AnySeq/GPU: A Novel Approach for Faster Sequence Alignment on GPUs

Ayush Raina, 22148

April 11, 2025

Overview

The paper introduces **AnySeq/GPU**, a fast sequence alignment library optimized for GPUs. It improves performance by reducing memory accesses using **warp shuffles** and **half precision arithmetic**, and it uses the **AnyDSL compiler** for efficient GPU code generation. The library achieves up to **3.8 TCUPS**, outperforming previous GPU-based aligners like GASAL2 and NVBIO by up to **19.2x**. It runs efficiently on both NVIDIA and AMD GPUs and is open-source.

Motivation

Sequence alignment is essential in bioinformatics, especially with the rapid growth of **Next-Generation Sequencing (NGS)** data. Traditional alignment algorithms like **Needleman-Wunsch** and **Smith-Waterman** use dynamic programming (DP) and have high time complexity, making them slow for large-scale NGS data (both short and long reads). While many alignment tools have been developed for **CPUs, GPUs, and FPGAs**, current GPU-based tools often suffer from: Inefficient memory access patterns, Lack of performance portability across different sequence lengths and alignment types, Limited optimization to only specific CUDA-enabled GPUs. To overcome these limitations, the authors propose AnySeq/GPU, an optimized GPU-based extension of the AnySeq library.

Methodology and Key Contributions

1. **Unique Parallelization Strategy:** Introduces a new fine grained GPU Parallelization technique using **warp intrinsics** and a novel **DP Matrix Partitioning** scheme that handles batches with varying sequence lengths efficiently.

2. **Hardware Independent Implementation:** It leverages **AnyDSL**, a domain specific language framework which is used to generate optimized kernels for both **NVIDIA** and **AMD** GPUs.

3. **High Performance:** Achieves over **80%** of peak GPU Performance across a range of alignment types with speedups of **3.6x to 19.2x** compared to existing GPU-based libraries.

Sequence Alignment

Given two DNA Sequences $Q = (q_1, q_2, ..., q_m)$ and $S = (s_1, s_2, ..., s_n)$, the goal of sequence alignment is to compute the best match between them using **Dynamic Programming (DP)**. The DP Matrix $H(i, j)$ stores the optimal alignment score between the $Q(1 : i)$ and $S(1 : j)$ subsequences and is recursively defined as:

$$H(i,j) = \max \begin{cases} H(i-1, j-1) + \sigma(q_i, s_j) \\ H(i-1, j) - \alpha \\ H(i, j-1) - \alpha \end{cases} \tag{1}$$

where α is the constant gap penalty and $\sigma(q_i, s_j)$ is the substitution cost function. The final alignment score is found in $H(m, n)$. In the original paper, other types of gap penalty scheme can be found, but we will consider the constant gap penalty scheme for simplicity.

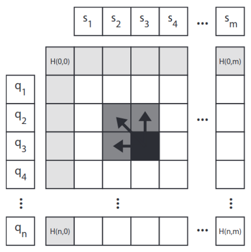


Figure 1: Dynamic Programming Matrix for Sequence Alignment

Mapping Matrix Cells and Sequences to Threads

Algorithm assigns each warp (32 threads $T_0, T_1, ..., T_{31}$) to compute pairwise alignment between seuqence Q of length m and sequence S of length n. The DP Matrix $\mathbf{H} \in \mathbb{R}^{(m+1)\times(n+1)}$ is computed in **tiles**, where each thread T_t is responsible for computing k columns of **H** giving the relation $n = kp - 1$. The matrix is filled diagonally over $m + p$ iterations, where in each iteration i, thread T_t computes $(i - t)^{th}$ row of its k assigned columns, according to recurrence relation given.

To reduce memory traffic, DP values are stored in **local-thread registers** and are shared across other threads in the warp using **warp shuffle instructions** to access neighbour's values. This allows for fast intra-warp communication and minimizes the need for global memory accesses. Subject characters S are loaded once by each thread and used to construct a scoring profile in shared memory. This stores $\sigma(q_i, s_j)$ for all $s_j \in S, q_i \in Q$ and enables efficient lookups.

Query characters Q, whose values change across rows, are loaded only every p iterations and then propagated between threads using registers and warp shuffles **(e.g., shfl_up and shfl_down)** to avoid global memory loads. Each thread maintains two registers c_{q0} and c_{q1}, where c_{q0} stores the current query character Q_{i-t-1} and c_{q1} stores the next query character for the next iteration. Only in the iterations where $i \mod p = 0$, each thread loads a new character q_{i+t} from memory into c_{q1}. In all other iterations, threads update c_{q0} and c_{q1} using warp shuffles.

Handling Long Sequences with Multi Stage Partioning

In modern GPUs, the number of registers per thread is limited (For NVIDIA, its 255), which restricts the size of k i.e number of columns that each thread is going to process. To support for the longer sequences, the algorithm partitions the DP Matrix into l non overlapping submatrices, each of size: $(m + 1) \times (\frac{n+1}{kp})$. These submatrices are computed sequentially from left to right in l stages.At the end of each stage, the rightmost DP values computed by thread T_{31} are stored in shared or global memory and these values are read by T_0 in the next stage. To minimize memory access overhead, this **stage-wise boundary communication** also uses warp shuffles where possible, similar to how query characters are propagated.

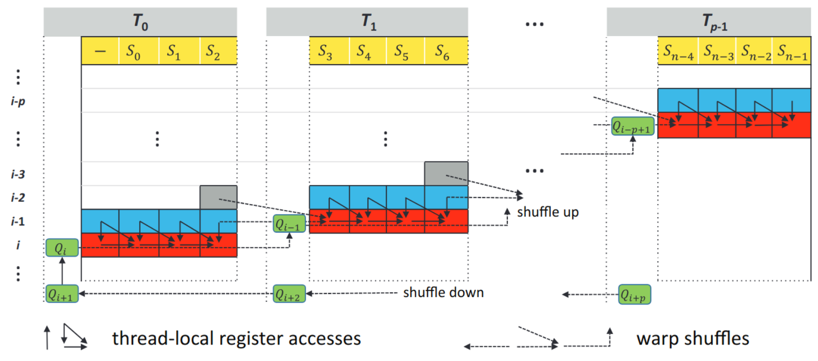
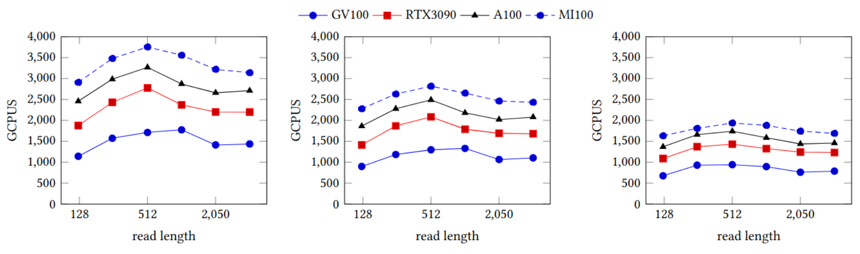


Figure 2: Partitioning of DP Matrix for Long Sequences

Automatic GPU Kernel Generation using AnyDSL

AnySeq/GPU uses user-guided partial evaluation (PE) within the AnyDSL framework to automatically generate GPU kernels that are specialized for specific alignment types such as local or global alignments, hardware architectures, sequence lengths, and scoring schemes such as linear or affine. This approach enables high performance while avoiding code duplication.



(a) global alignment, linear penalties **(b) global alignment, affine penalties** **(c) local alignment, affine penalties**

Figure 4: Median speed in GCPUS achieved for pairwise score computation using AnySeq/GPU for reads of varying lengths for a) global alignment, linear penalties; b) global alignment, affine penalties; c) local alignment, affine penalties.

Figure 4 reports the median performance in GCUPS (giga cell updates per second) for pairwise alignment using AnySeq/GPU across varying read lengths. Three alignment modes are evaluated:

(a) Global alignment with linear gap penalties
(b) Global alignment with affine gap penalties
(c) Local alignment with affine penalties

For all modes, performance peaks around read length ~ 512, where the throughput reaches up to 3,600 GCUPS on AMD MI100. Linear penalties (panel a) allow simpler recurrences, enabling higher throughput than affine gaps. Among GPUs, MI100 > A100 > RTX 3090 > GV100 in GCUPS, reflecting differences in memory bandwidth and compute power.

Critical Analysis

Strengths:
AnySeq/GPU introduces a warp-level parallelization strategy that leverages GPU registers and warp shuffles for efficient dynamic programming (DP) matrix computation. It supports both short and long reads, achieves high throughput (up to 3.8 TCUPS), and maintains >80% hardware efficiency across different GPUs. Its use of partial evaluation allows architecture-specific optimization while keeping the codebase general and portable.

Why these contributions matter:
Sequence alignment is a core step in many bioinformatics pipelines, and optimizing it for modern GPU hardware directly translates to faster genome analysis. Unlike earlier tools optimized only for short reads or specific architectures, AnySeq/GPU generalizes across read lengths and GPU vendors. Its in-register design and architecture-agnostic implementation enable portable performance without compromising flexibility in scoring schemes or alignment types.

Weaknesses:
There's no in-depth discussion of the correctness of alignments (e.g., compared to ground-truth alignments) or potential issues from using half precision arithmetic in scoring. This could be critical when porting the method to clinical or variant-sensitive pipelines.

Future Work: The work can be extended by adding *multi-GPU support*, enabling *runtime kernel specialization*, and integrating the alignment kernel into complete pipelines like `minimap2` or `GATK`. The same parallel pattern could also be applied to other DP problems like RNA folding or Viterbi decoding.