# GPU-Accelerated Sequence-to-Graph Alignment

Elizabeth Terveen
eterveen@andrew.cmu.edu

Samriddhi Bhardwaj
samriddb@andrew.cmu.edu

April 30, 2026

## Summary

We built a GPU-accelerated sequence-to-graph aligner implementing the Partial Order Alignment (POA) algorithm with affine gap penalties on an NVIDIA GeForce RTX 2080. Starting from a sequential CPU baseline, we developed four GPU versions that progressively exploit antidiagonal parallelism, rolling memory buffers, cooperative kernel execution, and inter-read parallelism. Our final implementation achieves a peak speedup of 43.7× over the sequential baseline demonstrating that combining intra-read antidiagonal parallelism with inter-read block-level concurrency is an effective strategy for GPU-accelerated graph alignment.

## 1 Background

### 1.1 Problem Description

**Motivation.** Modern genomic sequencing machines produce short DNA fragments called ***reads***, which must be aligned to a reference genome to identify genetic variants. The standard linear human reference genome cannot capture the full genetic diversity of a population and is systematically biased against individuals whose ancestry is underrepresented [1]. The pangenome graph addresses this by encoding an entire population's variation as a directed acyclic graph [2]. But mapping reads to this graph is far more computationally expensive than linear alignment. Further, in practice, sequencing platforms produce millions of reads per run, and biological conclusions are drawn from aggregate evidence across all of them. Intra-read parallelism reduces *latency* per alignment; inter-read parallelism reduces *throughput* time across the full dataset, which is what determines biological utility at scale. Our work exploits both axes simultaneously with GPU acceleration for both intra- and inter-read alignment.

**Formal Setup.** Given a *read* $Q$ of length $M$ and a sequence graph $G = (V, E)$ where each node $v \in V$ carries a nucleotide sequence of length $\ell_v$, the goal is to find the path through $G$ that best explains $Q$ under an affine gap scoring model. Let $N = \sum_{v \in V} \ell_v$ be the total number of bases in the graph. The alignment is computed via dynamic programming over an $M \times N$ matrix, giving a naive cost of $O(NM)$. This is the core computational bottleneck we seek to accelerate.

**DP Matrices.** The DP state at each cell $(i, j)$ consists of three values:

- $M[i][j]$: best score ending in a match or mismatch
- $I[i][j]$: best score with a gap (insertion) in the read
- $D[i][j]$: best score with a deletion in the read

These matrices are defined fully by the Smith-Waterman recurrence, described next.

### 1.2 Smith-Waterman Algorithm

Smith-Waterman (SW) [3] is the standard dynamic programming algorithm for local sequence alignment. Given a read $Q$ of length $M$ and a linear reference $R$ of length $N$, SW computes

the highest-scoring local alignment under a linear gap penalty model. The DP state at each cell $(i, j)$ is a single score matrix $H$, with recurrence:

$$H[i][j] = \max \begin{cases} 0 \\ H[i-1][j-1] + s(Q_i, R_j) \\ H[i-1][j] - g \\ H[i][j-1] - g \end{cases} \tag{1}$$

where $s(Q_i, R_j)$ is a match/mismatch score and $g$ is a fixed gap penalty. Gotoh [4] extended SW to an affine gap penalty model, introducing separate penalties for opening and extending a gap. This requires three matrices:

- $M[i][j]$: best score ending in a match or mismatch
- $I[i][j]$: best score with an insertion in the read (gap in reference)
- $D[i][j]$: best score with a deletion in the read (gap in query)

with recurrences:

$$M[i][j] = \max \begin{cases} 0 \\ M[i-1][j-1] + s(Q_i, R_j) \\ I[i-1][j-1] + s(Q_i, R_j) \\ D[i-1][j-1] + s(Q_i, R_j) \end{cases} \tag{2}$$

$$I[i][j] = \max \begin{cases} M[i-1][j] + g_{\text{open}} \\ I[i-1][j] + g_{\text{ext}} \end{cases} \tag{3}$$

$$D[i][j] = \max \begin{cases} M[i][j-1] + g_{\text{open}} \\ D[i][j-1] + g_{\text{ext}} \end{cases} \tag{4}$$

where $g_{\text{open}}$ is the gap opening penalty and $g_{\text{ext}}$ is the gap extension penalty. Affine penalties are biologically motivated. A multi-base insertion or deletion is more likely one mutation event than many independent ones, so extending an open gap should cost less than opening a new one. Our implementation uses the affine model. Both formulations have anti-diagonal independence because the cell $(i, j)$ depends only on $(i-1, j-1)$, $(i-1, j)$, and $(i, j-1)$ is the foundation of all GPU parallelism in this work.

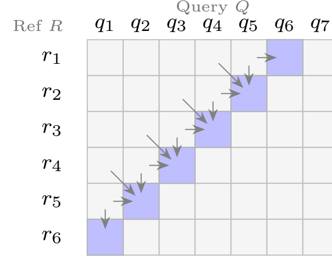


Figure 1: Smith-Waterman DP matrix with antidiagonal parallelism highlighted. Cells on the same antidiagonal (blue) share no data dependencies and can be computed simultaneously. Arrows show the three dependencies of each cell: top-left, top, and left.

### 1.3 Sequence-to-Graph Alignment (POA)

Partial Order Alignment (POA), introduced by Lee et al. [5], extends Smith-Waterman to a DAG reference. The DP recurrence differs from SW in the treatment of *boundary bases*.

**Interior Bases.** For a base $j$ in the interior of a node — that is, not the first base of its node — the recurrence is identical to the affine Smith-Waterman recurrence. These bases inherit their dependencies from the previous two antidiagonals within the same node, and antidiagonal parallelism applies without modification.

**Boundary Bases.** For the first base $j_v$ of a node $v$, the left neighbors are not within the same node but are the *last bases of all predecessor nodes.* The recurrence becomes:

$$M[i][j_v] = \max_{u \in \text{pred}(v)} \begin{cases} 0 \\ M[i-1][\text{last}(u)] + s(Q_i, g_{j_v}) \\ I[i-1][\text{last}(u)] + s(Q_i, g_{j_v}) \\ D[i-1][\text{last}(u)] + s(Q_i, g_{j_v}) \end{cases} \tag{5}$$

$$I[i][j_v] = \max_{u \in \text{pred}(v)} \begin{cases} M[i-1][\text{last}(u)] + g_{\text{open}} \\ I[i-1][\text{last}(u)] + g_{\text{ext}} \end{cases} \tag{6}$$

$$D[i][j_v] = \max_{u \in \text{pred}(v)} \begin{cases} M[i][\text{last}(u)] + g_{\text{open}} \\ D[i][\text{last}(u)] + g_{\text{ext}} \end{cases} \tag{7}$$

where $\text{pred}(v)$ is the set of predecessor nodes of $v$ and $\text{last}(u)$ denotes the index of the last base of node $u$. This boundary condition breaks simple antidiagonal parallelism — a node cannot begin computation until all its predecessors are complete, introducing a inter-node dependency that must be respected during GPU execution.

**Topological Order.** Because of these inter-node dependencies, the graph must be processed in topological order. All nodes at the same topological depth can be processed concurrently, while nodes at successive depths must be serialized.

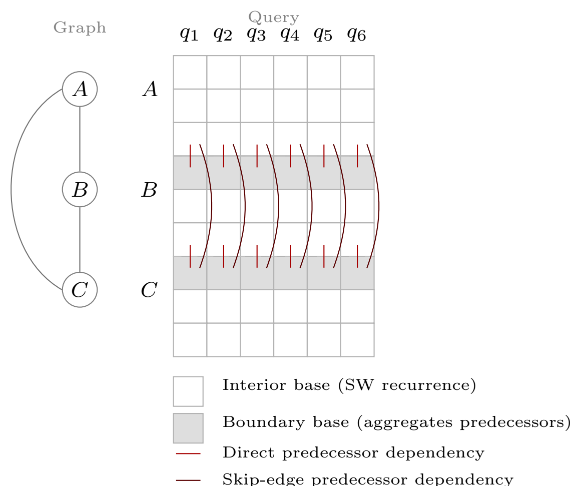


Figure 2: POA DP matrix over a sequence graph. Interior bases (white) follow the standard SW recurrence. Boundary bases (shaded) aggregate scores from predecessor nodes, breaking antidiagonal independence.

## 1.4 Related Work

**SIMD-Accelerated POA.** The dominant CPU approach to accelerating POA is SIMD vectorization, exemplified by SPOA [6] and abPOA [7]. Both tools exploit *intra-read* parallelism by placing SIMD lanes parallel to the query sequence, so that one SIMD instruction updates $k$ cells across $k$ different rows simultaneously, achieving up to $k = 8$ parallel units with SSE4.1 128-bit registers. abPOA further improves on SPOA with adaptive banded DP, which restricts computation to a dynamically chosen diagonal band around the likely alignment path, discarding cells unlikely to contain the optimal alignment.

However, both approaches are limited by the dependencies on neighboring bases, enforcing a strict left-to-right sequential dependency across columns. SIMD width is also a hard ceiling on parallelism. Our GPU approach sidesteps this limitation entirely by decomposing the DP matrix along antidiagonals. As described in Section 1, cells on the same antidiagonal are mutually independent and can be computed simultaneously, replacing a ceiling of $k = 8$ parallel units with thousands of concurrent GPU threads.

**GPU-Accelerated S2G Alignment.** HGA [8] is the first GPU-accelerated sequence-to-graph aligner and the closest prior work to ours. HGA achieves inter-sequence parallelism by assigning one thread per read, so that a batch of reads are aligned concurrently with no synchronization required between threads. To reduce global memory pressure, HGA introduces the GCSR data structure, which exploits the sparsity of variation graphs to shrink the graph representation and reduce memory loads. HGA also applies coalesced memory layouts for reads and a strided shared memory layout to avoid bank conflicts.

Our work differs from HGA in two key ways. First, HGA assigns one *thread* per read, meaning all intra-read parallelism is sacrificed — each thread computes the entire $M \times N$ DP matrix sequentially. We instead assign one *block* per read, exploiting antidiagonal parallelism within each read's DP matrix via warp-level cooperation. Second, HGA uses a linear gap penalty model, whereas our implementation uses the biologically motivated affine gap penalty model described in 1.2. This algorithm requires 3 matrices and hence 3× as much computation but is more biologically motivated and produces higher quality alignment results. The combination of inter-read parallelism across blocks and intra-read antidiagonal parallelism within blocks is the central contribution of our GPU design.

# 2 Approach

## 2.1 Overview

We describe five implementations of sequence-to-graph alignment, each motivated by the bottlenecks of the previous. First, a sequential CPU baseline (V0) serves as our correctness oracle and performance lower bound. A naïve GPU implementation (V1) exploits antidiagonal parallelism with a fully materialized DP matrix and one kernel launch per antidiagonal tile. V2 reduces memory footprint by utilizing rolling 2-row buffers instead of the full matrix. Finally, V3 eliminates kernel launch overhead entirely with a single cooperative kernel, `grid.sync()` barriers, strided row assignment, and precomputed row-to-node lookups. V4 incorporates inter-read alignment along with intra-read, utilizing the GPU resources more completely. All implementations are available at `https://github.com/Elizabetht1/418-final-project/`.

## 2.2 Technologies

Our implementation uses CUDA 11.7 on an NVIDIA GeForce RTX 2080 GPU. Host code is written in C++17. Graph input uses the GFA file format, parsed on the CPU before upload to device memory.

## 2.3 V0: Sequential CPU Baseline

Our sequential CPU implementation follows the standard POA algorithm. Nodes are processed in topological order; for each node, the DP submatrix is filled row by row. Interior bases follow the affine Smith-Waterman recurrence directly. Boundary bases — the first base of each node — aggregate scores from the last row of all predecessor nodes before applying the recurrence. This implementation serves as our correctness oracle and performance lower bound, achieving $O(NM)$ time with no parallelism. The topological sort written for the CPU code will be used in all future versions — the alignment section of the algorithm will be parallelized.

## 2.4 V1: Antidiagonal Tiling (Naïve GPU)

Our first GPU implementation exploits antidiagonal independence. Cells sharing antidiagonal index $d$ = row + col have no dependencies on each other and can be computed simultaneously.

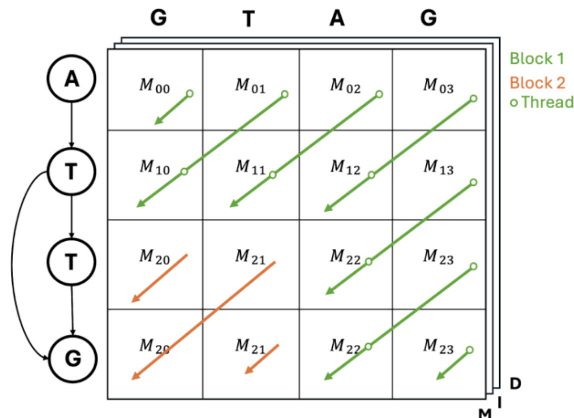


Figure 3: Antidiagonal parallelization of the affine-gap DP matrix. Rows correspond to flattened reference bases across graph nodes, columns to query read bases. Cells sharing an antidiagonal are data-independent and are computed simultaneously by individual CUDA threads (circles), partitioned across thread blocks (Block 1: green, Block 2: orange). The stacked layers represent the three DP matrices (`MAT_ALIGN`, `MAT_GAP_READ`, `MAT_GAP_REF`) for the affine recurrence.

**Thread and Block Mapping.** Each CUDA thread handles one cell on the current antidiagonal. Threads are organized into blocks of $T$ threads, each covering a contiguous tile of the antidiagonal. One kernel is launched per antidiagonal, for a total of $O(M+N)$ kernel launches.

**Memory Hierarchy.** The DP matrix and graph are stored in global memory. Intermediate results for a single thread are stored in registers.

There are two key disadvantages to this approach:

1. Multiple Kernel Launches. Each antidiagonal requires a separate kernel launch and

a `cudaDeviceSynchronize()` barrier before the next can begin, since each antidiagonal depends on the previous two. With $M + N - 1$ total antidiagonals, this means thousands of round-trips between host and device. Each kernel launch incurs a fixed overhead of several microseconds regardless of how much work is done. On short antidiagonals (near the corners of the matrix) most threads are idle and the launch cost dominates the compute cost.

2. Fully materialized DP matrix. The $M \times N$ DP matrix is stored in global memory, which is more expensive to access than local memory. The matrix stores three values per cell (M, I, D matrices). This costs $3 \cdot M \cdot N \cdot 2$ bytes with `int16` precision. For a graph with $N = 100{,}000$ bases and a read of length $M = 10{,}000$, this requires 6 GB, approaching the 8 GB limit of the RTX 2080 and leaving no headroom for real genomics workloads.

Profiling supports these results. Occupancy (amount of SMs used out of total capacity) is $<$ 20% even as read length increases to 1,000+ bp. Moreover, since we are constructing the entire DP, we are unable to increase read size to the point where all SMs are active.

| Metric | read_len = 512 | read_len = 2048 |
|---|---|---|
| Compute (SM) [%] | 0.03% | 0.15% |
| Memory [%] | 0.65% | 0.77% |
| Achieved Active Warps/SM | 2.25 | 6.01 |
| Achieved Occupancy | 7.03% | 18.78% |

Table 1: GPU kernel performance metrics for `alignerKernel` across read lengths.

Furthermore, the kernel uses entirely global memory accesses, which results in a memory-bound processes (see difference between compute % versus memory %).

### 2.5 V2: Antidiagonal Tiling with Rolling Buffers

V2 replaces the full DP matrix with rolling 2-row buffers, reducing memory from $O(NM)$ to $O(N \cdot |\text{nodes}|)$. Since cell $(i, j)$ depends only on the previous row, only two rows of state need to be retained at any time. A separate predecessor buffer stores the last-row scores of each node for use at boundary bases, indexed by node slot and column.

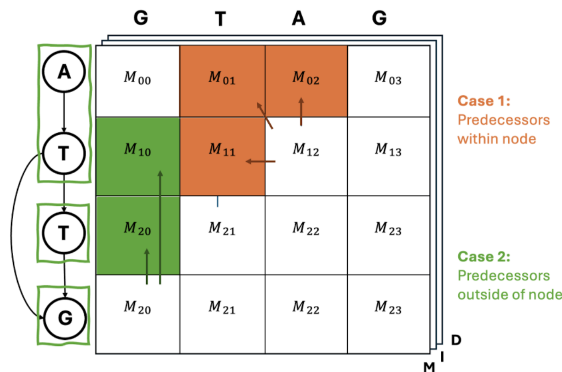


Figure 4: Two predecessor cases in the scrolling array optimization. Orange cells (Case 1) show an interior base whose predecessors lie on the immediately previous row, requiring only `dp_prev`. Green cells (Case 2) show the first base of a node whose predecessor comes from the last row of a graph-edge predecessor node, which may be arbitrarily far back — requiring the `node_bound` buffer.

**Memory Optimization: Scrolling Arrays** As Figure 4 illustrates, we consider two cases. For a base in the interior of a node, the only dependencies are on the previous row of the DP matrix. For the first base of a node, it must read scores from its predecessors, which may be arbitrarily far away in the graph. For interior bases, we only need to store the previous two rows. For boundary bases, we store the last row of every node in a separate `node_bound` buffer. The biological structure of variation graphs simplifies this: (a) the ratio of nodes to total base pairs is low, and (b) the branching factor at each node is low, so this buffer remains small in practice.

**Kernel Launch Overhead.** V2 processes one reference base row at a time, sweeping $2N$ antidiagonal kernel launches per row. This gives

$O(M \cdot N)$ total kernel launches — more than V1's $O(M)$ — because the rolling buffer design requires the host to synchronize after every row to scroll `dp_prev` $\leftarrow$ `dp_curr` before the next row can begin. The memory reduction comes at the cost of increased launch overhead, which V3 addresses.

**Memory Hierarchy.** All buffers (`dp_curr`, `dp_prev`, `node_bound`) are stored in global memory. Intermediate results for a single cell are stored in registers.

### 2.6 V3: Single Cooperative Kernel

V2 still incurs significant kernel launch overhead — one launch per antidiagonal per row, amounting to $O(M \cdot N)$ total launches. V3 eliminates this entirely with a single cooperative kernel launch using `cudaLaunchCooperativeKernel`.

**Cooperative Groups.** Rather than returning to the host between antidiagonals, the kernel owns all loops internally. A `grid.sync()` barrier replaces `cudaDeviceSynchronize()`, synchronizing all threads across the entire grid at the device level. This avoids the host-device round-trip cost entirely.

**Three Rotating Buffers.** Instead of `dp_curr` and `dp_prev`, V3 uses three rotating antidiagonal buffers indexed by $d \bmod 3$. Diagonal $d$ writes to buffer $d \bmod 3$ and reads from buffers $(d + 2) \bmod 3$ and $(d + 1) \bmod 3$. Memory remains $O(M)$ — three rows of $M$ values.

**Strided Row Assignment.** Threads are assigned rows with a stride pattern — thread $t$ handles rows $t, t + \text{stride}, t + 2 \cdot \text{stride}, \ldots$ — so a fixed grid size covers all $M$ rows regardless of graph size. Score tracking uses `atomicMax` on a device-side variable, eliminating the per-row `cudaMemcpy` back to the host from V2.

**Memory Hierarchy.** The three rotating buffers and predecessor buffer are stored in global memory. Intermediate results per cell are stored in registers.

**V3 Profiling.** Table 2 shows NCU metrics for `alignerKernel` on a 1024 bp read aligned to a 200K-base graph.

| Metric | Value |
|---|---|
| Achieved Occupancy | 100.00% |
| Theoretical Occupancy | 100.00% |
| Compute (SM) [%] | 11.72% |
| Memory [%] | 9.54% |
| DRAM Throughput | 0.01% |

Table 2: NCU metrics for V3 `alignerKernel` on a 1024 bp read against a 200K-base graph. Despite 100% warp occupancy, compute throughput is only 11.72% — threads are fully scheduled but idle at `grid.sync()` barriers between antidiagonals. This motivates V4's inter-read parallelism, which keeps threads productive by aligning multiple reads concurrently.

### 2.7 V4: Inter-read with Rolling Buffers

It is challenging to saturate available GPU resources without generating DP matrices that are prohibitively large for our target machine. Version 4 explores inter-read parallelism as a means to achieve better occupancy, exploiting that there are no dependencies between reads.

**Occupancy.** Fully utilizing GPU resources requires launching enough work to keep all SMs busy simultaneously. On the RTX 2080, which has 46 SMs, the number of concurrently resident blocks per SM is determined by shared memory demand. In one representative workload, the shared memory footprint of our kernel limited capacity to 3 blocks per SM, yielding an effective block size (EBS) of 138 blocks. Since each block processes one read, launching fewer than 138 reads leaves a significant fraction of SMs idle.
.

As Table 3 shows, compute and memory utilization scale predictably with the number of waves, reaching 29% at 1×EBS, 46% at 2×EBS, and plateauing near 46% beyond that. Achieved occupancy follows a similar trajectory, peaking at 68% before declining slightly at 3×EBS as scheduling pressure increases.

| Metric | 64 reads | 1×EBS (138) | 2×EBS (276) | 3×EBS (414) |
|---|---|---|---|---|
| Compute (SM) [%] | 15.73% | 29.37% | 45.86% | 46.08% |
| Memory [%] | 15.73% | 29.37% | 45.86% | 46.08% |
| Theoretical Occupancy | 75.00% | 75.00% | 75.00% | 75.00% |
| Achieved Occupancy | 35.09% | 52.52% | 68.35% | 60.81% |
| Waves | 0.46 | 1.00 | 2.00 | 3.00 |

Table 3: GPU kernel performance metrics for `alignerKernel` across read counts. EBS (Effective Block Size) = 138 blocks = 46 SMs × 3 blocks/SM.

**Synchronization Stalls.** The antidiagonal recurrence imposes an unavoidable inter-warp dependency at every step: no cell on diagonal $d$ can be computed until all cells on diagonals $d-1$ and $d-2$ are complete, requiring a barrier synchronization after each antidiagonal. To quantify the cost of these barriers, we profiled three configurations: the original kernel, a version with redundant synchronization removed (minimal synchronization needed for correctness), and a hypothetical no-sync baseline.

| Metric | Original | Reduced Sync | No Sync |
|---|---|---|---|
| Warp Cycles Per Issued Instruction | 15.13 | 11.95 | 6.28 |
| Barrier Stall Cycles | 7.0 | 5.8 | 0.0 |
| Barrier Stall [%] | 46.0% | 48.9% | 0.0% |
| Eligible Warps Per Scheduler | 0.37 | 0.60 | 0.94 |
| Issued Warp Per Scheduler | 0.27 | 0.39 | 0.54 |
| Avg. Not Predicated Off Threads | 20.27 | 20.45 | 21.09 |

Table 4: Warp statistics across sync optimization stages.

As Table 4 shows, eliminating redundant synchronization reduces warp cycles per issued instruction from 15.13 to 11.95 and increases eligible warps per scheduler from 0.37 to 0.60 — meaningful improvements, but far from the no-sync ceiling of 6.28 cycles and 0.94 eligible warps. This demonstrates that while unnecessary synchronization is worth removing, the dominant cost is structural: the antidiagonal dependency itself enforces a serialization that no amount of synchronization pruning can eliminate.

Despite the synchronization overhead, our memory access pattern is well-optimized.

As Table 5 shows, the L1/TEX hit rate is 98.62% and DRAM throughput is near zero at 0.11%, confirming that the kernel operates almost entirely out of shared memory with negligible global memory traffic.

| Metric | Value |
|---|---|
| *Cache* | |
| L1/TEX Hit Rate | 98.62% |
| L2 Hit Rate | 66.63% |
| DRAM Throughput | 0.11% |
| *Shared Memory* | |
| Dynamic Shared Memory Per Block | 20.21 KB |
| Block Limit (Shared Mem) | 3 blocks/SM |

Table 5: Memory access characteristics for `alignerKernel` (no-sync version, 414 reads, N=64).

The remaining memory latency is therefore internal to shared memory management. The primary source of waste is that dynamic shared memory per block is allocated to accommodate the longest node in the graph — 20.21 KB in our profiled workload — meaning that blocks processing shorter nodes carry unused allocation, directly limiting the block capacity per SM to 3 and suppressing occupancy. In workloads with highly uneven node length distributions, this effect is particularly pronounced.

**Warp Divergence.**

The final source of inefficiency is branch divergence within the kernel. Threads on the same warp follow divergent execution paths depending on whether they are processing interior bases, boundary bases, or out-of-bounds cells — each of which requires different logic and memory access patterns. The hardware resolves this by masking off divergent threads, but at the cost of reduced utilization: on average, only 20 of 32 threads per warp are active at any given instruction. While it is possible to reduce divergence by restructuring the boundary base logic, the fundamental branching between interior and predecessor-dependent cells is an intrinsic property of the POA recurrence and cannot be fully eliminated without a more substantial algorithmic redesign.

# 3 Results

## 3.1 Experimental Setup

**Performance Metric.** We report throughput in GCUPS (Giga Cell Updates Per Second), cal-

culated as:

$$\text{GCUPS} = \frac{|\text{graph bases}| \times |\text{read length}|}{t \times 10^9}$$

where $t$ is the kernel execution time in seconds. Kernel time is measured using CUDA events wrapping the alignment loop, excluding memory allocation and data transfer. Speedup is reported relative to our sequential CPU baseline (V0), measured with `std::chrono`.

**Profiling.** Per-kernel metrics (cache hit rates, achieved occupancy, warp divergence) are collected using NVIDIA Nsight Compute (NCU), which profiles a representative kernel launch. Since these are per-kernel properties they are valid from a single sample. Note: for version 4, we report metrics over a batch of reads as opposed to single read

## 3.2 Test Inputs

We generate synthetic sequence graphs and reads following the methodology described in the HGA paper [8]. Our synthetic graphs model the most common structural variant in population genomes: the single-nucleotide polymorphism (SNP). SNPs encode small local variations. We parameterize graph generation along three axes: total node count, average node length, and branching factor, allowing us to isolate the effect of each structural property on alignment performance. Reads are generated by sampling paths through the graph and introducing random substitutions to simulate sequencing error. For our performance evaluation we vary read length as the primary read-side parameter; error rate is held constant. Error rate may impact correctness, but does not alter the runtime of our algorithim.

## 3.3 Effect of Workload / Problem Size: Read Length

**Intra-Read**

**Inter-Read**

We expect alignment throughput in GCUPS to increase with read length, since longer reads expose more intra-read parallelism along the antidiagonal. In the ideal case this relationship would be linear — a perfectly parallel implementation would absorb additional read length at no extra time cost, keeping wall-clock time constant as GCUPS scales proportionally.

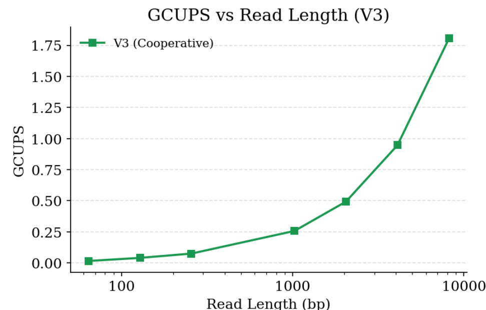


Figure 5: GCUPS vs. Total Read Length for Version 4 (Interread)

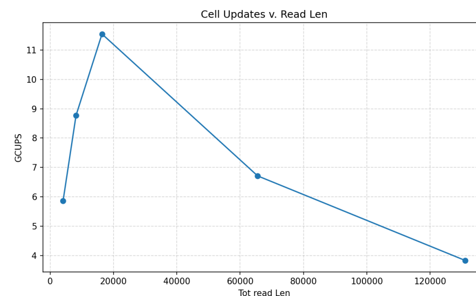


Figure 6: GCUPS vs. Total Read Length for Version 4 (Interread)

Our results follow this trend for shorter reads: throughput increases across the first few data points, albeit sublinearly, reflecting the overhead of kernel launches and boundary synchronization that prevent full utilization. Beyond a critical read length, however, we observe a sharp drop in throughput.

This inflection point corresponds to the average read length in the batch exceeding the thread block size T. At this threshold, each antidiagonal can no longer be processed in a single pass, and the read must instead be handled in chunks of width T. Each additional pass introduces nontrivial overhead: threads at the chunk boundary must reload the last column of the previous pass from memory, and threads near the ends of short

antidiagonals remain idle, reducing effective occupancy. The result is a superlinear increase in wall-clock time beyond this threshold, visible as a cliff in the GCUPS curve.

### 3.4 Performance vs Branching Factor

The branching factor of a graph is defined as the average number of incoming edges per node, and controls how many predecessor nodes a boundary base must aggregate scores from.

**Intra-Read (V3).** Figure 7 shows GCUPS as a function of branching factor for V3. We expect GCUPS to degrade with higher branching factor for two reasons: first, boundary bases must read from `node_bound` for each predecessor, increasing global memory traffic; second, warps that straddle boundary and interior bases serialize due to divergence. However, results are mixed and noisy. GCUPS does decrease modestly with higher branching factor, consistent with our expectations, though the effect is small. We attribute this to the prevalence of SNPs in variation graphs — short predecessor nodes limit the cost of additional predecessor lookups, dampening the impact of higher branching on overall throughput.

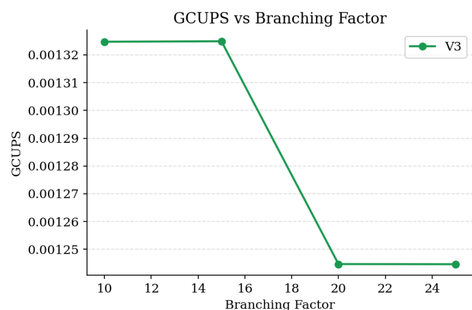


Figure 7: GCUPS vs. branching factor for V3 (intra-read).

**Inter-Read (V4).** Figure 8 shows execution time as a function of branching factor for V4. Results do not show a clear monotonic trend, suggesting that for inter-read alignment, branching factor has a weak effect on overall throughput. Since V4 blocks are fully independent, the cost of boundary base predecessor lookups is localized to individual warps and does not create global bottlenecks. Variability in the results likely reflects differences in graph topology between synthetic inputs rather than a consistent effect of branching factor.

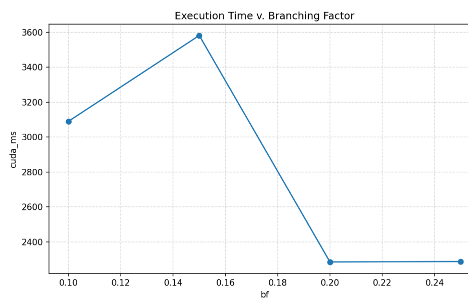


Figure 8: Execution time vs. branching factor for V4 (inter-read).

### 3.5 Performance vs Average Node Length

**Inter-Read** Figure 9 shows alignment time as a function of average node length, with total reference bases held constant. As node length increases, the graph contains fewer but longer nodes. The sequential baseline (red dashed) remains roughly constant across node lengths, since the total number of DP cells is fixed. GPU runtime decreases significantly with longer nodes, improving by over an order of magnitude from 57 bp to 1069 bp average node length.

This result is explained by idle thread utilization. Each thread block processes one node at a time, sweeping antidiagonals of width up to $\min(\text{node_len}, N)$. Short nodes produce narrow antidiagonals where most threads are idle — a node of length 57 bp can only keep 57 threads active per diagonal regardless of block size. Longer nodes sustain wider antidiagonals for more diagonals per kernel, amortizing the fixed synchronization cost of `grid.sync()` over more useful work per barrier.

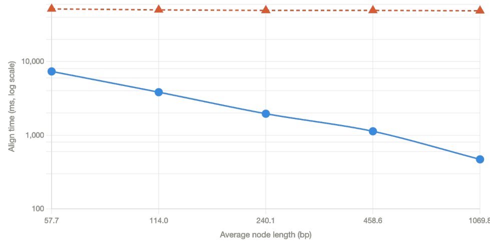


Figure 9: Alignment time (log scale) vs. average node length, with total reference bases held constant. Sequential time (red dashed) is flat while GPU time (blue) decreases sharply, showing that longer nodes improve GPU utilization by sustaining wider antidiagonals.

### 3.6 Inter-Read Performance

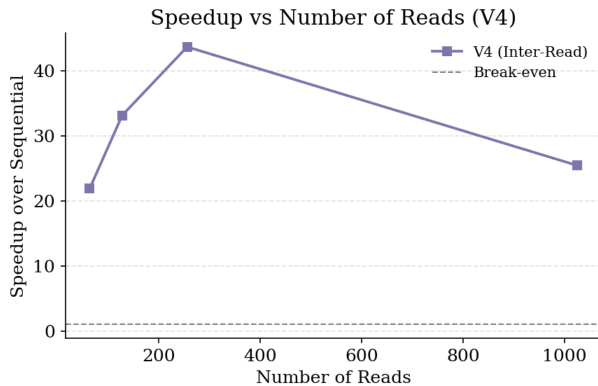


Figure 10: Speedup over sequential baseline vs. number of reads for V4 (inter-read). Peak speedup of 43.7× is achieved at 256 reads.

| Reads | Read Len (bp) | GCUPS | Speedup |
|---|---|---|---|
| 64 | 4,096 | 5.86 | 21.9× |
| 128 | 8,192 | 8.77 | 33.1× |
| 256 | 16,384 | 11.55 | 43.7× |
| 1024 | 65,536 | 6.71 | 25.5× |

Table 6: V4 performance varying number of reads (fixed graph, 834 nodes).

| Branching Factor | GCUPS | Speedup |
|---|---|---|
| 10 | 4.25 | 16.2× |
| 15 | 3.66 | 14.1× |
| 20 | 5.75 | 22.0× |
| 25 | 5.75 | 21.9× |

Table 7: V4 performance varying branching factor (1024 reads).

Table 6 and Figure 10 show V4 performance across varying batch sizes. Speedup increases from 21.9× at 64 reads to a peak of 43.7× at 256 reads, before declining to 25.5× at 1024 reads. The increase from 64 to 256 reads reflects better GPU utilization as more blocks become resident on the SMs simultaneously: at 64 reads, many SMs sit idle between waves, while at 256 reads the GPU approaches full occupancy. The decline at 1024 reads is consistent with memory pressure from the larger batch: each read requires its own `node_bound` buffer, and at 1024 reads the aggregate buffer size begins to exceed L2 cache capacity, increasing DRAM traffic.

Table 7 shows performance across varying branching factors. Results are noisy and do not show a clear monotonic trend, consistent with our intra-read analysis. SNP-dominated variation graphs limit the overhead of additional predecessor lookups, dampening the effect of branching factor on throughput.

Peak throughput of 11.55 GCUPS is achieved at 256 reads. To contextualize this against the literature, HGA [8] reports 93 GCUPS on a single RTX 2080 Ti using inter-sequence parallelism with one thread per read. Our design assigns one block per read, combining inter-read parallelism with intra-read antidiagonal parallelism, and uses the affine gap penalty model which requires three DP matrices versus HGA's linear gap model. The gap in absolute GCUPS reflects both the higher per-cell cost of the affine model and the single-GPU constraint of our implementation.

# 4 Discussion

**Algorithim Decomposition** Our algorithm decomposes into two execution phases: inte-

rior base processing, where cells follow the standard affine Smith-Waterman recurrence within a node, and boundary base processing, where the first base of each node aggregates scores from all predecessor nodes via the node boud buffer. The majority of execution time is concentrated in the interior phase, where the dominant cost is not arithmetic but synchronization.

**Bottlenecks.**

**1. Synchronization Overhead.** The primary bottleneck in our intra-read kernel is synchronization overhead. Profiling reveals approximately 11 cycles per issued warp instruction (11 times worse than the optimum 1 cycle / instruction). This indicates that warps spend the majority of their time stalled at barriers rather than doing useful work. This is a fundamental consequence of antidiagonal parallelism: every antidiagonal step requires a grid-wide synchronization before the next can begin, and the inherent serialization of these barriers leaves the scheduler with few eligible warps to hide the latency.

**2. Shared Memory Allocation.** Finally, shared memory allocation is sized to the longest node in the graph, meaning that for workloads with highly uneven node length distributions, a significant fraction of the shared memory budget is reserved but unused for shorter nodes, limiting the number of blocks that can co-reside on each SM.

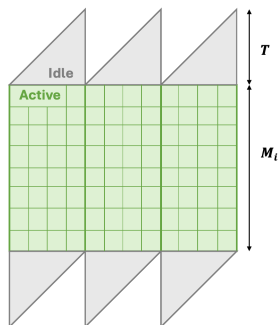


Figure 11: Illustration of Idle Thread Time due to Warp Divergence, where $M_i$ is the node length and $T$ is the thread block size.

**3. Warp Divergence.** A secondary bottleneck is warp divergence: because antidiagonals vary in length across the DP matrix (narrowest at the corners and widest at the center) threads assigned to out-of-bounds cells are masked off by the hardware rather than retired, reducing effective occupancy to roughly 21 of 32 threads per warp on average. See Figure 11.

Taken together, these bottlenecks reflect a fundamental constraint of the antidiagonal approach: the degree of exploitable parallelism is upper-bounded by the width of the current antidiagonal, which varies continuously throughout execution and is never simultaneously maximal for all threads.

**Opportunities for Improvement** The largest remaining source of inefficiency is branch divergence within the kernel. Our current implementation uses deeply nested conditionals for bounds checking and boundary base detection, which causes threads within the same warp to follow divergent execution paths and stall. Merging redundant checks or restructuring the loop to separate interior and boundary bases into distinct kernel phases would reduce the number of masked-off threads per warp and improve effective occupancy.

A second avenue is memory layout: shared memory allocation remains sized to the maximum node length, which artificially caps the number of blocks that can co-reside per SM even when most nodes are shorter. A dynamic or per-node allocation strategy — or packing the shared memory buffer more aggressively — would increase occupancy in the inter-read case and better utilize the available SM capacity. Further, we have discussed the inherent limitations of antidiagonal parallelism.

**Machine Target Choice.** Our results suggest that a CPU implementation would likely be more competitive for workloads with short reads, short nodes, or small graph sizes. In these regimes, antidiagonals are narrow, GPU thread utilization is low, and the overhead of kernel launches and synchronization barriers outweighs the benefit of parallelism — conditions under which a well-

vectorized CPU implementation such as abPOA would excel. More broadly, our profiling reveals that memory capacity was the binding constraint on the GPU rather than compute throughput: the RTX 2080 memory limit was approached under large inputs but SM utilization remained well below saturation. This is an artefact of the nature of the problem and it's dependencies however, and this was exasperated by our decision to implement the more biologically relevant but also more memory hungry affine gap penalty.

**Future Work.** Two optimizations from the literature remain unimplemented. First, replacing our pointer-of-pointers predecessor structure with a CSR-style flat array (analogous to HGA's GCSR [8]) would reduce global memory loads at boundary bases. Second, adopting abPOA's adaptive banded DP [7] would restrict computation to a diagonal band around the likely alignment path, reducing the number of active cells per antidiagonal and improving utilization.

## Work Distribution

Both authors contributed equally to this project (50%–50%). We alternated implementation responsibilities across versions, with each author independently developing successive iterations of the aligner. After each iteration, we met to discuss performance bottlenecks and correctness concerns, using these discussions to motivate the design of the next version. The author responsible for each implementation was also responsible for collecting the corresponding profiling data. Report writing ( including the milestone reports and this final paper) was done collaboratively.

## References


[1] G. Sirugo, S. M. Williams, and S. A. Tishkoff. *The Missing Diversity in Human Genetic Studies.* Cell, 177(1):26–31, 2019. https://doi.org/10.1016/j.cell.2019.02.048

[2] W. Liao et al. *A draft human pangenome reference.* Nature, 602:51–58, 2023. https://doi.org/10.1038/s41586-023-05896-x

[3] T. F. Smith and M. S. Waterman. *Identification of Common Molecular Subsequences.* Journal of Molecular Biology, 147(1):195–197, 1981. https://doi.org/10.1016/0022-2836(81)90087-5

[4] O. Gotoh. *An Improved Algorithm for Matching Biological Sequences.* Journal of Molecular Biology, 162(3):705–708, 1982. https://doi.org/10.1016/0022-2836(82)90398-9

[5] C. Lee, C. Grasso, and M. F. Sharlow. *Multiple Sequence Alignment Using Partial Order Graphs.* Bioinformatics, 18(3):452–464, 2002. https://doi.org/10.1093/bioinformatics/18.3.452

[6] R. Vaser, I. Sović, N. Nagarajan, and M. Šikić. *Fast and accurate de novo genome assembly from long uncorrected reads.* Genome Research, 27(5):737–746, 2017. https://doi.org/10.1101/gr.214270.116

[7] Y. Gao et al. *abPOA: an SIMD-based C library for fast partial order alignment using adaptive band.* Bioinformatics, 37(15):2209–2211, 2021. https://doi.org/10.1093/bioinformatics/btaa963

[8] Z. Feng and Q. Luo. *Accelerating Sequence-to-Graph Alignment on Heterogeneous Processors.* In Proceedings of ICPP 2021. https://doi.org/10.1145/3472456.3472505

[9] *Sequence graphs and assemblies around immune system loci C4, LRC, MHC.* Zenodo record 6056061. https://zenodo.org/records/6056061