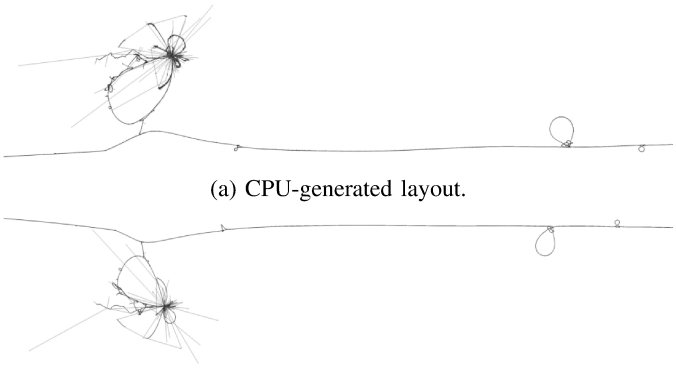
TABLE VIII: **Layout quality comparison with sampled path stress (SPS)** — SPS ratio is computed by $\text{GPU}_{\text{SPS}}/\text{CPU}_{\text{SPS}}$.

Pan.	CPU $\text{CI}_{95\%}$	A6000 $\text{CI}_{95\%}$	SPS ratio	A100 $\text{CI}_{95\%}$	SPS ratio	Pan.	CPU $\text{CI}_{95\%}$	A6000 $\text{CI}_{95\%}$	SPS ratio	A100 $\text{CI}_{95\%}$	SPS ratio	Pan.	CPU $\text{CI}_{95\%}$	A6000 $\text{CI}_{95\%}$	SPS ratio	A100 $\text{CI}_{95\%}$	SPS ratio
Chr.1	[0.77, 1.72]	[0.86, 1.28]	0.86	[0.88, 1.28]	0.87	Chr.9	[0.58, 2.93]	[0.73, 1.34]	0.59	[-0.15, 3.05]	0.83	Chr.17	[0.45, 0.45]	[0.60, 0.61]	1.34	[0.58, 0.58]	1.29
Chr.2	[0.31, 0.76]	[0.21, 0.29]	0.47	[0.35, 0.56]	0.85	Chr.10	[0.13, 0.17]	[0.14, 0.19]	1.13	[0.13, 0.17]	1.04	Chr.18	[0.49, 0.61]	[0.53, 0.63]	1.05	[0.54, 0.57]	1.00
Chr.3	[0.26, 0.28]	[0.29, 0.31]	1.12	[0.29, 0.30]	1.09	Chr.11	[0.12, 0.32]	[0.14, 0.18]	0.72	[0.14, 0.19]	0.75	Chr.19	[0.17, 0.19]	[0.22, 0.26]	1.30	[0.19, 0.21]	1.11
Chr.4	[0.28, 0.31]	[0.28, 0.30]	1.00	[0.28, 0.30]	1.00	Chr.12	[0.12, 0.17]	[0.13, 0.14]	0.96	[0.13, 0.16]	0.99	Chr.20	[0.19, 0.97]	[0.38, 0.41]	0.68	[0.38, 0.41]	0.67
Chr.5	[0.18, 0.20]	[0.22, 0.27]	1.26	[0.20, 0.23]	1.13	Chr.13	[0.38, 0.39]	[0.46, 0.51]	1.26	[0.41, 0.48]	1.16	Chr.21	[0.36, 0.47]	[0.41, 0.54]	1.15	[0.36, 0.72]	1.31
Chr.6	[0.30, 0.31]	[0.32, 0.33]	1.05	[0.31, 0.32]	1.03	Chr.14	[0.19, 0.33]	[0.20, 0.88]	2.11	[0.15, 0.61]	1.48	Chr.22	[0.37, 0.42]	[0.23, 1.14]	1.73	[0.43, 0.50]	1.17
Chr.7	[0.28, 0.29]	[0.29, 0.30]	1.04	[0.29, 0.33]	1.08	Chr.15	[0.85, 1.60]	[1.20, 1.85]	1.24	[1.13, 1.67]	1.14	Chr.X	[0.49, 0.51]	[0.58, 0.61]	1.19	[0.58, 0.60]	1.17
Chr.8	[0.27, 0.27]	[0.27, 0.28]	1.02	[0.27, 0.28]	1.02	Chr.16	[0.67, 0.68]	[0.69, 0.71]	1.03	[0.71, 0.72]	1.05	Chr.Y	[0.58, 0.89]	[-0.26, 3.66]	2.31	[0.63, 0.75]	0.94
Geometric Mean			**1.08**		**1.03**												

B. Overall Performance

Table VII shows the overall performance for all human chromosomes. Our optimized GPU design achieves a 57.3× speedup on A100, and a 27.7× speedup on RTX A6000 over the 32-thread CPU baseline *odgi-layout*. This reduces the average computation time from 1-2 hours down to just a few minutes.

We evaluate the layout quality of GPU-generated layouts using both quantitative and qualitative methods. Quantitatively, we measure sampled path stress, as shown in Table VIII, where the average ratio of sampled path stress between GPU and CPU layouts is close to 1, indicating no quality loss in the GPU-generated layouts. Qualitatively, visual inspection confirms that GPU-generated layouts do not have noticeable differences compared to CPU-generated layouts, as demonstrated in Fig. 14 for Chr.7. Additionally, we conduct 15 runs for each pangenome and confirm the consistency and repeatability of these layouts.



(a) CPU-generated layout.

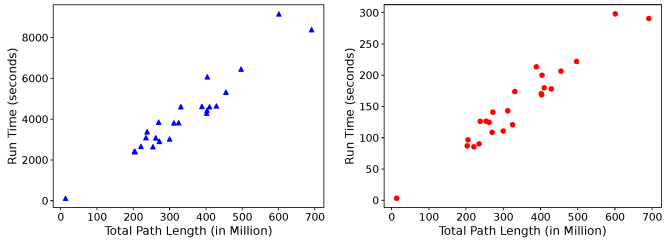
(b) GPU-generated layout.

Fig. 14: **CPU and GPU-generated layouts of Chr.7** — only the central, most complex parts are displayed, as the entire chromosome is too long.

The 27.7× speedup achieved by our optimized GPU design on the same NVIDIA RTX A6000 GPU is significantly higher than the 6.8× improvement achieved by our initial PyTorch implementation, as discussed in Sec. IV. This demonstrates that the custom optimizations in our GPU design effectively exploit the GPU's computing power.

Our design can be seamlessly integrated into the ODGI [28] framework to facilitate the adoption of our GPU implementation. To enable it, a user can simply add the `--gpu` argument, making the solution effortlessly accessible.

We also perform a scalability study on human pangenome graphs. Fig. 15 demonstrates linear scaling in both CPU and GPU implementations. This result aligns with the expectation, as the number of updates is proportional to the total path length, which is the sum of the nodes in each path.



(a) CPU. (b) RTX A6000 GPU.

Fig. 15: **Scalability study on the size of pangenomes.**

C. Ablation Study

Fig. 16 shows the incremental performance gains achieved with each optimization. Our approach begins with a base CUDA kernel to exploit the data-level parallelism. Building on this, we develop an optimized CUDA kernel by introducing three kernel optimization methods.

In the following sections, we evaluate the effects of the individual kernel optimizations. We apply the methods to the base CUDA kernel individually and evaluate its effects, demonstrating improvements in both run time and key performance metrics. Since we find similar improvements in all chromosomes, we show here only the results for Chr.1.

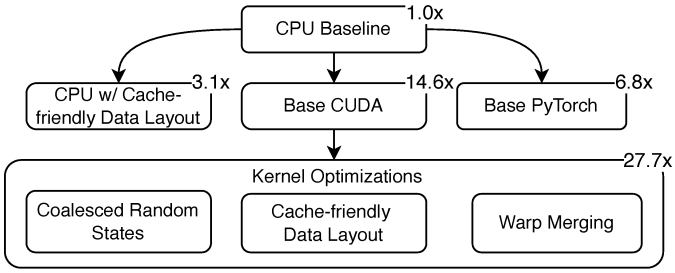


Fig. 16: **Speedup through successive optimizations** – the baseline is *odgi-layout* on CPU.

1) Cache-friendly Data Layout (CDL): Since CDL is effective across both CPU and GPU, we apply it on both the base CUDA kernel and the CPU baseline. As shown in Table IX, the improved spatial locality with CDL significantly reduces Last Level Cache (LLC) loads and LLC misses on CPU, and reduces DRAM access on GPU.

TABLE IX: **Effects of cache-friendly data layout**.

Method		w/o CDL	w/ CDL	Improv.
CPU	LLC-loads (#)	3.0×10^{12}	9.4×10^{11}	3.2×
	LLC-load-misses (#)	2.7×10^{12}	8.1×10^{11}	3.3×
	CPU Run Time (s)	9,158.4	2,935.2	3.1×
GPU	DRAM access (GB)	5,191.9	3,974.4	1.3×
	GPU Run Time (s)	569.4	393.1	1.4×

2) Coalesced Random States (CRS): Table X reports the effects of CRS. The "L1 sectors per request" metric reflects the level of memory request coalescence within a warp. Here, a request denotes a single instruction requesting a memory operation, and a sector represents an aligned 32B chunk of memory. Each request may access one or more sectors. Hence, fewer sectors per request indicate improved coalescence of memory requests. The CRS method notably decreases the L1 sectors per request, thereby reducing memory accesses to L1, L2 caches, and DRAM.

TABLE X: **Effects of coalesced random states**.

Method	w/o CRS	w/ CRS	Improv.
L1 Sectors / Req (#)	26.8	9.9	2.7×
L1 Cache Access (GB)	8,686.7	4,787.7	1.8×
L2 Cache Access (GB)	7,498.9	4,339.3	1.7×
DRAM Access (GB)	5,191.9	4,077.8	1.3×
GPU Run Time (s)	569.4	471.7	1.2×

3) Warp Merging (WM): As seen in Table XI, using WM significantly reduces the number of instructions executed. The average number of active threads is increased to 27.9, which is close to the full complement of 32 threads within a warp, indicating a considerable reduction in warp divergence.

D. A Case Study: Explore the Performance-Quality Trade-off with Sampled Path Stress

Sampled path stress proposed in Sec. VI-B allows us to evaluate a chromosomal layout quantitatively in minutes, enabling the exploration of the effects of algorithmic changes on the layout quality of large-scale pangenomes.

TABLE XI: **Effects of warp merging**.

Method	w/o WM	w/ WM	Improv.
Executed Instructions (# in billions)	131.3	90.1	1.5×
Avg. Active Threads Per Warp (#)	20.5	27.9	1.4×
GPU Run Time (s)	569.4	527.4	1.1×

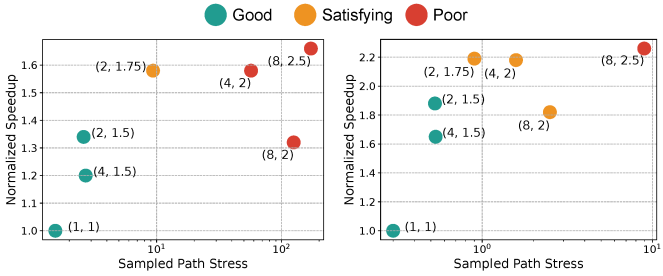
While randomness is crucial for achieving high-quality layouts, it limits data reuse, which adversely affects performance. We conduct a case study by applying warp-level data reuse on top of the optimized GPU design to explore the performance-quality trade-off with sampled path stress.

1) Methods: We aim to increase data reuse with minimal randomness degradation. This is accomplished by shuffling node data within the same warp using CUDA warp-level primitives, enabling direct data sharing between thread registers within the same warp without using shared or global memory.

The data reuse scheme consists of data reuse factor (DRF) and step reduction factor (SRF). This modified approach increases the number of updates per step by *DRF*, and reduces the number of steps by *SRF*. Each step involves selecting one node pair but performing multiple updates via warp shuffling. Through warp shuffling, we reuse the cached data to randomly form a new node pair. This method reduces randomness of node selection, potentially affecting the layout quality.

2) Results: Fig. 17 illustrates the study of the trade-off between performance (represented by normalized speedup) and quality (represented by sampled path stress) in Chr.1 and Chr.2. Layouts with stress less than twice that of baseline layouts are considered as "good", less than ten times as "satisfying", and more than ten times as "poor".

An increase in both *DRF* and *SRF* generally leads to an increase in sampled path stress, indicating a loss of layout quality. This trend is consistent for most input data. Overall, schemes with *DRF* of 2 usually produce good or satisfying layouts, while schemes with a *DRF* of 8 tend to produce poor layouts in many cases. This is due to reusing data 8 times within a warp's 32 threads, which greatly reduces randomness in node pair selection.



(a) Chr.1. (b) Chr.2.

Fig. 17: **Design space exploration on data reuse schemes** — each datapoint represents a $scheme = (DRF, SRF)$.

Leveraging scalable sampled path stress allows us to explore the trade-offs between performance and layout quality. Tested on the RTX A6000 GPU across all 24 human pangenome graphs, we discover that it is possible to achieve an additional 1.5x speedup over the optimized GPU implementation, while still maintaining good layout quality.

VIII. Related Work

Several widely-used graph layout tools include Gephi [23], NetworkX [49], and Graphviz [50]. Efforts have been made to accelerate these tools on GPUs, as seen in works like [51], [52]. However, the distinct biological meaning of nodes and paths in pangenome graphs limit these tools' suitability.

Numerous pangenome graph layout tools have been developed to better understand the intricate relationships and variations among genomes [3]. AGB [18] and VG view [21] employ the rank-based layout algorithm on the Graphviz backend. GfaViz [22] and BandageNG [19] adopt the force-directed layout algorithm on the OGDF [53] backend. SGTK [54] applies the force-directed layout algorithm on the Cytoscape.js [55] backend. Despite their approaches, none have demonstrated scalability to the giga-basepair level, as highlighted in a comprehensive review [3] of pangenome graph visualization tools. Among the available tools, *odgi-layout* [20] stands to be the only tool capable of scaling to whole-chromosome pangenome graphs containing millions of nodes. Despite taking hours, it far surpasses the prior leading tool, BandageNG, which fails to produce a layout within 7 days [20].

Many efforts have focused on accelerating genomics applications, from GPU-accelerated sequence alignment [56]–[60], metagenome assembly [61] and classification [62], to custom hardware for read assembly [63] and read mapping [64]. Yet, there is a noticeable gap in acceleration work for pangenome graphs, presenting ample opportunity.

IX. Conclusion and Future Work

We present a fast GPU-based pangenome graph layout solution, achieving a 57.3× speedup on average over the current state-of-the-art multi-threaded CPU baseline, and 18.5× speedup over our own optimized version of the CPU solution. We leverage the compute power of GPUs with custom optimizations to address the memory-bound and randomness challenges. Our work enables the layout of the entire human pangenome dataset in just a few minutes, which greatly facilitates pangenomics research. For future work, we believe scaling our work to a multi-GPU setup is essential to meet the rapid increase in genome data, and extending to other pangenome analysis applications in the face of increasing data availability.

Acknowledgment

For LLM usage disclosure, ChatGPT was utilized to assist and provide suggestions on polishing the text of this paper.

We are grateful to all the reviewers for their valuable comments. This research is supported in part by NSF Awards #2118709 and #2118743.

References

[1] H. Cheng, M. Asri, J. Lucas, S. Koren, and H. Li, "Scalable telomere-to-telomere assembly for diploid and polyploid genomes with double graph," *arXiv preprint arXiv:2306.03399*, 2023.

[2] M. Rautiainen, S. Nurk, B. P. Walenz, G. A. Logsdon, D. Porubsky, A. Rhie, E. E. Eichler, A. M. Phillippy, and S. Koren, "Telomere-to-telomere assembly of diploid chromosomes with verkko," *Nature Biotechnology*, pp. 1–9, 2023.

[3] J. M. Eizenga, A. M. Novak, J. A. Sibbesen, S. Heumos, A. Ghaffaari, G. Hickey, X. Chang, J. D. Seaman, R. Rounthwaite, J. Ebler, M. Rautiainen, S. Garg, B. Paten, T. Marschall, J. Sirén, and E. Garrison, "Pangenome graphs," *Annual review of genomics and human genetics*, vol. 21, pp. 139–162, 2020.

[4] B. Paten, A. M. Novak, J. M. Eizenga, and E. Garrison, "Genome graphs and the evolution of genome inference," *Genome research*, vol. 27, no. 5, pp. 665–676, 2017.

[5] A. A. Golicz, P. E. Bayer, P. L. Bhalla, J. Batley, and D. Edwards, "Pangenomics comes of age: from bacteria to plant and animal applications," *Trends in Genetics*, vol. 36, no. 2, pp. 132–145, 2020.

[6] S. Ballouz, A. Dobin, and J. A. Gillis, "Is it time to change the reference genome?" *Genome biology*, vol. 20, no. 1, pp. 1–9, 2019.

[7] W.-W. Liao, M. Asri, J. Ebler, D. Doerr, M. Haukness, G. Hickey, S. Lu, J. K. Lucas, J. Monlong, H. J. Abel, , S. Buonaiuto, X. H. Chang, H. Cheng, J. Chu, V. Colonna, J. M. Eizenga, X. Feng, C. Fischer, R. S. Fulton, S. Garg, C. Groza, A. Guarracino, W. T. Harvey, S. Heumos, K. Howe, M. Jain, T.-Y. Lu, C. Markello, F. J. Martin, M. W. Mitchell, K. M. Munson, M. N. Mwaniki, A. M. Novak, H. E. Olsen, T. Pesout, D. Porubsky, P. Prins, J. A. Sibbesen, J. Sirén, C. Tomlinson, F. Villani, M. R. Vollger, L. L. Antonacci-Fulton, G. Baid, C. A. Baker, A. Belyaeva, K. Billis, A. Carroll, P.-C. Chang, S. Cody, D. E. Cook, R. M. Cook-Deegan, O. E. Cornejo, M. Diekhans, P. Ebert, S. Fairley, O. Fedrigo, A. L. Felsenfeld, G. Formenti, A. Frankish, Y. Gao, N. A. Garrison, C. G. Giron, R. E. Green, L. Haggerty, K. Hoekzema, T. Hourlier, H. P. Ji, E. E. Kenny, B. A. Koenig, A. Kolesnikov, J. O. Korbel, J. Kordosky, S. Koren, H. Lee, A. P. Lewis, H. Magalhães, S. Marco-Sola, P. Marijon, A. McCartney, J. McDaniel, J. Mountcastle, M. Nattestad, S. Nurk, N. D. Olson, A. B. Popejoy, D. Puiu, M. Rautiainen, A. A. Regier, A. Rhie, S. Sacco, A. D. Sanders, V. A. Schneider, B. I. Schultz, K. Shafin, M. W. Smith, H. J. Sofia, A. N. Abou Tayoun, F. Thibaud-Nissen, F. F. Tricomi, J. Wagner, B. Walenz, J. M. D. Wood, A. V. Zimin, G. Bourque, M. J. P. Chaisson, P. Flicek, A. M. Phillippy, J. M. Zook, E. E. Eichler, D. Haussler, T. Wang, E. D. Jarvis, K. H. Miga, E. Garrison, T. Marschall, I. M. Hall, H. Li, and B. Paten, "A draft human pangenome reference," *Nature*, vol. 617, no. 7960, pp. 312–324, 2023.

[8] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt, J. D. Gocayne, P. Amanatides, R. M. Ballew, D. H. Huson, J. R. Wortman, Q. Zhang, C. D. Kodira, X. H. Zheng, L. Chen, M. Skupski, G. Subramanian, P. D. Thomas, J. Zhang, G. L. G. Miklos, C. Nelson, S. Broder, A. G. Clark, J. Nadeau, V. A. McKusick, N. Zinder, A. J. Levine, R. J. Roberts, M. Simon, C. Slayman, M. Hunkapiller, R. Bolanos, A. Delcher, I. Dew, D. Fasulo, M. Flanigan, L. Florea, A. Halpern, S. Hannenhalli, S. Kravitz, S. Levy, C. Mobarry, K. Reinert, K. Remington, J. Abu-Threideh, E. Beasley, K. Biddick, V. Bonazzi, R. Brandon, M. Cargill, I. Chandramouliswaran, R. Charlab, K. Chaturvedi, Z. Deng, V. D. Francesco, P. Dunn, K. Eilbeck, C. Evangelista, A. E. Gabrielian, W. Gan, W. Ge, F. Gong, Z. Gu, P. Guan, T. J. Heiman, M. E. Higgins, R.-R. Ji, Z. Ke, K. A. Ketchum, Z. Lai, Y. Lei, Z. Li, J. Li, Y. Liang, X. Lin, F. Lu, G. V. Merkulov, N. Milshina, H. M. Moore, A. K. Naik, V. A. Narayan, B. Neelam, D. Nusskern, D. B. Rusch, S. Salzberg, W. Shao, B. Shue, J. Sun, Z. Y. Wang, A. Wang, X. Wang, J. Wang, M.-H. Wei, R. Wides, C. Xiao, C. Yan, A. Yao, J. Ye, M. Zhan, W. Zhang, H. Zhang, Q. Zhao, L. Zheng, F. Zhong, W. Zhong, S. C. Zhu, S. Zhao, D. Gilbert, S. Baumhueter, G. Spier, C. Carter, A. Cravchik, T. Woodage, F. Ali, H. An, A. Awe, D. Baldwin, H. Baden, M. Barnstead, I. Barrow, K. Beeson, D. Busam, A. Carver, A. Center, M. L. Cheng, L. Curry, S. Danaher, L. Davenport, R. Desilets, S. Dietz, K. Dodson, L. Doup, S. Ferriera, N. Garg, A. Gluecksmann, B. Hart, J. Haynes, C. Haynes, C. Heiner, S. Hladun, D. Hostin, J. Houck, T. Howland, C. Ibegwam, J. Johnson, F. Kalush, L. Kline, S. Koduru, A. Love, F. Mann, D. May, S. McCawley, T. McIntosh, I. McMullen, M. Moy, L. Moy, B. Murphy, K. Nelson, C. Pfannkoch, E. Pratts, V. Puri, H. Qureshi, M. Reardon,

R. Rodriguez, Y.-H. Rogers, D. Romblad, B. Ruhfel, R. Scott, C. Sitter, M. Smallwood, E. Stewart, R. Strong, E. Suh, R. Thomas, N. N. Tint, S. Tse, C. Vech, G. Wang, J. Wetter, S. Williams, M. Williams, S. Windsor, E. Winn-Deen, K. Wolfe, J. Zaveri, K. Zaveri, J. F. Abril, R. Guigó, M. J. Campbell, K. V. Sjolander, B. Karlak, A. Kejariwal, H. Mi, B. Lazareva, T. Hatton, A. Narechania, K. Diemer, A. Muruganujan, N. Guo, S. Sato, V. Bafna, S. Istrail, R. Lippert, R. Schwartz, B. Walenz, S. Yooseph, D. Allen, A. Basu, J. Baxendale, L. Blick, M. Caminha, J. Carnes-Stine, P. Caulk, Y.-H. Chiang, M. Coyne, C. Dahlke, A. D. Mays, M. Dombroski, M. Donnelly, D. Ely, S. Esparham, C. Fosler, H. Gire, S. Glanowski, K. Glasser, A. Glodek, M. Gorokhov, K. Graham, B. Gropman, M. Harris, J. Heil, S. Henderson, J. Hoover, D. Jennings, C. Jordan, J. Jordan, J. Kasha, L. Kagan, C. Kraft, A. Levitsky, M. Lewis, X. Liu, J. Lopez, D. Ma, W. Majoros, J. McDaniel, S. Murphy, M. Newman, T. Nguyen, N. Nguyen, M. Nodell, S. Pan, J. Peck, M. Peterson, W. Rowe, R. Sanders, J. Scott, M. Simpson, T. Smith, A. Sprague, T. Stockwell, R. Turner, E. Venter, M. Wang, M. Wen, D. Wu, M. Wu, A. Xia, A. Zandieh, and X. Zhu, "The sequence of the human genome," *science*, vol. 291, no. 5507, pp. 1304–1351, 2001.

[9] T. Wang, L. Antonacci-Fulton, K. Howe, H. A. Lawson, J. K. Lucas, A. M. Phillippy, A. B. Popejoy, M. Asri, C. Carson, M. J. Chaisson, , X. Chang, R. Cook-Deegan, A. L. Felsenfeld, R. S. Fulton, E. P. Garrison, N. A. Garrison, T. A. Graves-Lindsay, H. Ji, E. E. Kenny, B. A. Koenig, D. Li, T. Marschall, J. F. McMichael, A. M. Novak, D. Purushotham, V. A. Schneider, B. I. Schultz, M. W. Smith, H. J. Sofia, T. Weissman, P. Flicek, H. Li, K. H. Miga, B. Paten, E. D. Jarvis, I. M. Hall, E. E. Eichler, D. Haussler, and the Human Pangenome Reference Consortium, "The human pangenome project: a global resource to map genomic diversity," *Nature*, vol. 604, no. 7906, pp. 437–446, 2022.

[10] M. Eisenstein, "Every base everywhere all at once: Pangenomics comes of age," *Nature*, vol. 616, no. 7957, pp. 618–620, 2023.

[11] C. Groza, C. Schwendinger-Schreck, W. A. Cheung, E. G. Farrow, I. Thiffault, J. Lake, W. B. Rizzo, G. Evrony, T. Curran, G. Bourque, and T. Pastinen, "Pangenome graphs improve the analysis of rare genetic diseases," *medRxiv*, pp. 2023–05, 2023.

[12] Z. Yang, A. Guarracino, P. J. Biggs, M. A. Black, N. Ismail, J. R. Wold, T. R. Merriman, P. Prins, E. Garrison, and J. de Ligt, "Pangenome graphs in infectious disease: a comprehensive genetic variation analysis of neisseria meningitidis leveraging oxford nanopore long reads," *Frontiers in Genetics*, vol. 14, 2023.

[13] A. Guarracino, S. Buonaiuto, L. G. de Lima, T. Potapova, A. Rhie, S. Koren, B. Rubinstein, C. Fischer, H. J. Abel, L. L. Antonacci-Fulton, M. Asri, G. Baid, C. A. Baker, A. Belyaeva, K. Billis, G. Bourque, A. Carroll, M. J. P. Chaisson, P.-C. Chang, X. H. Chang, H. Cheng, J. Chu, S. Cody, D. E. Cook, R. M. Cook-Deegan, O. E. Cornejo, M. Diekhans, D. Doerr, P. Ebert, J. Ebler, E. E. Eichler, J. M. Eizenga, S. Fairley, O. Fedrigo, A. L. Felsenfeld, X. Feng, P. Flicek, G. Formenti, A. Frankish, R. S. Fulton, Y. Gao, S. Garg, N. A. Garrison, C. G. Giron, R. E. Green, C. Groza, L. Haggerty, I. Hall, W. T. Harvey, M. Haukness, D. Haussler, S. Heumos, G. Hickey, K. Hoekzema, T. Hourlier, K. Howe, M. Jain, E. D. Jarvis, H. P. Ji, E. E. Kenny, B. A. Koenig, A. Kolesnikov, J. O. Korbel, J. Kordosky, H. Lee, A. P. Lewis, H. Li, W.-W. Liao, S. Lu, T.-Y. Lu, J. K. Lucas, H. Magalhães, S. Marco-Sola, P. Marijon, C. Markello, T. Marschall, F. J. Martin, A. McCartney, J. McDaniel, K. H. Miga, M. W. Mitchell, J. Monlong, J. Mountcastle, K. M. Munson, M. N. Mwaniki, M. Nattestad, A. M. Novak, S. Nurk, H. E. Olsen, N. D. Olson, B. Paten, T. Pesout, A. B. Popejoy, D. Porubsky, P. Prins, D. Puiu, M. Rautiainen, A. A. Regier, S. Sacco, A. D. Sanders, V. A. Schneider, B. I. Schultz, K. Shafin, J. A. Sibbesen, J. Sirén, M. W. Smith, H. J. Sofia, A. N. A. Tayoun, F. Thibaud-Nissen, C. Tomlinson, F. F. Tricomi, F. Villani, M. R. Vollger, J. Wagner, B. Walenz, T. Wang, J. M. D. Wood, A. V. Zimin, J. M. Zook, J. L. Gerton, A. M. Phillippy, V. Colonna, and E. Garrison, "Recombination between heterologous human acrocentric chromosomes," *Nature*, vol. 617, no. 7960, pp. 335–343, May 2023. [Online]. Available: https://doi.org/10.1038/s41586-023-05976-y

[14] S. Hübner, "Are we there yet? driving the road to evolutionary graph-pangenomics," *Current Opinion in Plant Biology*, vol. 66, p. 102195, 2022.

[15] J. M. Eizenga, A. M. Novak, E. Kobayashi, F. Villani, C. Cisar, S. Heumos, G. Hickey, V. Colonna, B. Paten, and E. Garrison, "Efficient dynamic variation graphs," *Bioinformatics*, vol. 36, no. 21, pp. 5139–5144, 2020.

[16] T. Shiina, K. Hosomichi, H. Inoko, and J. K. Kulski, "The hla genomic loci map: expression, interaction, diversity and disease," *Journal of human genetics*, vol. 54, no. 1, pp. 15–39, 2009.

[17] D. J. Langton, S. C. Bourke, B. A. Lie, G. Reiff, S. Natu, R. Darlay, J. Burn, and C. Echevarria, "The influence of hla genotype on the severity of covid-19 infection," *Hla*, vol. 98, no. 1, pp. 14–22, 2021.

[18] A. Mikheenko and M. Kolmogorov, "Assembly graph browser: interactive visualization of assembly graphs," *Bioinformatics*, vol. 35, no. 18, pp. 3476–3478, 2019.

[19] R. R. Wick, M. B. Schultz, J. Zobel, and K. E. Holt, "Bandage: interactive visualization of de novo genome assemblies," *Bioinformatics*, vol. 31, no. 20, pp. 3350–3352, 2015.

[20] S. Heumos, A. Guarracino, J.-N. M. Schmelzle, J. Li, Z. Zhang, J. Hagmann, S. Nahnsen, P. Prins, and E. Garrison, "Pangenome graph layout by Path-Guided Stochastic Gradient Descent," *Bioinformatics*, vol. 40, no. 7, p. btae363, 07 2024. [Online]. Available: https://doi.org/10.1093/bioinformatics/btae363

[21] E. Garrison, J. Sirén, A. M. Novak, G. Hickey, J. M. Eizenga, E. T. Dawson, W. Jones, S. Garg, C. Markello, M. F. Lin, B. Paten, and R. Durbin, "Variation graph toolkit improves read mapping by representing genetic variation in the reference," *Nature biotechnology*, vol. 36, no. 9, pp. 875–879, 2018.

[22] G. Gonnella, N. Niehus, and S. Kurtz, "Gfaviz: flexible and interactive visualization of gfa sequence graphs," *Bioinformatics*, vol. 35, no. 16, pp. 2853–2855, 2019.

[23] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: an open source software for exploring and manipulating networks," in *Proceedings of the international AAAI conference on web and social media*, vol. 3, no. 1, 2009, pp. 361–362.

[24] J. X. Zheng, S. Pawar, and D. F. Goodman, "Graph drawing by stochastic gradient descent," *IEEE transactions on visualization and computer graphics*, vol. 25, no. 9, pp. 2738–2748, 2018.

[25] T. M. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software: Practice and experience*, vol. 21, no. 11, pp. 1129–1164, 1991.

[26] M. Jacomy, T. Venturini, S. Heymann, and M. Bastian, "Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software," *PloS one*, vol. 9, no. 6, p. e98679, 2014.

[27] Y. Hu, "Efficient, high-quality force-directed graph drawing," *Mathematica journal*, vol. 10, no. 1, pp. 37–71, 2005.

[28] A. Guarracino, S. Heumos, S. Nahnsen, P. Prins, and E. Garrison, "Odgi: understanding pangenome graphs," *Bioinformatics*, vol. 38, no. 13, pp. 3319–3326, 2022.

[29] E. Garrison, A. Guarracino, S. Heumos, F. Villani, Z. Bao, L. Tattini, J. Hagmann, S. Vorbrugg, S. Marco-Sola, C. Kubica, D. G. Ashbrook, K. Thorell, R. L. Rusholme-Pilcher, G. Liti, E. Rudbeck, S. Nahnsen, Z. Yang, M. N. Moses, F. L. Nobrega, Y. Wu, H. Chen, J. de Ligt, P. H. Sudmant, N. Soranzo, V. Colonna, R. W. Williams, and P. Prins, "Building pangenome graphs," *bioRxiv*, pp. 2023–04, 2023.

[30] L. Community, "Perf: Linux profiling with performance counters," https://github.com/torvalds/linux/tree/master/tools/perf, accessed: 2023-09-17.

[31] Intel, "Intel vtune profiler," https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html, accessed: 2023-09-17.

[32] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *Advances in neural information processing systems*, vol. 24, 2011.

[33] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 35–44.

[34] D. Blackman and S. Vigna, "Scrambled linear pseudorandom number generators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 47, no. 4, pp. 1–32, 2021.

[35] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[36] NVIDIA, "Nvidia nsight systems," https://developer.nvidia.com/nsight-systems, accessed: 2023-09-17.

[37] ——, "Nvidia curand library," https://developer.nvidia.com/curand, accessed: 2023-09-17.

[38] G. Marsaglia, "Xorshift rngs," *Journal of Statistical Software*, vol. 8, no. 14, p. 1–6, 2003. [Online]. Available: https://www.jstatsoft.org/index.php/jss/article/view/v008i14

[39] H. Purchase, "Which aesthetic has the greatest effect on human understanding?" in *International Symposium on Graph Drawing*. Springer, 1997, pp. 248–261.

[40] H. Gibson, J. Faith, and P. Vickers, "A survey of two-dimensional graph layout techniques for information visualisation," *Information visualization*, vol. 12, no. 3-4, pp. 324–357, 2013.

[41] H. Haleem, Y. Wang, A. Puri, S. Wadhwa, and H. Qu, "Evaluating the readability of force directed graph layouts: A deep learning approach," *IEEE computer graphics and applications*, vol. 39, no. 4, pp. 40–53, 2019.

[42] T. Dwyer, B. Lee, D. Fisher, K. I. Quinn, P. Isenberg, G. Robertson, and C. North, "A comparison of user-generated and automatic graph layouts," *IEEE transactions on visualization and computer graphics*, vol. 15, no. 6, pp. 961–968, 2009.

[43] J. Blythe, C. McGrath, and D. Krackhardt, "The effect of graph layout on inference from social network data," in *International symposium on graph drawing*. Springer, 1995, pp. 40–51.

[44] T. Kamada, S. Kawai *et al.*, "An algorithm for drawing general undirected graphs," *Information processing letters*, vol. 31, no. 1, pp. 7–15, 1989.

[45] E. R. Gansner, Y. Koren, and S. North, "Graph drawing by stress majorization," in *Graph Drawing: 12th International Symposium, GD 2004, New York, NY, USA, September 29-October 2, 2004, Revised Selected Papers 12*. Springer, 2005, pp. 239–250.

[46] L. Le Cam, "The central limit theorem around 1935," *Statistical science*, pp. 78–91, 1986.

[47] S. G. Kwak and J. H. Kim, "Central limit theorem: the cornerstone of modern statistics," *Korean journal of anesthesiology*, vol. 70, no. 2, p. 144, 2017.

[48] NVIDIA, "Nvidia nsight compute," https://developer.nvidia.com/nsight-compute, accessed: 2023-09-17.

[49] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[50] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz—open source graph drawing tools," in *Graph Drawing: 9th International Symposium, GD 2001 Vienna, Austria, September 23–26, 2001 Revised Papers 9*. Springer, 2002, pp. 483–484.

[51] G. G. Brinkmann, K. F. Rietveld, and F. W. Takes, "Exploiting gpus for fast force-directed visualization of large-scale networks," in *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 2017, pp. 382–391.

[52] A. Fender, B. Rees, and J. Eaton, "Rapids cugraph," in *Massive Graph Analytics*. Chapman and Hall/CRC, 2022, pp. 483–493.

[53] M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein, and P. Mutzel, "The open graph drawing framework (ogdf)." *Handbook of graph drawing and visualization*, vol. 2011, pp. 543–569, 2013.

[54] O. Kunyavskaya and A. D. Prjibelski, "Sgtk: a toolkit for visualization and assessment of scaffold graphs," *Bioinformatics*, vol. 35, no. 13, pp. 2303–2305, 2019.

[55] M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, and G. D. Bader, "Cytoscape. js: a graph theory library for visualisation and analysis," *Bioinformatics*, vol. 32, no. 2, pp. 309–311, 2016.

[56] E. F. de Oliveira Sandes, G. Miranda, X. Martorell, E. Ayguade, G. Teodoro, and A. C. M. Melo, "Cudalign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in gpu clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2838–2850, 2016.

[57] S. D. Goenka, Y. Turakhia, B. Paten, and M. Horowitz, "Segalign: A scalable gpu-based whole genome aligner," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–13.

[58] K. Zhao and X. Chu, "G-blastn: accelerating nucleotide alignment by graphics processors," *Bioinformatics*, vol. 30, no. 10, pp. 1384–1391, 2014.

[59] A. Zeni, G. Guidi, M. Ellis, N. Ding, M. D. Santambrogio, S. Hofmeyr, A. Buluç, L. Oliker, and K. Yelick, "Logan: High-performance gpu-based x-drop long-read alignment," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 462–471.

[60] A. Müller, B. Schmidt, R. Membarth, R. Leißa, and S. Hack, "Anyseq/gpu: a novel approach for faster sequence alignment on gpus," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–11.

[61] M. G. Awan, S. Hofmeyr, R. Egan, N. Ding, A. Buluc, J. Deslippe, L. Oliker, and K. Yelick, "Accelerating large scale de novo metagenome assembly using gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–11.

[62] R. Kobus, A. Müller, D. Jünger, C. Hundt, and B. Schmidt, "Metacache-gpu: ultra-fast metagenomic classification," in *Proceedings of the 50th International Conference on Parallel Processing*, 2021, pp. 1–11.

[63] L. Guo, J. Lau, Z. Ruan, P. Wei, and J. Cong, "Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 127–135.

[64] D. S. Cali, K. Kanellopoulos, J. Lindegger, Z. Bingöl, G. S. Kalsi, Z. Zuo, C. Firtina, M. B. Cavlak, J. Kim, N. M. Ghiasi *et al.*, "Segram: A universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 638–655.