Motivation

GPU-Accelerated Sequence-to-Graph Alignment

Elizabeth Terveen & Samriddhi Bhardwaj

Sequencing machines produce millions of DNA fragments — to make sense of them, each fragment must be aligned to a reference genome.

1. Reads need a reference

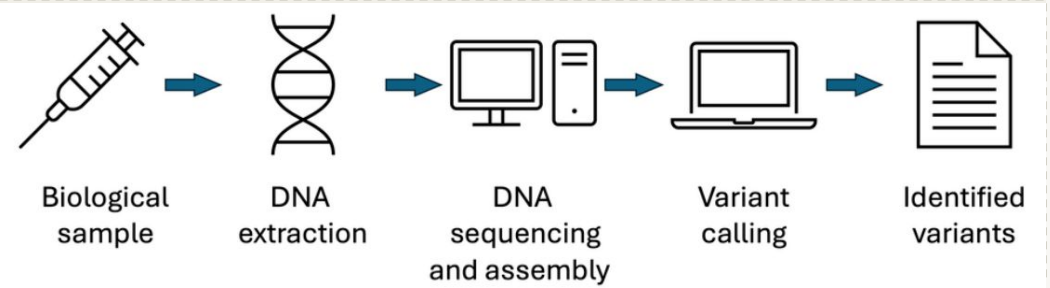
The linear reference is a single sequence — but biased toward European ancestry, missing variants from billions of people

2. Pangenome graphs fix bias

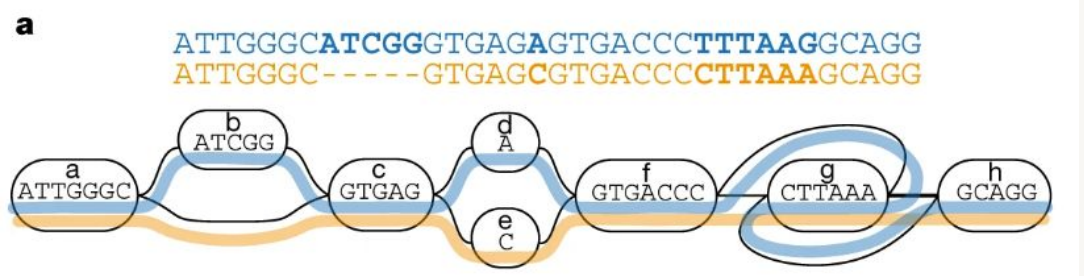
Encode full population diversity as a DAG — but alignment cost explodes to O(NM) DP over a graph

3. Scale is the bottleneck

Millions of reads per sequencing run — every one must be aligned — cost is critical. Each alignment is O(NM) — at 30M reads per run, this is the dominant compute cost



Sequencing pipeline



Pangenome graph encoding population variants (adapted from Liao et al. 2023)

We GPU-accelerate graph alignment using parallelism within reads and batch parallelism across reads.

43.7×

speedup over sequential · RTX 2080

Background & Related Work

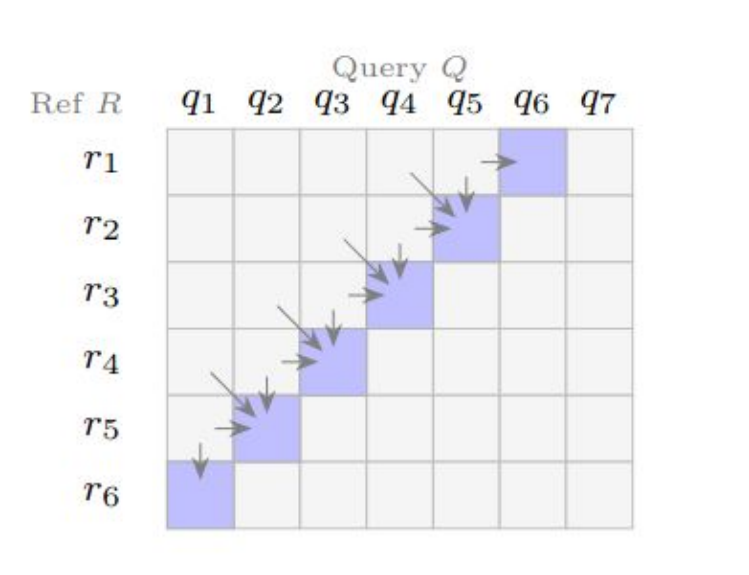


Fig 1: Smith-Waterman DP matrix

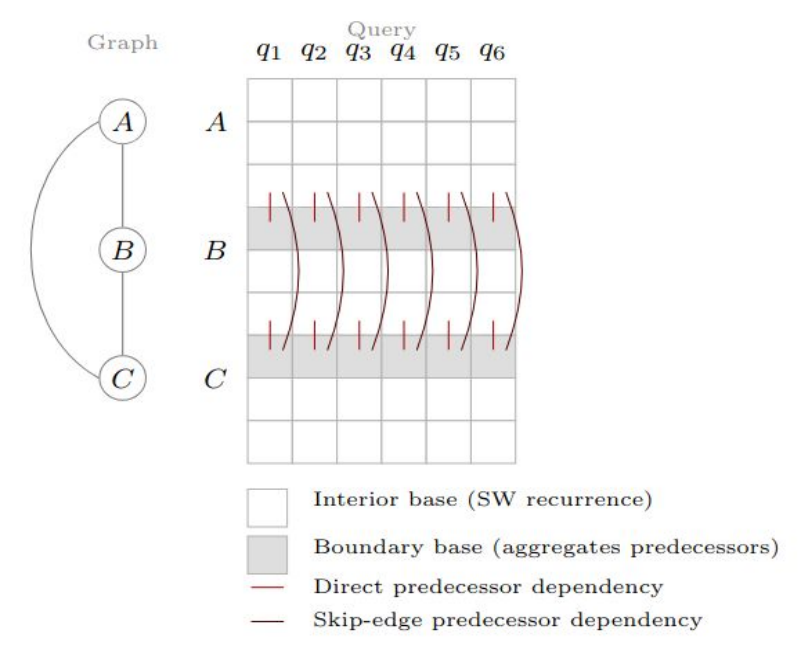


Fig 2: POA DP over a graph.

Smith-Waterman + Gotoh (1981–82)

- Standard DP for local sequence alignment with affine gap penalties (open + extend)
- Antidiagonal independence: cell (i,j) depends only on (i−1,j−1), (i−1,j), (i,j−1)

Partial Order Alignment / POA (Lee 2002)

- Extends SW to a DAG reference — interior bases follow standard SW recurrence
- Boundary bases aggregate scores from all predecessor nodes → breaks antidiagonal independence
- Graph must be processed in topological order

HGA vs. Our Work

	HGA (Feng & Luo 2021)	Our Work
Parallelism	Inter-read only (1 thread/read)	Inter + intra-read (1 block/read)
Gap model	Linear (1 matrix)	Affine (3 matrices)

**Affine model requires 3× more computation per cell than linear gap*

V1 & V2: Antidiagonal Tiling

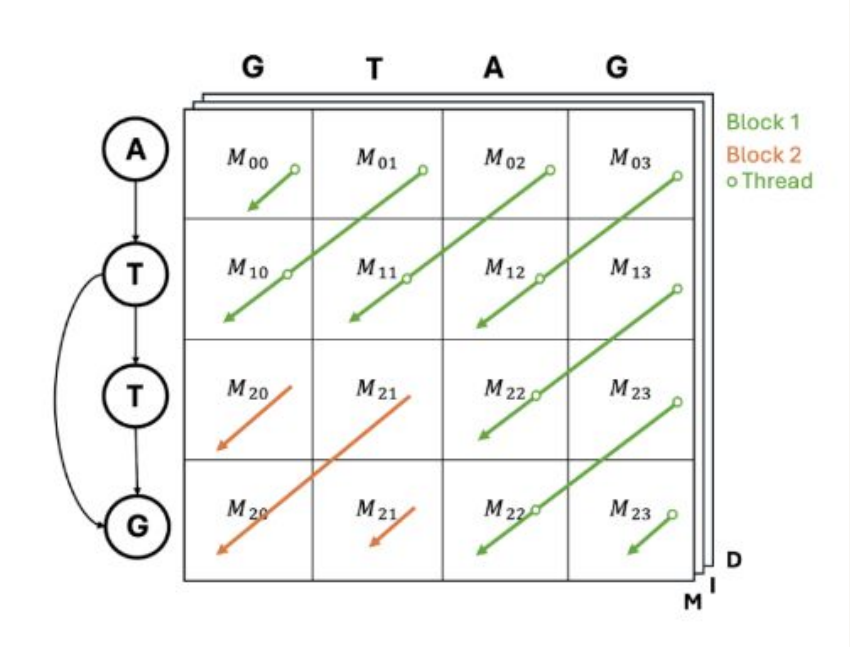


Fig 1: V1 CUDA thread mappings

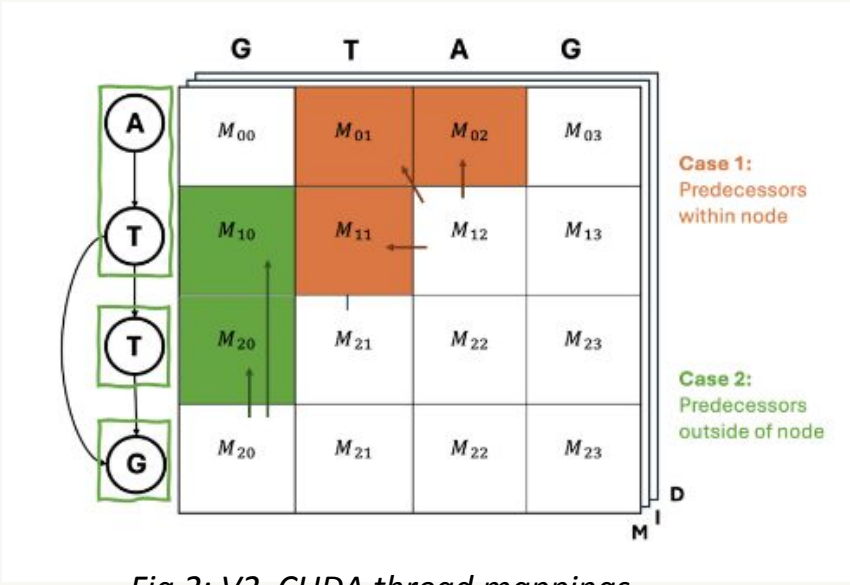


Fig 2: V2 CUDA thread mappings

V1: Antidiagonal Tiling

- One thread per cell on current antidiagonal
- One kernel launch per antidiagonal → O(M+N) launches + cudaDeviceSynchronize overhead
- Full DP matrix in global memory → up to 6 GB for large inputs
- Occupancy <20% — short antidiagonals leave most threads idle

Bottleneck: memory — 6 GB approaches RTX 2080 limit, cannot scale

V2: Rolling Buffer Optimization

- Replaces full DP matrix with 2 rolling rows → memory O(NM) → O(N·|nodes|)
- node_bound buffer stores last row of each predecessor node for boundary bases
- Tradeoff: host scrolls dp_prev ← dp_curr after every row → kernel launches jump to O(M·N)

Bottleneck: kernel launch overhead — O(M·N) host-device round trips dominate runtime

V1 hit a memory wall. V2 fixed memory but exposed kernel launch overhead. Both motivate V3's single cooperative kernel.

V3: Single Cooperative Kernel

V2's O(M·N) kernel launches are eliminated entirely — one kernel owns all loops, synchronizing at the device level.

Key Optimizations

1. **Single cooperative kernel launch**

   cudaLaunchCooperativeKernel replaces per-antidiagonal launches. The kernel owns all loops internally — no host-device round trips.

2. **grid.sync() device-level barrier**

   Replaces cudaDeviceSynchronize(). Synchronizes all threads across the entire grid at device level between antidiagonals.

3. **3 rotating antidiagonal buffers**

   Diagonal d writes to buffer d mod 3, reads from (d+2) mod 3 and (d+1) mod 3. Memory stays O(M) — three rows of values.

4. **Strided row assignment**

   Thread t handles rows t, t+stride, t+2·stride, … A fixed grid size covers all M rows regardless of graph size. atomicMax tracks best score — no per-row memcpy.

Profiles Results:

Metric	Value
Achieved Occupancy	**100%**
Theoretical Occupancy	**100%**
Compute (SM) %	**11.72%**
Memory %	9.54%
DRAM Throughput	0.01%

The occupancy paradox

Warps are fully scheduled (100% occupancy) but idle at grid.sync() barriers between antidiagonals. The antidiagonal dependency enforces serialization that no optimization can eliminate — compute is only 11.72%.

100% occupancy but only 11.72% compute — warps stall at barriers. The fix: keep threads busy with multiple reads simultaneously.

V4: Inter-Read Parallelism

Align multiple reads concurrently so idle warps do useful work.

Key Ideas

Occupancy scaling

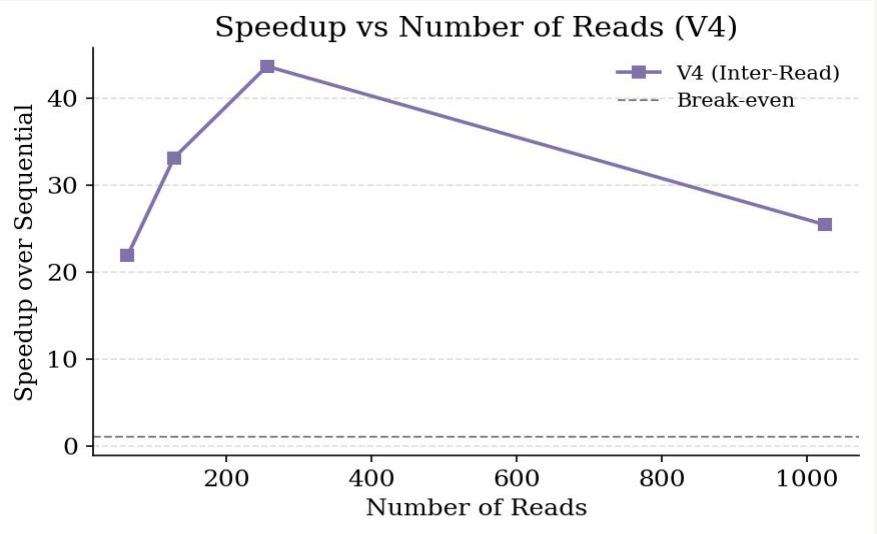
- One block per read — blocks fully independent, no cross-read synchronization
- RTX 2080: 46 SMs × 3 blocks/SM = 138 effective block size (EBS)
- Fewer than 138 reads SMs sit idle. 2×EBS (276 reads) reaches 46% compute

Batch size	Compute %	Occupancy
64 reads	15.7%	35%
138 (1×EBS)	29.4%	53%
276 (2×EBS)	**45.9%**	**68%**

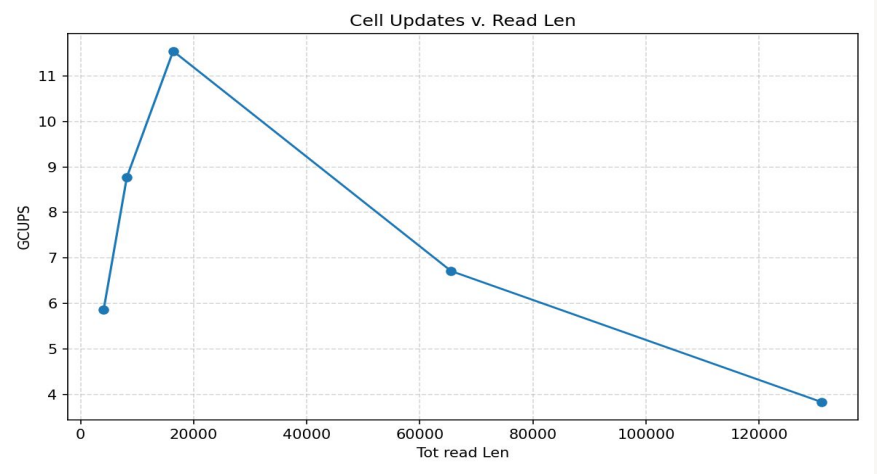
Memory characteristics

- L1/TEX cache hit rate: 98.62% — kernel runs almost entirely from shared memory
- DRAM throughput: 0.11% — near zero global memory traffic

Peak speedup



GCUPS vs. total read length

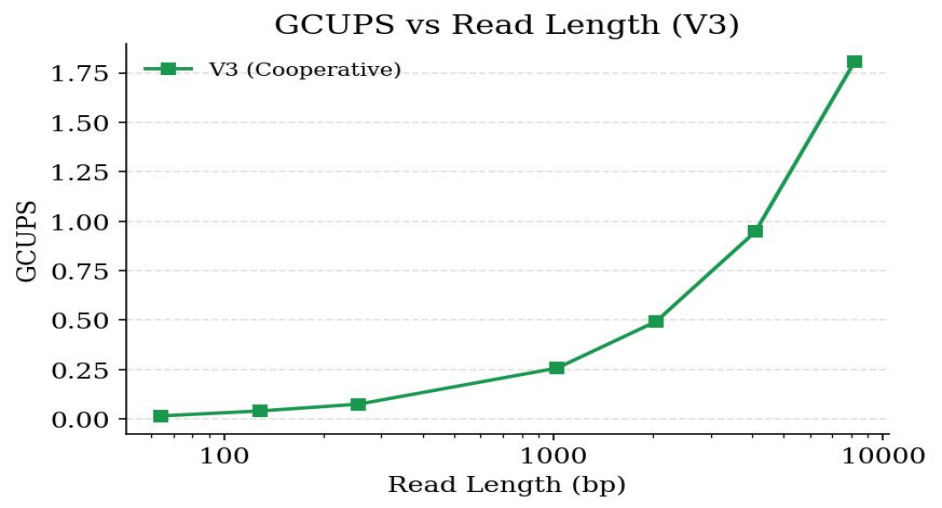


Combining inter-read batching with intra-read antidiagonal parallelism achieves 43.7× speedup — neither axis alone saturates the GPU.

Version Progression

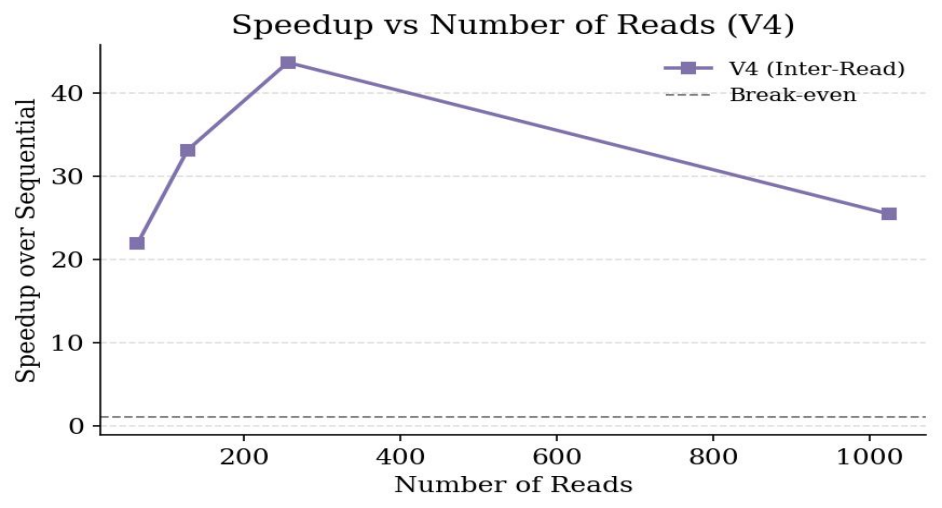
Version	Key change	Bottleneck fixed	New bottleneck	Occupancy
V0 (CPU baseline)	Sequential POA	—	—	—
V1 (Naïve GPU)	Antidiagonal parallelism	Sequential compute	Memory (8 GB)	<20%
V2 (Rolling buffers)	2-row rolling buffer	Memory wall	O(M·N) kernel launches	<20%
V3 (Cooperative)	Single kernel + grid.sync()	Kernel launch overhead	Barrier stalls (48.9%)	**100%**
V4 (Inter-read)	Batch reads, 1 block/read	Idle warps at barriers	—	**68%**

V3 intra-read



GCUPS grows superlinearly with read length.

V4 intra- + inter-read

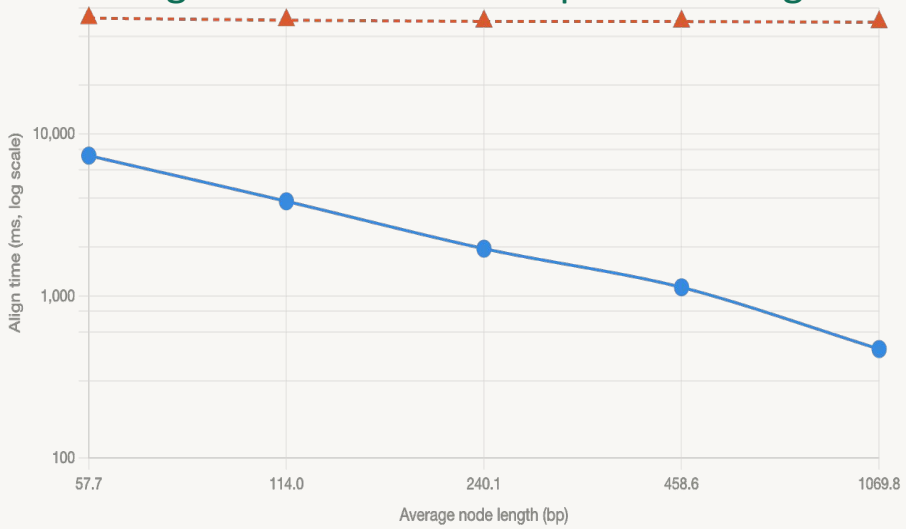


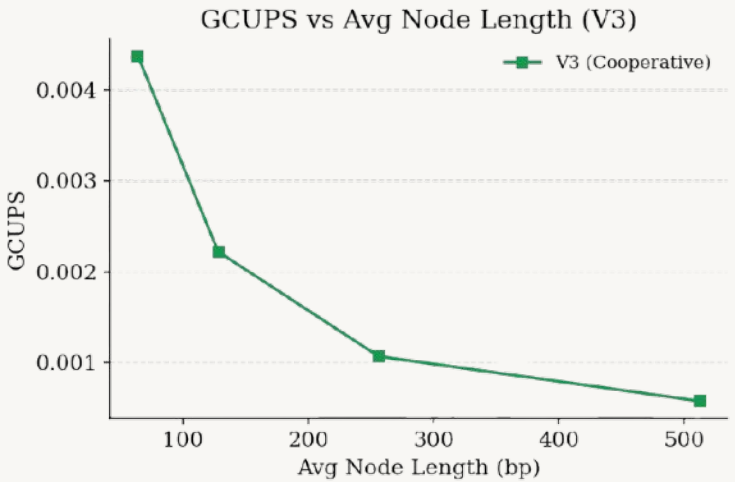
Speedup rises to 43.7× as more blocks fill SMs, then falls at 1024 reads as node_bound buffer exceeds L2 cache.

V3 optimizes intra-read parallelism. V4 adds inter-read batching to keep all SMs busy — together they achieve 43.7× speedup.

Graph Structure Effects

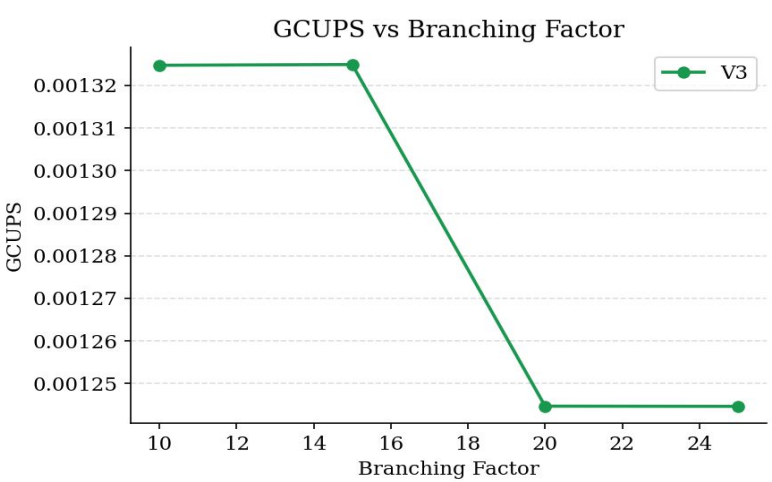
Node length: GPU runtime drops as nodes get longer (total bases fixed)



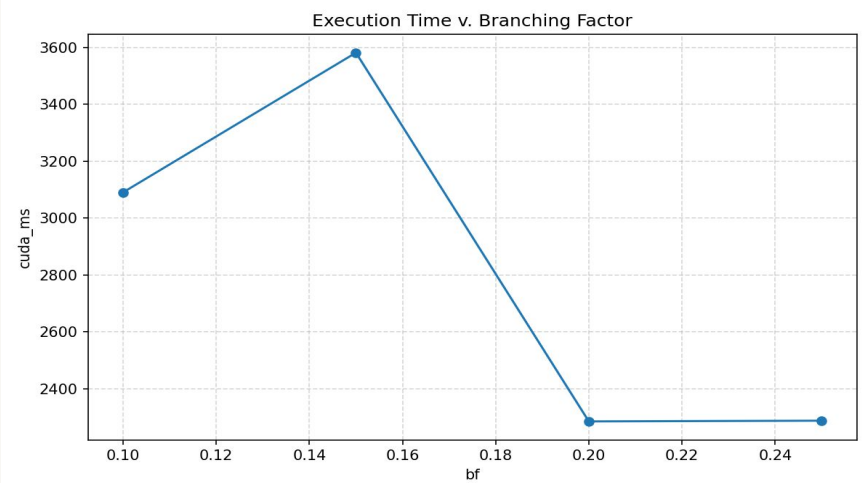


Short nodes (57bp) produce narrow antidiagonals — most threads idle. Longer nodes (1069bp) sustain wide antidiagonals, amortizing sync cost over more work. Sequential time (red) stays flat — total cells unchanged.

Branching factor — V3 (intra-read)



Branching factor — V4 (inter-read)



Node length is the dominant graph structure variable — longer nodes = wider antidiagonals = better GPU utilization. Branching factor has weak effect in SNP-dominated graphs.

Discussion & Future Work

Synchronization Overhead

V4 is barrier-stall bound. This is a fundamental consequence of the antidiagonal recurrence — not an implementation flaw.

Metric	Value
Compute (SM) %	45.86%
Memory %	45.86%
L1/TEX hit rate	**98.62%**
DRAM throughput	**0.11%**
Barrier stall %	**48.9%**

Compute and Memory utilization are balanced. 48.9% of warp cycles stalled at grid.sync() barriers. 11.95 cycles/instruction vs. ideal 1. Antidiagonal dependency forces serialization — structural, not fixable.

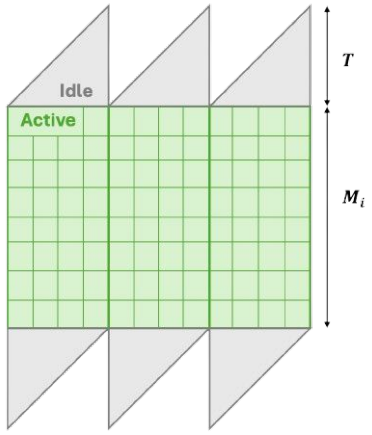
Other Bottlenecks

1 Shared memory sizing

Allocated to max node length — unused for shorter nodes. Caps blocks/SM to 3, suppressing occupancy in uneven graphs.

2 Warp divergence

Interior vs. boundary vs. out-of-bounds paths diverge within warps. Only ~20/32 threads active per warp on average.



Lessons learned

Intra + inter-read parallelism together are essential — neither alone saturates the GPU
Memory capacity was the binding scalability constraint, not compute throughput
Affine gap model (3× memory vs. linear) improves biology but costs raw GCUPS
CPU (abPOA) is more competitive for short reads and small graphs — GPU wins at scale

Future work

CSR-style flat predecessor array (like HGA's GCSR) to reduce boundary base memory loads
Adaptive banded DP (abPOA-style) to restrict active cells per antidiagonal and improve utilization
Dynamic shared memory allocation per node length to increase blocks/SM

43.7× speedup achieved — but the antidiagonal dependency is a fundamental constraint that no implementation trick can fully eliminate.