Appears in the Proceedings of the 2025 IEEE Int'l Symp. on Workload Characterization (IISWC), October 2025

PangenomicsBench: A Benchmark Suite and Characterization of Pangenomics

Noah Kaplan*, Jan-Niklas Schmelzle†, Yufeng Gu*, Erik Garrison‡, Christopher Batten†, Reetuparna Das*
*University of Michigan
{kaplannp,yufenggu,reetudas}@umich.edu
†Cornell University
{jms854,cbatten}@cornell.edu
‡University of Tennessee Health Science Center
egarris5@uthsc.edu

Abstract

Cheaper and more accurate sequencing technologies have led to a large volume of genetic data that poses significant computational challenges and requires novel computing solutions to keep pace. This increased volume has also enabled the use of pangenome graph references, which provide better quality alignments because they represent variation, but they require new algorithms that are usually slower than those using a traditional reference genome, and exhibit different computational characteristics.

We introduce PangenomicsBench, the first benchmark suite targeting computational pangenomics, with six CPU and two GPU kernels extracted from popular tools, designed to guide future research in pangenomics software and hardware acceleration. We characterize these workloads to reveal the following key insights: (a) Seq2Graph mapping algorithms are limited by control complexity rather than memory access to the reference graph because they process small, cache-friendly subgraphs. (b) GPUs have the potential for large speedups, but are limited by control divergence for mapping workloads. (c) Pangenomics introduces computational patterns different from traditional genomics like stochastic gradient descent. (d) Pangenomic mapping algorithms are highly sensitive to reference graph structures. (e) There are opportunities for optimizing existing software.

1. Introduction

Genome sequencing is a key component of precision health, allowing us to tailor treatment to individuals based on their genetics. Decreased sequencing costs have resulted in a large volume of sequencing data and assembled genomes, which have given rise to the new subfield of *computational pangenomics*.

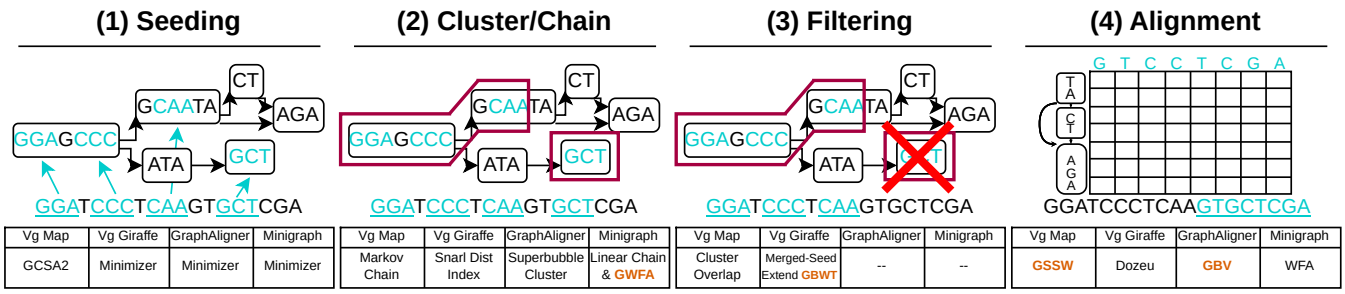
A traditional reference genome is represented as a sequence of base pairs (A, T, C, and G); a *pan*genome contains many genomes, often represented as a *sequence graph* with directed edges. Each node contains a subsequence of base pairs, and each path represents a sequence (Figure 1.1).

Most genomes today are sequenced with a single reference, but reference genomes can miss up to 10% of the basepairs in under-represented populations [1] leading to decreased sequencing accuracy and missed variants. For example, recent work suggests pangenomics could have been used to more efficiently discover a critical genetic variant associated with increased heart attack risk [1]. Pangenomes can be used to make comparisons within a population, draw out new biological insights [2], and provide more accurate alignments [3].

Compared to traditional genomics, Pangenomics poses distinct computational challenges due to its reliance on graph representations. The same pangenome can be modeled differently with distinct graphs, each offering unique insights. These graphs demand more computational resources because they are larger and more complex than sequences. For example, standard sequence-to-sequence mapping techniques, like coordinate-based distance estimation, are ineffective for graphs which require path finding. Common dynamic programming algorithms have to incorporate graph nodes and edges. Building and visualizing pangenomic graphs also introduce uncommon computational patterns not typically seen in genomics. These complexities make software development and optimization more demanding, and introduce new bottlenecks like low SIMD utilization, low GPU occupancy, and, high branch misprediction rate.

Previous work has characterized the behavior of genomics kernels, and curated benchmarks to guide hardware and software performance optimization [4–8], but pangenomics requires new benchmarks to guide hardware and software optimization and evaluate new software. Building such a benchmark suite requires comprehensive understanding of pangenomics workflows and their usage in the field including tools, configurations, and representative datasets. Identifying bottlenecks and sourcing appropriately-sized genomic data also requires considerable effort.

In this paper, we examine two pangenomic workflows: the *Seq2Graph mapping pipeline*, which significantly changes compute patterns to enable mapping to pangenome graphs, and the *graph-building pipeline*, a new workflow specific to pangenomics. We evaluate these workflows because they are time consuming, distinct from traditional genomics, and common to most pangenomics tasks. Other downstream pangenomics analyses like variant calling and genome wide association studies (GWAS) depend on graph building and Seq2Graph mapping as preliminary steps.



Vg Map	Vg Giraffe	GraphAligner	Minigraph
GCSA2	Minimizer	Minimizer	Minimizer

Vg Map	Vg Giraffe	GraphAligner	Minigraph
Markov Chain	Snarl Dist Index	Superbubble Cluster	Linear Chain & GWFA

Vg Map	Vg Giraffe	GraphAligner	Minigraph
Cluster Overlap	Merged-Seed Extend GBWT	--	--

Vg Map	Vg Giraffe	GraphAligner	Minigraph
GSSW	Dozeu	GBV	WFA

Figure 1: Sequence To Graph Mapping Pipeline.

We analyze four tools—GraphAligner [9], Vg Map [10], Vg Giraffe [11], and Minigraph [12]—that implement the end-to-end Seq2Graph mapping pipeline, along with the Minigraph-Cactus [13], and PGGB suites [14], which are used for graph construction. These tools can take days to process real-world datasets and are mainly used in an iterative manner. Computational biologists use the tools to produce initial alignments or pangenome graphs, analyze the results, and then rerun the tools to produce more biologically useful output. For benchmarking, we extract and characterize computationally intensive regions of code (*kernels*) from each workload.

In summary, this paper makes the following contributions:

- We examine five widely adopted pangenomics tools from two key workflows in pangenomics: Seq2Graph mapping and graph construction.
- We identify eight computationally intensive and representative kernels from these workflows to develop PangenomicsBench, the first open-source benchmark suite designed for pangenomics.
- We carry out a timing and thread-scaling analysis of the workflows and perform microarchitectural and source code-level analyses on the extracted kernels. Furthermore, we present case studies comparing Seq2Seq with Seq2Graph mapping and exploring the impact of reference graph variation.
- Our characterization yields several insights: (a) We find Seq2Graph algorithms have good memory locality because they operate on smaller, cache-friendly local subgraphs, but their performance is limited by the control complexity required to support graph references. (b) GPUs can show large speedups, but performance varies by workload, and is limited by control divergence for mapping workloads. (c) We observe novel pangenomics computational patterns, such as PGSGD, distinct from traditional genomics. (d) Pangenomic mapping algorithms are sensitive to the reference graph structure. (e) Our analysis indicates potential opportunities for software optimization of existing code.

2. Pangenomics Pipelines

In this section, we describe two common pangenomics pipelines, Sequence to Graph Mapping and Graph Building. We select six tools from these pipelines and analyze the time spent in each workflow.

2.1. Sequence to Graph Mapping

Sequence to Graph (Seq2Graph) Mapping follows the same algorithmic steps as Sequence to Sequence Mapping (Seq2Seq) as illustrated in Figure 1. First, a fast heuristic is used to find exact matches, **seeds**, between the query sequence (usually a read) and the reference graph. Next, seeds are grouped into **clusters** and **chains** by locality in the graph and query. Some algorithms also implement a **filtering** step to reduce the number of clusters/chains sent to alignment. Lastly, seed hits are **aligned** to the reference using dynamic programming algorithms. More aggressive pruning in the first three steps results in less work during alignment.

Extending the steps used in Seq2Seq mapping to work with a graph reference introduces new design trade-offs, increases complexity, adds dependencies, and causes control divergence. For instance, Seq2Seq clustering and chaining techniques typically calculate distance between seeds as the difference between coordinate locations on the reference. In graph mapping, however, this distance is the shortest path length, which is complicated by the presence of cycles and requires graph traversal or memoization in large data structures. Additionally, graph references introduce new dependencies in alignment, as discussed in Section 3. These complexities make Seq2Graph algorithms slow.

To estimate the runtime to assemble a full human genome, we time four Seq2Graph tools and one Seq2Seq tool using the datasets in Section 4.2, and scale by the number of reads needed for 30x coverage, Table 1. Figure 2 shows a further breakdown of time spent in the seeding, clustering/chaining, filtering, and alignment steps for the Seq2Graph tools. **Vg Giraffe** and **Vg Map** are characterized for short reads while **GraphAligner** and **Minigraph** are characterized for long reads (**Minigraph-lr**). We also characterize **Minigraph** for the mapping of assemblies to graphs (**Minigraph-cr**) as it is used in graph building. Dataset details are found in section 4.2. We discuss a few observations below.

Most of the Seq2Graph algorithms we review use minimizer based seeding, which does the same computation as Seq2Seq minimizers, but with larger memory requirements.

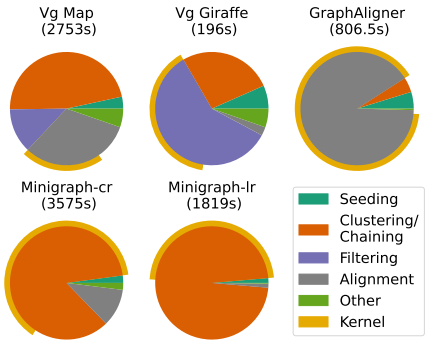


Figure 2: Seq2Graph Timing Breakdown Extracted With Vtune: Yellow arcs show kernel mapping step and fraction of runtime. Absolute tool timing is shown beside the toolname. These times should be compared with caution as the tools operate on different data (see Sec 4.2).

Seq2Graph				Seq2Seq
VG Map	VG Giraffe	GraphAligner	Minigraph-lr	BWA-MEM2
67.1h	4.8h	9.1h	20.5h	1.3h

TABLE 1: Estimated Full Genome Assembly Runtime

As a result, subsequent steps that become more complex in Seq2Graph mapping dominate the runtime.

In the clustering/chaining and filtering steps, some tools employ advanced heuristics to minimize the computational burden on downstream alignment. *Minigraph*, designed for long sequences where alignment is particularly slow, thus uses 2D dynamic programming within the chaining stage to reduce the number of downstream alignments. From this stage, we extract the Graph Wavefront Alignment (`GWFA`) kernel, which accounts for 75% of the total clustering/chaining step when aligning chromosomal assemblies, and 47% when aligning long reads. The difference is caused by different default parameters and the alignment of longer sequences for chromosomes. In Figure 2, the run time of the kernels (e.g., GWFA), within their respective tools (e.g., Minigraph) is represented by the yellow arc.

Vg Giraffe also features a sophisticated and time-consuming filtering step, which performs light-weight seed extension through the graph using operations on the Graph Burrows-Wheeler Transform (GBWT) Index. For our benchmark suite, we select a single representative operation as the `GBWT` kernel.

In contrast, *GraphAligner* uses lightweight clustering (5% of runtime), and spends the remaining 90% on alignment. To make this feasible for long reads, GraphAligner uses aggressive heuristics (non affine-gap scoring) in the alignment step which trade accuracy to make performance manageable. We extract the graph bitvector (GBV) kernel from the alignment stage of GraphAligner for our benchmark suite.

Vg Map falls between these extremes, with significant time distributed across all stages. From the alignment stage, we extract the Graph SIMD Smith-Waterman (`GSSW`) kernel, as it is dominated by a single hot function. In contrast, the clustering and chaining stages are fragmented across numerous shorter function calls.

In summary, the mapping tools we analyzed make distinct trade-offs between accuracy and performance across various stages. While we might expect similar tools to allocate comparable proportions of time to each stage, we instead observe that different tools prioritize different stages. No single step or computational pattern stands out as a clear bottleneck, suggesting that all stages could benefit from optimization and acceleration.

2.2. Graph Building

There are two popular graph-building pipelines in pangenomics research. Both are commonly used and have different merits. While *Minigraph-Cactus's* (MC) [13] run time scales linearly with the number of input genomes, the *PanGenome Graph Builder* (PGGB) [14] prioritizes eliminating reference bias. Both pipelines begin with a collection of sequences and construct a graph in four computational stages (see Figure 3): (1) *Alignment* identifies matching sub-sequences within the input reference sequences; (2) *Graph induction* constructs a pangenome graph from the matches; (3) *Polishing* refines the pangenome graph, for instance, by filtering out short variations; (4) *Visualization* constructs a 2D representation of the graph for evaluation. These four steps are part of an interactive, iterative, and time-consuming process where scientists build the graph, evaluate its 2D representation, manually update the graph-building parameters, and repeat. The run times in Figure 3 are collected for 14 chromosome 20 assemblies, but the HPRC [15] graph contains 47 diploid human genomes. We scaled runtime by the ratio of the HPRC dataset size to our dataset size to estimate it would take 2 weeks to build the HPRC graph on our server.

The *alignment* stage differs significantly between the two pipelines. MC iteratively grows a pangenome starting from a single reference using Minigraph Seq2Graph mapping which includes the `GWFA` kernel. This is shown in in Figure 3.1. The choice of this starting sequence affects the final quality, resulting in potential *reference bias*. PGGB removes reference bias by computing pairwise alignments for all combinations of input sequences. The all-to-all alignments are generated using wfmash [16], which combines the WaveFront Algorithm (WFA) [17] with MashMap [18]. We include `TSU` as a recent GPU implementation of WFA [19]. CPU versions have been included in prior benchmarks [7]. Although all-to-all alignment scales quadratically with the number of sequences, wfmash can be faster than MC for few inputs, depending on the configuration parameters such as identity threshold and alignment block size.

The *graph induction* stage constructs an (initial) pangenome graph using the previously created alignments. *Cactus*, from the MC pipeline, converts the previously generated minigraph alignments into a cactus graph. Here, performance is constrained by the Adaptive Banded Partial Order Alignment algorithm (abPOA) [20], a variation of which is found in previous benchmark suites [4]. *Seqwish* [21], the tool used by PGGB, utilizes wfmash's all-to-all alignments and performs a transclosure step on them, mapping matching nucleotides to pangenome graph nodes.

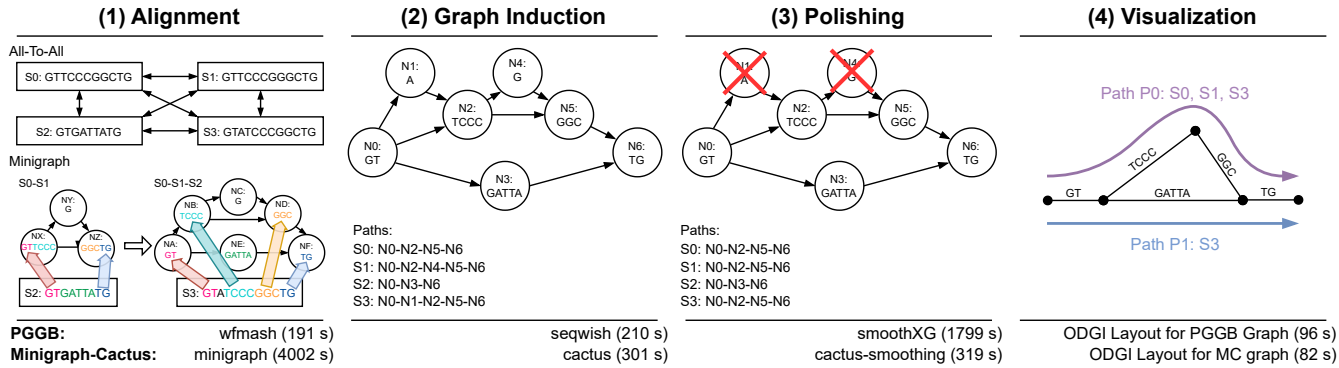


Figure 3: Pangenome Graph Building Pipeline: Run time was measured for both pipelines for processing a small pangenome consisting of 14 high-quality, whole-chromosome chromosome 20 assemblies (see Sec. 4.2).

Subsequently, seqwish compresses multiple non-branching nodes into a single node and inserts the corresponding paths through the graph. We extract the transclosure step as the `TC` kernel, as it accounts for more than 75% of the compute time.

The initial graphs contain complex local structures like poorly aligned regions which are removed in the *polishing* step. MC removes these and redundant paths using GFAffix [22] and other algorithms. SmoothXG [23], from the PGGB pipeline, similarly removes small cyclic regions within the graph. Its compute time is dominated by Partial Order Alignment (POA), which takes about 80% of run time. POA, a dynamic programming kernel, is part of prior benchmarks [4] and has been a target of hardware acceleration [24].

As modern pangenome graphs are large and complex datasets, researchers use *visualizations* to explore their inherent structure. ODGI [25], a software toolkit for exploring pangenome graphs, includes a subcommand to generate layouts for 2D visualizations of pangenome graphs in both pipelines. It employs the iterative Path-Guided Stochastic Gradient Descent (PGSGD) [26, 27] algorithm to compute the layout by defining it as an optimization problem. This step is dominated by the `PGSGD` kernel.

3. PangenomicsBench Benchmark Suite

We extract hot snippets of code from our six tools and release them as a *benchmark suite*. We discuss these kernels in depth here.

Graph SIMD Smith-Waterman (`GSSW`): Graph SIMD Smith-Waterman [28] is a dynamic programming algorithm used in Vg Map to map read fragments ($\approx$ 150 base pairs) to acyclic subgraphs extracted around seed hit locations from the pangenome reference graph.

It is based on the Seq2Seq algorithm, Smith-Waterman [29], which fills in an $n \times m$ dynamic programming (DP) matrix where m is the length of the query, and n is the length of a reference substring. Farrar's algorithm finds strip mined SIMD parallelism by speculating away dependencies between cells and recomputing mispeculated cells [30]. This is illustrated in Figure 4a for a word size of three, where cells of the same color are packed into a SIMD word.

`GSSW` uses SIMD Smith-Waterman, but instead of a single genome reference on the i-axis, it aligns to a topological sort of a subgraph of the reference with directed edges between nodes. Cells within the body of a node are computed with SIMD Smith-Waterman, but cells in the first row of a node depend on the node's parents, as shown in Figure 4a with red arrows. This node initialization is done separately, causing the algorithm to alternate between dense SIMD regions and indirect graph accesses.

`GSSW` vectorizes well, but it scales poorly with read length because it computes many cells in the DP matrix which do not contribute to the final solution.

Graph Myers's Bitvector (`GBV`): Graph Myers's Bitvector [31] is a dynamic programming algorithm used to align long reads in GraphAligner. As shown in Figure 2, GraphAligner exerts little effort filtering seed hits, which results in many alignments. This, and the presence of long reads create abundant work for the alignment stage. `GBV` manages this by trading alignment quality for computational efficiency.

Myers's bitvector [32], does not support the sophisticated affine gap scoring method used in Smith-Waterman which allows it to store rows as bitvectors. This removes cell dependencies, allowing rows to be processed in parallel with SIMD width equal to the word size of the machine, 64 by default.

In Graph Myers's bitvector, each row represents a one-base pair node with directed edges to its children, Figure 4b. This makes each row dependent on its parent rows (i.e. nodes), shown with red arrows in Figure 4b.

Since `GBV` aligns to cyclic graphs, the current row being computed, R_i, may change the value of its parent R_p. Thus, some rows will need to be recomputed until the score stabilizes. To keep track of the nodes that need to be recomputed, `GBV` pushes rows to a priority queue when their parent changes, and processes them in the queue order. These changes to the algorithm made to accommodate graphs introduce unpredictable branching behavior to `GBV`.

Graph Burrows-Wheeler Transform (`GBWT`): Vg Giraffe uses the GBWT Index [33] in the filtering step of the

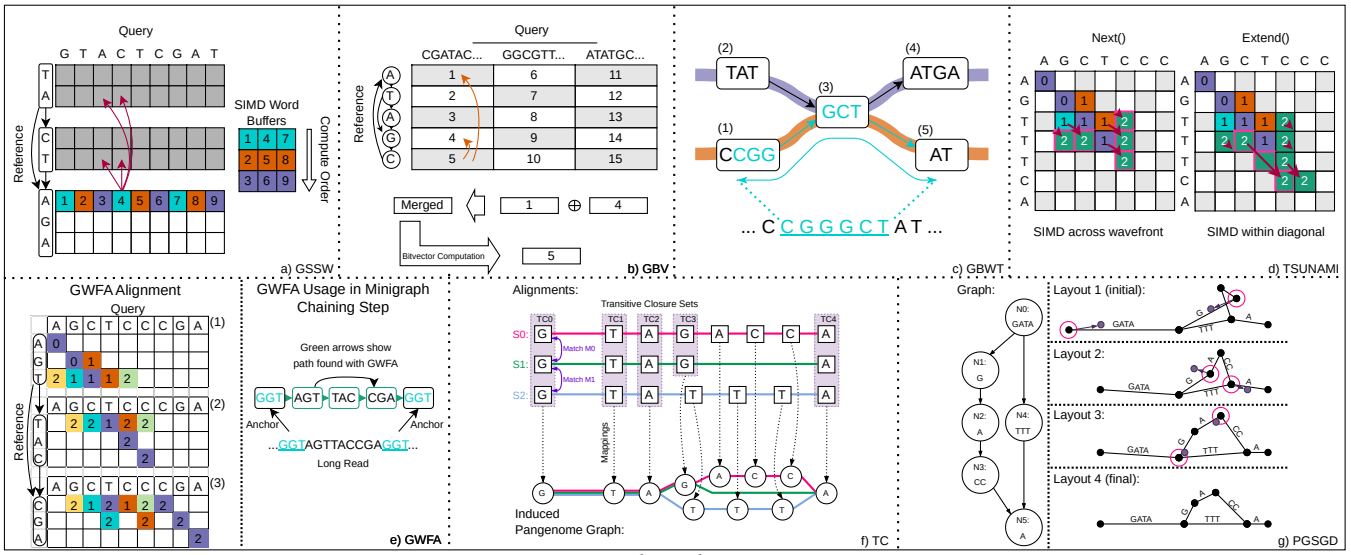


Figure 4: Kernel Explanatory Figures.

mapping pipeline, Figure 1.3, to extend clustered seed hits along graph paths. Many paths lead to unrealistic results, so we only follow paths that correspond to subsequences of a single reference sequence, i.e. haplotypes. For example, in Figure 4c there is a haplotype $1 \rightarrow 3 \rightarrow 5$, and another haplotype $2 \rightarrow 3 \rightarrow 4$, but since the seed exension has already gone through $1 \rightarrow 3$, it cannot go to 4 because there is no haplotype $1 \rightarrow 3 \rightarrow 4$.

More generally, this is accomplished with the GBWT Index function, *find* that takes a sequence of nodes, $S = (s_1, s_2, \ldots, s_n)$ as input and returns all possible next nodes. This function is representative of operations using GBWT Index, and so we extract it as a kernel, even though Vg Giraffe actually makes use of many GBWT Index functions.

The GBWT Index is a haplotype-aware graph FM-Index [34] that uses last-first mapping to look up nodes. It is built from a multi-string Burrows-Wheeler Transform (BWT) applied to *haplotype paths* through the graph. Unlike the FM-Index used in Seq2Seq mapping, which applies the BWT to a single string composed of base pairs, the GBWT Index is based on multiple sequences (haploytpe paths) of *node IDs* in the graph, which enables graph specific optimizations to the index.

Tsunami (TSU): Tsunami [19] (TSU) is a GPU-accelerated variant of the Wavefront Algorithm (WFA) [17] for Seq2Seq alignment. Minigraph uses WFA to improve mapping quality after applying GWFA, and PGGB's wfmash uses WFA to generate all-to-all alignments. The CPU-based WFA kernel used in Minigraph and wfmash is already included in a prior benchmark suite [7], so we include a more recent GPU variant, TSU [19].

In WFA, cells are computed along *diagonals* (purple, blue and orange strips in Figure 4d). WFA alternates between the *Next* step, where it pushes the diagonals one cell further to the green cells, and the *Extend* step where it pushes diagonals as far as possible along exact matches.

TSU allocates one 32-thread block to each alignment. In *Next* it assigns each diagonal to a thread, but in *Extend* diagonals will push different depths resulting in control divergence. E.g. in Figure 4d-right, the center diagonal doesn't extend at all, but the diagonal to its left extends two matches. In more realistic examples, this difference can be thousands of base pairs long. To improve warp utilization TSU speculates that the diagonal will have many matches, and each thread processes one cell in the diagonal (pink highlighted cells in Figure 4d-right).

Graph Wavefront Algorithm (GWFA): The Graph Wavefront Algorithm [35] is used in Minigraph to bridge gaps between seed hits (also called anchors) in different nodes. Given two anchors, it finds a path (sequence of nodes) connecting them, Figure 4e-right. This requires less accuracy than base-level alignment, so GWFA uses non-affine gap scoring.

GWFA is a direct extension of WFA. Each node has its own dynamic programming matrix with the query sequence on one axis, and the node on the other axis (Figure 4e). The matrices are connected by edges corresponding to the edges between nodes. When a diagonal reaches the end of a node, it expands the digaonals into each child node as shown by the blue diagonal in Figure 4e. This leads to more diagonals to process scattered across different nodes resulting in irregular accesses. Despite this, prior work shows GWFA is the fastest alignment algorithm reviewed [35] because it computes far fewer cells of the DP-Matrix.

Transclosure (TC): PGGB [14] utilizes Seqwish [21] to induce pangenome graphs from input sequences and their respective all-to-all alignments. Seqwish's run time is dominated by its transclosure (TC) kernel, which generates the set of pangenome graph nodes: each pairwise alignment from the all-to-all alignments specifies matching characters in their sequences. For instance, the match M0 between sequences S0 and S1 in Figure 4f creates the *Transitive*

Closure set TC0. These sets are extended via the transitive property. For example, M1 matches S1 to S2, so S2's character is added to TC0. To generate all closures, TC iterates over the set of input characters and locates their matches, as well as matches connected via the transitive property. Connected matches are found using a binary search on the sorted seqeunces. Each transitive closure is then mapped to a graph node.

TC introduces new, heterogenous compute patterns to the genomics space that leverage clever data structures, such as implicit interval trees [36], which are used for efficient union-find operations but require high-performance sorting steps [37] for efficient data access. These data structures require 100s of GBs of memory space, for real-world pangenomes. To enable memory-limited systems to perform the TC computation, TC memory-maps these structures to files. The TC kernel is slightly modified from seqwish's implementation to enable single threaded execution.

Path-Guided Stochastic Gradient Descent (PGSGD): ODGI [25] employs Path-Guided Stochastic Gradient Descent (PGSGD) [26, 27] to generate 2D layouts of pangenome graphs in the visualization step. PGSGD iteratively optimizes the layout to match the distances between nodes in the pangenome graph. Figure 4g demonstrates this. In the initial Layout 1, the line lengths poorly match the corresponding sequence lengths, and the pangenome is twisted. In each update step, two anchors within the layout are randomly selected, and their coordinates are adjusted to better match the corresponding path-distance in the pangenome graph (Figure 4g, left). In Layout 1, the left anchor of N0 and the anchor between N1 and N2 are chosen. Since they are too far apart, PGSGD moves them closer together, resulting in layout 2. After numerous update steps, the distances within the layout converge to path distances in the graph, as shown by Layout 4. As PGSGD contains millions of update steps, they are parallelized across threads using the lock-free Hogwild! approach [38]. In the rare case that a race condition occurs, future iterations will quickly correct the incorrect update. In this benchmark suite, we investigate both the CPU and GPU variants of the PGSGD kernel [27]. Like the CPU implementation, the GPU implementation utilizes the lock-free Hogwild! approach. Thus, each thread within a warp randomly picks a pair of anchors to update in parallel. To improve performance, the GPU kernel contains methods to reduce branch divergence within a warp and an optimized data layout for the random number generator states to enable coalesced memory accesses.

PGSGD introduces an entirely new task to the pangenomics workflow utilizing stochastic gradient descent. It also requires pseudo-random memory accesses to the graph and uses the Hogwild! parallelization method, which distinguishes it from other workloads.

4. Methodology

4.1. Procedure and Tools

We first use VTune [39] hotspot analysis and timers to identify hotspots within tools using all available threads. We then isolate *kernels* from the hotspots for our benchmark suite and perform three single threaded analyses for each kernel using VTune and PIN: **microarchitecture exploration** to understand hardware utilization, **cache miss rates** to find misses per kilo instruction, and a customized version of **MICA PIN** [40] [41] to get a dynamic instruction count. Additionally, we use NVIDIA Nsight Compute (NCU) to get GPU utilization statistics. The scripts for running these analyses, benchmarks, and datasets are open-sourced in our github repository. Lastly, we perform three case studies on selected kernels, described in section 6.

4.2. Datasets

Tool	# Seqs	Avg. Seq Len	Input Type	Size
Vg Map	7284888	149	Short Read	8.5GB
Vg Giraffe	7284888	149	Short Read	2.7GB
GraphAligner	158643	14950	Long Read	1.3GB
Minigraph-lr	158643	14950	Long Read	1.3GB
Minigraph-cr	1	67156117	Chromosome	2.3GB
Mini-Cactus	14	67939604	Chromosome	0.9GB
PGGB	14	67939604	Chromosome	0.9GB

TABLE 2: Tool Dataset Information For Chrom 20

Kernel	Parent Tool	# Inputs	Input Type	Size
GSSW	Vg Map	146105	Read Fragment	3.4GB
GBWT	Vg Giraffe	9986857	GBWT Query	6.0GB
GBV	GraphAligner	8240	Clusters	3.7GB
GWFA-lr	Minigraph	104205	Read Gaps	1.2GB
GWFA-chr	MC	291650	Chrom Gaps	1.2GB
TC	PGGB	32677	Alignments	0.9GB
Pgsgd	PGGB	588480	Pangenome	0.3GB
Tsunami	PGGB/MC	50000	10K long seqs	3.0GB

TABLE 3: Kernel Dataset Information For Chrom 20

GBV	GSSW	GBWT	GWFA-cr	GWFA-lr	PGSGD	TC
192s	35s	23s	16657s	720s	285s	755s

TABLE 4: Kernel Measured Execution Time (Machine B)

Table 2 shows the dataset we used for each tool. We used BWA-MEM2 and minimap2 [42] to filter chromosome 20 short and long reads respectively by aligning to HG002 [43] thus limiting analysis to chromosome 20. Seq2Graph tools map Illumina HiSeq short reads (150bp) and PacBio HiFi long reads (15000bp and 20000bp) from sample HG002 of HPRC to the Minigraph-Cactus constructed HPRC graph [15] [44]. BWA-MEM2 [45] in Table 1 maps the same short reads to HG002. Graph Building tools used 14 chromosome 20 assemblies including T2T CHM13 [43] and 13 from HPRC [46].

Table 3 shows kernel datasets. We generate GBWT inputs by randomly sampling subsequences from the haplotypes in the graph with lengths between 1 and 100. We use the full graph (not just chromosome 20) because GBWT cache behavior is especially sensitive to graph size. TSU runs on pairs of 10K base pair sequences with error rate 1% generated with the TSU script (see github). We also evaluate performance for different lengths. All other kernel datasets are produced by running the tool with datasets in table 2 up until the kernel and then storing the inputs to

the kernel in a file, sometimes downsampling to keep the dataset size reasonable. We include this code to generate new kernel datasets so researchers can analyze their own workloads. Note for TC and PGSGD, human data is too slow for PIN analysis, so we use Yeast [47] and DRB1-3123 [48] datasets for PIN analysis only. Table 4 shows the measured runtime for our kernels on these datasets using Machine B (see section 4.3). Increasing the size of datasets will increase these runtimes, but we expect the microarchitecture characteristics to remain the same.

4.3. Machine Configuration

	Machine A (2 sockets)	Machine B (2 sockets)
CPU	Intel Xeon E5-2697 v3	Intel Xeon Gold 6326
Base frequency	2.6GHz	2.9GHz
Core/thread count	14/28	16/32
L1 I cache	32KB, 8-way	32KB, 8-way
L1 D cache	32KB, 8-way	48KB, 12-way
L2 cache	256KB, 8-way	1.25MB, 20-way
L3 cache	35MB	24MB, 12-way
Memory	176GB DDR4 2400MHz	128GB DDR4 3200MHz
GPU	-	Nvidia RTX A6000
GPU-Mem	-	48GB GDDR6
GPU-Mem BW	-	768GB/s

TABLE 5: System configuration (core/thread count and L3 cache size are per socket).

We evaluated the tools and kernels using Machine A and Machine B from Table 5. Both machines are dual-socket systems. Machine A, with its larger memory capacity, was used to evaluate the tools with full datasets, while Machine B was used for the microarchitectural analysis of CPU and GPU kernels.

5. Results

In this section, we first evaluate the end-to-end thread-scaling capabilities of the selected tools. We then perform a detailed microarchitecture analysis of the extracted kernels. Finally, we evaluate the performance of the two GPU implementations.

5.1. Thread Scaling

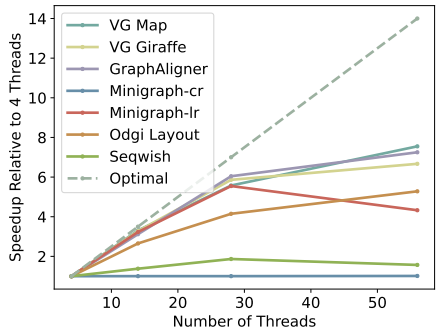


Figure 5: Thread Scaling: Runs executed on Machine A of Table 5 with hyperthreading for 4, 14, 28, and 56 threads. Results displayed as speedup relative to 4 threads.

Seq2Graph mapping tools process reads independently on different threads which results in good thread scaling up to 28 cores where hyperthreading reduces scaling potential (Figure 5). This performance drop is likely due to resource contention between threads because Seq2Graph kernels generally have high core utilization for a single thread.

We show Minigraph-lr operating on a single batch of 158K reads in Figure 5 because memory usage is low (below 60GB), and tail latency limits performance at smaller batches.

Minigraph-cr, in contrast, exausts the 176GB memory on our server (Machine A) for our downsized 14-genome dataset. We therefore evaluate for a single chromosome. Minigraph doesn't extract intra chromosome/read parallelism, so this only achieves single threaded execution. Even for large datasets and servers, we expect thread parallelism to be limited. HPRC for example uses 47 diploid human genomes and a multinode system consisting of 6 machines, each having 28 cores and 384GB of RAM [15].

PGSGD uses the Hogwild! parallelization approach [38], which enables near linear scaling with thread count. However, in Figure 5, the end-to-end scaling of Odgi Layout deviates from this linear behavior due to a sequential pre-processing step that generates the path-index. Its processing time becomes increasingly prominent as PGSGD's parallelization increases. Beyond this deviation, we observe near linear scaling efficiency below the theoretical optimum, which we attribute to the memory bottleneck caused by PGSGD's random access pattern. Synchronization barriers between the kernel's 30 iterations further limit parallelization, as faster threads must wait for slower threads to complete their iteration before proceeding.

For seqwish (v0.7.11), performance gains beyond four threads appear to be negligible. Closer inspection reveals that the latency-hiding method utilized within the TC loop limits potential performance gains. This method starts the graph emission for the current chunk in a separate thread while initiating the parallelized main transclosure computation of the next chunk. However, graph emission of the next chunk can only begin once the previous emission completes. Consequently, parallelizing the main transclosure computation shifts the bottleneck to the graph emission logic.

Furthermore, even if graph emission was additionally accelerated, thread scaling within the transclosure operation remains limited, partially due to load balancing between threads in the overlap-collect and union-find operation. Moreover, seqwish contains significant setup and teardown code with varying scalability characteristics. While some components scale well (such as unpacking the PAF alignments), others (like the final GFA file generation) contain significant sequential code segments. However, achieving high end-to-end thread scalability requires parallelization of all components.

5.2. CPU Microarchitecture

In this section we analyze the microarchitectural bottlenecks of the extracted benchmarks. Figure 6 shows a

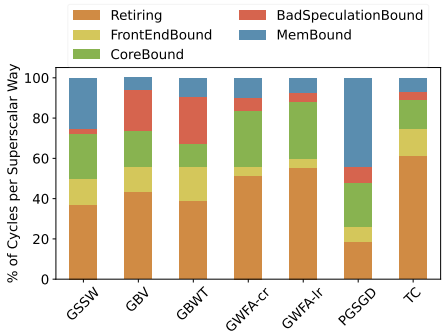


Figure 6: Top-Down Microarchitectural Analysis for Our Benchmarks: The y-axis shows the fraction of superscalar ways retiring or stalled for different reasons.

GSSW	GBV	GBWT	GWFA-cr	GWFA-lr	PGSGD	TC
1.77	2.22	1.92	2.67	2.90	0.88	3.14

TABLE 6: Kernel IPC (4-way CPU core)

breakdown of bottlenecks collected from top-down VTune microarchitecture analysis [49]. The y-axis shows the fraction of superscalar ways per cycle that are doing useful work (*Retiring*), stalled for some reason (*FrontEndBound*, *CoreBound*, or *MemoryBound*), or squashed due to misspeculation, which is mostly due to branch misprediction in our workloads (*BadSpeculationBound*). If an instruction was not issued because the backend of the pipeline was stalled, we classify the way as *CoreBound* or *MemoryBound*. If the backend had free slots, but there was no micro-op to dispatch, the way is classified as *FrontEndBound*. Table 6 shows the IPC for each kernel.

The results show that dynamic programming kernels GSSW, GBV, and GWFA, are core bound. This is understandable because these are compute-intensive kernels with complex data dependencies on previous cells, but unlike GBV and GWFA, GSSW uses affine gap scoring, which triples the memory footprint. Because of this, and other memory bottlenecks discussed in Section 6.1, GSSW *is also memory bound*.

In contrast, GBV is limited by branch misprediction attributed partly to the traceback step implemented within the kernel and partly to the merge operations between parent cells (Figure 4 b)). The merge operation is distinctly a graph-based step, where data from incoming edges is combined.

We notice that GWFA has a higher IPC when aligning long reads compared to a single chromosome. This is likely because the gaps to be aligned are larger for chromosomes than long reads. The longer sequences cover more nodes in the graph resulting in more control and memory divergence seen in figure 6.

Prior work has found the traditional BWT-based FM-Index to be memory bandwidth-intensive [4, 50], but GBWT is *not* memory bound. Due to the limited alphabet of four base pairs, substrings of the reference text are often repeated, so for a short substring there are many possible prefixes, any of which could be the next hop. This results in unpredictable memory accesses to the compressed occurrence table [34]. In contrast, GBWT is a multistring BWT over haplotypes (sequences of node IDs). These haplotypes are less likely to repeat themselves, so for a given substring, there are usually only a handful of prefixes. This limits the range of lookups, making index search queries more likely to access nearby nodes and therefore adjacent entries in the occurrence table [33]. The lack of memory bottlenecks exposes branch misses and frontend stalls from the data dependent control flow in GBWT.

Unlike the alignment kernels which align reads to small localized subgraphs, graph visualization is performed on the entire graph, resulting in a much larger memory footprint (1.7Gb for chromosome 20). To ensure the generation of high-quality layouts, PGSGD performs uniform random queries to this structure, independent of the graph structure. This results in a memory bottleneck. Furthermore, the coordinate update involves multiple divisions and square-root operations (e.g., for computing the Pythagorean theorem). Memory and core-boundness result in a low IPC.

TC also operates on a full chromosome dataset composed of multiple pairwise alignments, but it's accesses are much more regular. TC performance is dominated by efficient accesses to optimized data structures (like implicit interval trees [36] and atomic bitvectors [51]), the simple appending of data to files, and highly optimized sorting [37], resulting in a high percentage of retiring superscalar ways and IPC.

Perhaps surprisingly, we do not observe memory bottlenecks from indirect *graph accesses* in the kernels we have reviewed. GSSW, GBV, and GWFA operate on subgraphs which fit in the cache. GBWT exhibits good locality because of its graph representation and cache-friendly optimizations. PGSGD is memory bound because of its random sampling method, not because of the graph structure, and TC utilizes efficient data structures and can exploit some spatial locality, e.g., during the bitvector accesses.

Figure 7 shows cache misses per kilo-instructions. We note that the dynamic programming kernels rarely miss the $l3$ indicating that they are not making many irregular accesses to the full pangenome. Instead, they mostly miss the $l1$. This is likely because they perform local alignment on small subgraphs. In contrast, PGSGD stands out with a high miss rate across all levels of the cache. This is because PGSGD has completely random accesses to the graph, that does not fit in any level of the cache, despite using a cache-optimized data layout [27].

Figure 8 gives a dynamic instruction count for each of the kernels. Because x86 is a CISC instruction set, many instructions fit multiple categories, for instance an instruction might be a vector control flow operation. We chose to bin instructions hierarchically in the order they appear in the legend (read top to bottom, left to right). We classify mov instructions between registers as *Register* instructions.

We analyze usage of vector instructions in the dynamic programming benchmarks: GSSW, GBV, GWFA. Notice that GWFA has fewest vector instructions. This is surprising because previous work suggests that WFA is easily autovectorized [17]. Our results indicate that graph related

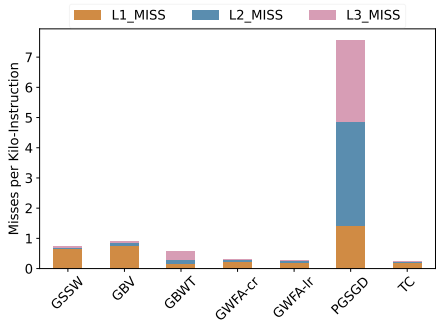


Figure 7: Misses per Kilo-Instruction for Our Benchmarks: collected with VTune: These counts are exclusive, that is if an instruction misses in *l*2 it is not shown as an *l*1 miss.

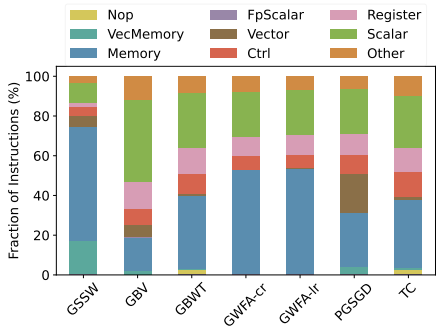


Figure 8: Dynamic Instruction Count for Our Benchmarks: collected with Intel PIN.

modifications prevent the compiler (g++11) from autovectorizing the code. Manually vectorizing the code could result in performance gains for GWFA. In contrast, we see GSSW has a significant fraction of vector and memory operations because it is a memory-bound, hand-vectorized code. GBV bitvectors are restricted to 64 bits in the code, so these operations are classified as scalar. We also observe that PGSGD makes heavy use of SSE instructions, which are classified as vector, but a closer look reveals that these are mostly SSE floating point scalar operations (e.g. MULSD) used in distance calculations. The other kernels, GBWT and TC, consist mostly of scalar operations. GBWT alternates between scalar operations on compressed data and lookups to find the next node, resulting in a mix of scalar and memory operations.

5.3. GPU Utilization

Table 7 shows GPU utilization metrics for TSU and PGSGD-GPU.

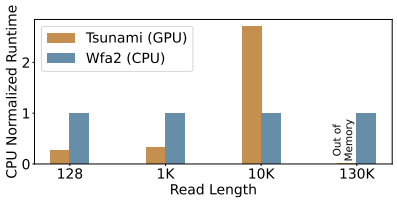


Figure 9: GPU vs CPU WFA Timing

	Occupancy	Warp Util.	Memory BW Util.
TSU	32.97%	69.72%	39.89%
PGSGD	53.85%	88.31%	41.91%

TABLE 7: GPU Microarchitecture Utilization

We find TSU is bottlenecked by thread divergence rather than memory bandwidth. Although the warp utilization is measured at 69.72%, many of these warps are not contributing useful work. Furthermore, the achieved warp utilizations comes from limiting the block size to 32. This in turn leads to poor occupancy, limiting ability to mask latency. On average, warps in TSU are issued only every 2.3 cycles instead of every cycle.

We evaluate the runtime of TSU over a range of read lengths comparing to the state of the art CPU version, WFA2-lib, Figure 9. For short reads TSU attains a speedup of up to 3.7x, but for long reads, it gives a slowdown because most diagonals don't have enough work in the *Extend* step to utilize the thread block. For 10K base pair reads, 74% of diagonals use only a single thread of the block compared to 0.3% for 128 base pair reads.

PGSGD-GPU achieved a 3.8x speedup over the CPU variant running with 32 threads on Machine B. This speedup is lower than the 8.9x speedup reported by Li et al. [27] because we measure end-to-end performance including setup/teardown, and we evaluate a smaller pangenome.

The achieved occupancy of 53.8% (see Table 7) is constrained by two factors. First, the theoretical occupancy is limited to 66.7%, establishing an upper bound. Second, due to memory bottlenecks, warps are issued only every 41.7 cycles per scheduler rather than every cycle. Memory bandwidth utilization reaches only 41.9% of capacity, attributed to low L1 and L2 cache hit rates of 31.5% and 49.3%, respectively. These low hit rates result from the random access patterns required by PGSGD and the large pangenome graph that cannot fit in cache. Furthermore, the random access patterns of different threads within the same warp lead to uncoalesced memory access, forcing sequential memory operations to different regions for each thread. Warps can only be issued once data from all threads within the warp is received. Besides, the PGSGD-GPU's warp merging technique appears effective, as profiling shows high warp utilization of 88.3%.

Theoretical occupancy improvements would directly increase achieved occupancy. Full occupancy (100%) cannot be achieved on the RTX A6000 because each PGSGD-GPU thread requires 44 registers, exceeding the Streaming Multiprocessor's (SM) register capacity at full occupancy. However, reducing the block size from 1024 to 256 threads would improve the theoretical occupancy to 83.3% (+16.7%) by enabling the GPU to schedule five thread blocks per SM simultaneously. This modification increases achieved occupancy by 20.8% and improves L1 and L2 hit rates by 10.3% and 6.4%, respectively, resulting in a modest 0.7% memory throughput increase. Overall, reducing the block size from 1024 to 256 accelerates the end-to-end run time by 1.1x.

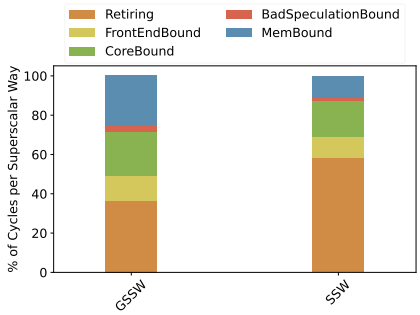


Figure 10: Comparison of Seq2Seq and Seq2Graph Mapping: Microarchitectural analysis of Graph SIMD Smith-Waterman (GSSW) and Seq2Seq algorithm, Striped Smith-Waterman (SSW) collected with Intel VTune.

6. Case Studies

6.1. Seq2Seq and Seq2Graph Comparison

In this case study, we compare GSSW with the Seq2Seq algorithm it was developed from , Stripped Smith-Waterman (SSW). We generate inputs for SSW by aligning our short reads (Section 4) to reference HG002 with BWA-MEM [52]. The input traces to the alignment function are recorded and subsequently used for SSW. For GSSW, we align the same reads to the Minigraph-Cactus graph using Vg Map.

The microarchitectural analysis in Figure 10 shows GSSW has $\approx 3\times$ more memory stalls. VTune shows these stalls result from swizzle writes to the Dynamic Programming matrix from the packed SIMD buffers shown in Figure 4a, which exhibits poor locality.

In Seq2Seq alignment, the computation of a row i depends *only* on row $i-1$. Therefore, SSW stores only the previous row. In contrast, graph alignment algorithm GSSW stores all rows of the dynamic programming matrix because row i may access any nodes it depends on, resulting in more swizzle writes (see incoming edges in Figure 4a). However, we observe that within a node, the rows exhibit linear dependencies, meaning these rows do not need to be stored. This optimization could improve performance by avoiding the costly writebacks from SIMD buffers to DP matrix.

6.2. Graph Variation

Reference genomes typically have only one representation, but the same pangenome can be represented by many graphs with different computational behavior. To explore this we compare GSSW performance on the Chr20 Minigraph-Cactus graph [15] (the M-Graph), and a graph with smaller nodes, the Split-M-Graph. We produce the Split-M-Graph by artificially splitting each node in the M-Graph with more than 8 base pairs into a chain of nodes with 8 base pairs each, reducing the average node length from 27.22 to 6.89 base pairs. We run Vg Map on the graphs and record the input traces to GSSW. The average subgraph size in the M-Graph traces is 450 base pairs, but in the Split-M-Graph it is only 233 base pairs because finer-grained nodes enable the chaining, clustering, and filtering steps to more precisely identify reference graph regions. This explains the faster runtime shown in Figure 11 despite similar microarchitectural utilization, and demonstrates how different graphs of the same pangenome can have large effects on performance.

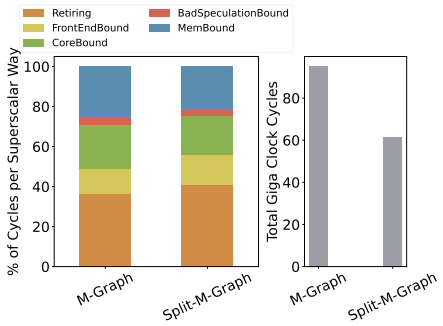


Figure 11: Comparison of M and Split-M Graph with GSSW: GSSW runs on Minigraph-Cactus graph and modified Minigraph-Cactus graph with split nodes. Left: microarchitecture analysis. Right: number of cycles GSSW took.

7. Background and Related Work

BioPerf and BioBench are some of the earliest genomics benchmarks [5, 6]. These include a diverse set of kernels for DNA and Protein analysis including alignment and phylogenetic analysis. GenomicsBench profiles more recent tools from de novo assembly, reference-based assembly, and metagenomics on CPU and GPU [4]. GenArchBench ports the kernels from GenomicsBench to ARM for HPC, and it adds some new kernels, notably WFA and Myers's Bitvector Alignment [7]. Genomics-GPU is another benchmark suite written specifically for GPU containing many kernels from the mapping pipeline [8]. PangenomicsBench stands apart from these as the first benchmark suite targeting pangenomics, which we have shown to be an interesting subfield with different computational characteristics.

8. Conclusion

In this work, we analyzed widely used tools from two pangenomic workflows, including a thread-scaling analysis. We identified key computational CPU and GPU kernels and developed a new benchmark suite, *PangenomicsBench*. We characterized the kernels based on their microarchitectural bottlenecks and conducted two case studies to explore the effects of input graph variation and the differences between Seq2Seq and Seq2Graph alignment. We find new challenges and opportunities for hardware software codesign in pangenomics that distinguish it from traditional genomics.

Acknowledgments

This work was generously supported by NSF GRFP fellowship, NSF awards (#2403119, #2118709 #2118743, and #2118709), NIH awards. (R01HG013618, U01HG013760, U01DA057530, U41HG010972) and the Tennessee Center for Integrative and Translational Genomics.

References

[1] R. M. Sherman and S. L. Salzberg, "Pan-genomics in the human genome era," *Nature Reviews Genetics*, vol. 21, no. 4, pp. 243–254, 2020.

[2] A. Guarracino, S. Buonaiuto, L. G. de Lima, T. Potapova, A. Rhie, S. Koren, B. Rubinstein, C. Fischer, J. L. Gerton *et al.*, "Recombination between heterologous human acrocentric chromosomes," *Nature*, vol. 617, no. 7960, pp. 335–343, 2023.

[3] "Computational pan-genomics: status, promises and challenges," *Briefings in bioinformatics*, vol. 19, no. 1, pp. 118–135, 2018.

[4] A. Subramaniyan, Y. Gu, T. Dunn, S. Paul, M. Vasimuddin, S. Misra, D. Blaauw, S. Narayanasamy, and R. Das, "Genomicsbench: A benchmark suite for genomics," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 1–12.

[5] D. A. Bader, Y. Li, T. Li, and V. Sachdeva, "Bioperf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications," in *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.* IEEE, 2005, pp. 163–173.

[6] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "Biobench: A benchmark suite of bioinformatics applications," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* IEEE, 2005, pp. 2–9.

[7] L. López-Villellas, R. Langarita-Benítez, A. Badouh, V. Soria-Pardos, Q. Aguado-Puig, G. López-Paradís, M. Doblas, J. Setoain, C. Kim, M. Ono *et al.*, "Genarchbench: A genomics benchmark suite for arm hpc processors," *Future Generation Computer Systems*, vol. 157, pp. 313–329, 2024.

[8] Z. Liu, S. Zhang, J. Garrigus, and H. Zhao, "Genomics-gpu: A benchmark suite for gpu-accelerated genome analysis," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 178–188.

[9] M. Rautiainen and T. Marschall, "Graphaligner: rapid and versatile sequence-to-graph alignment," *Genome biology*, vol. 21, no. 1, p. 253, 2020.

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

[11] J. Sirén, J. Monlong, X. Chang, A. M. Novak, J. M. Eizenga, C. Markello, J. A. Sibbesen, G. Hickey, P.-C. Chang, A. Carroll *et al.*, "Pangenomics enables genotyping of known structural variants in 5202 diverse genomes," *Science*, vol. 374, no. 6574, p. abg8871, 2021.

[12] H. Li, X. Feng, and C. Chu, "The design and construction of reference pangenome graphs with minigraph," *Genome biology*, vol. 21, pp. 1–19, 2020.

[13] G. Hickey, J. Monlong, J. Ebler, A. M. Novak, J. M. Eizenga, Y. Gao, T. Marschall, H. Li, and B. Paten, "Pangenome graph construction from genome alignments with minigraph-cactus," *Nature biotechnology*, pp. 1–11, 2023.

[14] E. Garrison, A. Guarracino, S. Heumos, F. Villani, Z. Bao, L. Tattini, J. Hagmann, S. Vorbrugg, S. Marco-Sola, C. Kubica *et al.*, "Building pangenome graphs," *Nature Methods*, pp. 1–5, 2024.

[15] W.-W. Liao, M. Asri, J. Ebler, D. Doerr, M. Haukness, G. Hickey, S. Lu, J. K. Lucas, J. Monlong, H. J. Abel *et al.*, "A draft human pangenome reference," *Nature*, vol. 617, no. 7960, pp. 312–324, 2023.

[16] A. Guarracino, N. Mwaniki, S. Marco-Sola, and E. Garrison, "wfmash: whole-chromosome pairwise alignment using the hierarchical wavefront algorithm. GitHub," 2021. [Online]. Available: https://github.com/waveygang/wfmash

[17] S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa, "Fast gap-affine pairwise alignment using the wavefront algorithm," *Bioinformatics*, vol. 37, no. 4, pp. 456–463, 2021.

[18] C. Jain, S. Koren, A. Dilthey, A. M. Phillippy, and S. Aluru, "A fast adaptive algorithm for computing whole-genome homology maps," *Bioinformatics*, vol. 34, no. 17, pp. i748–i756, 09 2018. [Online]. Available: https://doi.org/10.1093/bioinformatics/bty597

[19] G. Gerometta, A. Zeni, and M. D. Santambrogio, "Tsunami: A gpu implementation of the wfa algorithm," in *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2023, pp. 150–161.

[20] Y. Gao, Y. Liu, Y. Ma, B. Liu, Y. Wang, and Y. Xing, "abpoa: an simd-based c library for fast partial order alignment using adaptive band," *Bioinformatics*, vol. 37, no. 15, pp. 2209–2211, 11 2020. [Online]. Available: https://doi.org/10.1093/bioinformatics/btaa963

[21] E. Garrison and A. Guarracino, "Unbiased pangenome graphs," *Bioinformatics*, vol. 39, no. 1, p. btac743, 11 2022. [Online]. Available: https://doi.org/10.1093/bioinformatics/btac743

[22] D. Doerr, "Gfaffix," https://github.com/marschall-lab/GFAffix, 2022, accessed: 2024-12-11.

[23] A. Guarracino, S. Heumos, A. Novak, G. Hickey, P. Prins, J. Eizenga, A. Leonard, and E. Garrison, "smoothXG. GitHub," 2024. [Online]. Available: https://github.com/pangenome/smoothxg

[24] Y. Gu, A. Subramaniyan, T. Dunn, A. Khadem, K. Chen, S. Paul, M. Vasimuddin, S. Misra, D. Blaauw, S. Narayanasamy, and R. Das, "Gendp: A framework of dynamic programming acceleration for genome sequencing analysis," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ACM, 2023, pp. 1–15.

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

[26] 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, 2024.

[27] J. Li, J.-N. Schmelzle, Y. Du, S. Heumos, A. Guarracino, G. Guidi, P. Prins, E. Garrison, and Z. Zhang, "Rapid gpu-based pangenome graph layout," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '24. IEEE Press, 2024. [Online]. Available: https://doi.org/10.1109/SC41406.2024.00035

[28] M. Zhao, W.-P. Lee, E. P. Garrison, and G. T. Marth, "Ssw library: an simd smith-waterman c/c++ library for use in genomic applications," *PloS one*, vol. 8, no. 12, p. e82138, 2013.

[29] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[30] M. Farrar, "Striped smith–waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[31] M. Rautiainen, V. Mäkinen, and T. Marschall, "Bit-parallel sequence-to-graph alignment," *Bioinformatics*, vol. 35, no. 19, pp. 3599–3607, 2019.

[32] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *Journal of the ACM (JACM)*, vol. 46, no. 3, pp. 395–415, 1999.

[33] J. Sirén, E. Garrison, A. M. Novak, B. Paten, and R. Durbin, "Haplotype-aware graph indexes," *Bioinformatics*, vol. 36, no. 2, pp. 400–407, 2020.

[34] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform," *bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[35] H. Zhang, S. Wu, S. Aluru, and H. Li, "Fast sequence to graph alignment using the graph wavefront algorithm," *arXiv preprint arXiv:2206.13574*, 2022.

[36] E. Garrison, "mmmulti," Jun. 2020. [Online]. Available: https://github.com/ekg/mmmulti

[37] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, "In-Place Parallel Super Scalar Samplesort (IPSSSSo)," in *25th Annual European Symposium on Algorithms (ESA 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 87. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 9:1–9:14.

[38] 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.

[39] 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.

[40] K. Hoste, "Mica: Minimal instruction set architecture characterization," 2023, accessed: 2024-12-18. [Online]. Available: https://github.com/boegel/MICA

[41] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[42] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[43] S. Nurk, S. Koren, A. Rhie, M. Rautiainen, A. V. Bzikadze, A. Mikheenko, M. R. Vollger, N. Altemose, L. Uralsky, A. Gershman, S. Aganezov, S. J. Hoyt, M. Diekhans, G. A. Logsdon, M. Alonge, S. E. Antonarakis, M. Borchers, G. G. Bouffard, S. Y. Brooks, G. V. Caldas, N.-C. Chen, H. Cheng, C.-S. Chin, W. Chow, L. G. de Lima, P. C. Dishuck, R. Durbin, T. Dvorkina, I. T. Fiddes, G. Formenti, R. S. Fulton, A. Fungtammasan, E. Garrison, P. G. S. Grady, T. A. Graves-Lindsay, I. M. Hall, N. F. Hansen, G. A. Hartley, M. Haukness, K. Howe, M. W. Hunkapiller, C. Jain, M. Jain, E. D. Jarvis, P. Kerpedjiev, M. Kirsche, M. Kolmogorov, J. Korlach, M. Kremitzki, H. Li, V. V. Maduro, T. Marschall, A. M. McCartney, J. McDaniel, D. E. Miller, J. C. Mullikin, E. W. Myers, N. D. Olson, B. Paten, P. Peluso, P. A. Pevzner, D. Porubsky, T. Potapova, E. I. Rogaev, J. A. Rosenfeld, S. L. Salzberg, V. A. Schneider, F. J. Sedlazeck, K. Shafin, C. J. Shew, A. Shumate, Y. Sims, A. F. A. Smit, D. C. Soto, I. Sović, J. M. Storer, A. Streets, B. A. Sullivan, F. Thibaud-Nissen, J. Torrance, J. Wagner, B. P. Walenz, A. Wenger, J. M. D. Wood, C. Xiao, S. M. Yan, A. C. Young, S. Zarate, U. Surti, R. C. McCoy, M. Y. Dennis, I. A. Alexandrov, J. L. Gerton, R. J. O'Neill, W. Timp, J. M. Zook, M. C. Schatz, E. E. Eichler, K. H. Miga, and A. M. Phillippy, "The complete sequence of a human genome," *Science*, vol. 376, no. 6588, pp. 44–53, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.abj6987

[44] "Human pangenome reference consortium s3 bucket," accessed: 16-Jun-2024. [Online]. Available: https://s3-us-west-2.amazonaws.com/human-pangenomics/index.html?prefix=pangenomes/freeze/freeze1/minigraph-cactus/hprc-v1.1-mc-grch38/

[45] M. Vasimuddin, S. Misra, H. Li, and S. Aluru, "Efficient architecture-aware acceleration of bwa-mem for multicore systems," in *2019 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 2019, pp. 314–324.

[46] 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 *et al.*, "The human pangenome project: a global resource to map genomic diversity," *Nature*, vol. 604, no. 7906, pp. 437–446, 2022.

[47] J.-X. Yue, J. Li, L. Aigrain, J. Hallin, K. Persson, K. Oliver, A. Bergström, P. Coupland, J. Warringer, M. C. Lagomarsino *et al.*, "Contrasting evolutionary genome dynamics between domesticated and wild yeasts," *Nature genetics*, vol. 49, no. 6, pp. 913–924, 2017.

[48] A. Guarracino, S. Heumos, S. Nahnsen, P. Prins, and E. Garrison, "Optimized Dynamic Genome/Graph Implementation (ODGI)," Sep. 2021. [Online]. Available: https://github.com/pangenome/odgi

[49] 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.

[50] A. Subramaniyan, J. Wadden, K. Goliya, N. Ozog, X. Wu, S. Narayanasamy, D. Blaauw, and R. Das, "Accelerated seeding for genome sequence alignment with enumerated radix trees," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21, 2021.

[51] E. Garrison, "atomicbitvector," 2020. [Online]. Available: https://github.com/ekg/atomicbitvector

[52] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with bwa-mem," *arXiv preprint arXiv:1303.3997*, 2013.

Appendix

1. Abstract

The PangenomicsBench GitHub repository contains the code for the CPU/GPU tools, extracted kernels, and profiling scripts (run time, microarchitecture, and thread-scaling analysis). Furthermore, the repository contains instructions for the scripts and data access.

2. Artifact check-list (meta-information)

- **Algorithm:** Genomics, Pangenomics, Sequence Alignment, Pangenome Construction.
- **Program:** PGGB, Minigraph-Cactus, Vg, GraphAligner.
- **Compilation:** GNU, CUDA.
- **Binary:** x86.
- **Data set:** HPRC, VG, GFA, OG Graphs, Long Reads, Short Reads, Assemblies.
- **Hardware:** NVIDIA GPU, Intel CPU.
- **Metrics:** Run time, VTune Uarch, VTune Hotspots, NCU Profiles.
- **Output:** Run times, VTune and NCU Profiles.
- **How much disk space required (approximately)?:** 100GB of storage.
- **How much time is needed to prepare workflow (approximately)?:** 2 hours (mostly automated).
- **How much time is needed to complete experiments (approximately)?:** 2 hours for kernel timings. 1 day/overnight for all studies.
- **Publicly available?:** Github: https://github.com/UM-mbit/pangenomicsBench and Zenodo: https://doi.org/10.5281/zenodo.16907988Zenodo.
- **Workflow automation framework used?:** Toil for Minigraph-Cactus.
- **Archived (provide DOI)?:** Zenodo: https://doi.org/10.5281/zenodo.16907988

3. Description

The paper makes use of two kinds of studies: kernel studies, and tool studies.

The kernel studies make up our benchmark suite and running them is mostly automated. The user can change configuration variables in the `mainRun.py` to enable or disable profiling studies. By default, we only run timing analysis. Data from these kernels can be used to produce Figures 6, 8, and 7 and Table 6. We also include scripts for running GPU experiments to gather the data for Figure 9 and Table 7. Documentation for running the kernel studies can be found in the README of the GitHub repository.

The tool studies, including the thread-scaling analysis, are also in the GitHub repository under `ToolAnalysis`. These require building the full pangenomics tools, and running through the steps of the `ToolAnalysis/README.md`. This data is used for Figures 2, 3, and 5.

3.1. How to access. The code and documentationd are available on GitHub: https://github.com/UM-mbit/pangenomicsBench.

3.2. Hardware dependencies. An x86 CPU is required to run the CPU kernels and tools. A GPU is needed for the `TSUNAMI` and `PGSGD-GPU` kernels. Due to the profiling tools (Intel VTune and NVIDIA Nsight compute) used in this work, an Intel CPU and an NVIDIA GPU are required for profiling.

3.3. Software dependencies. The benchmark suite is tested for Ubuntu 24.04 and 22.04, and has the following dependencies:

1) conda (tested with 24.7.1).
2) cmake (tested with 3.26.0)
3) gcc/g++ (We use gcc 9, 11, and 13).

Additional dependencies for each kernel can be found in the READMEs of their submodules located in `<kernelName>/deps`. This documentation is repeated in the GitHub README.

3.4. Data sets. Kernel datasets can be found at https://genomicsbench.eecs.umich.edu/Kernels.tar.gz. Tool datasets can be found at https://genomicsbench.eecs.umich.edu/ToolDataPangenomicsBench.tar.gz. They can be downloaded following the instructions in the READMEs.

4. Installation

To install PangenomicsBench, run the following script. Please do not forget the `--recursive` flag to ensure submodules installed. Follow the instructions of the main README afterwards.

```
git clone --recursive https://github.com/UM-mbit/
    pangenomicsBench
```

5. Experiment workflow

The kernel benchmarks can be built with a bash script, and then run with a python script which is configurable based on the studies you want to run. This workflow is described in the GitHub README.

The tool analyses require each pangenomics tool to be built seperately. Then the profiling analyses can be run with one script, and the thread scaling can be run with another. This workflow is described in the github README in the `ToolAnalysis` directory.

6. Evaluation and expected results

6.1. CPU kernels. `mainRun.py` generates the directory `AllRunsOut` which includes data produced by each kernel. Summary statistics are produced in the directory `AllRunsOut/<KernelName>/Results`. Execution logs, VTune profiles, and VTune reports are also produced for each kernel. The data created depends on the types of profiling runs configured in the `mainRun.py` script. By default, only the run time is produced. Microarchitecture, cache, and dynamic instruction count profiling can be activated.

6.2. GPU kernels. `runGpu.sh` produces run times and nsight-compute profiling reports for the GPU kernels.

6.3. Tool-thread-scaling. `ToolAnalysis/runScaling.sh` produces run times for each tool at different thread counts.

6.4. Tool-timing-breakdown. For each Seq2Graph tool `ToolAnalysis/runVtune.sh` generates a VTune Hotspot Analysis, which can be used to generate a timing breakdown. For the graph building pipelines, we produce run time breakdowns for each stage of the pipeline which are printed to standard output and captured in log files (`ToolAnalysis/PipelineResults/MinigraphCactus`, `ToolAnalysis/PipelineResults/Pggb/runPGGBBreakdown.sh`).