AGAThA: Fast and Efficient GPU Acceleration of Guided Sequence Alignment for Long Read Mapping

Seongyeon Park
Seoul National University
Seoul, South Korea
syeonp@snu.ac.kr

Junguk Hong
Seoul National University
Seoul, South Korea
junguk16@snu.ac.kr

Jaeyong Song
Seoul National University
Seoul, South Korea
jaeyong.song@snu.ac.kr

Hajin Kim
Yonsei University
Seoul, South Korea
kimhajin@yonsei.ac.kr

Youngsok Kim
Yonsei University
Seoul, South Korea
youngsok@yonsei.ac.kr

Jinho Lee*
Seoul National University
Seoul, South Korea
leejinho@snu.ac.kr

Abstract

With the advance in genome sequencing technology, the lengths of deoxyribonucleic acid (DNA) sequencing results are rapidly increasing at lower prices than ever. However, the longer lengths come at the cost of a heavy computational burden on aligning them. For example, aligning sequences to a human reference genome can take tens or even hundreds of hours. The current de facto standard approach for alignment is based on the guided dynamic programming method. Although this takes a long time and could potentially benefit from high-throughput graphic processing units (GPUs), the existing GPU-accelerated approaches often compromise the algorithm's structure, due to the GPU-unfriendly nature of the computational pattern. Unfortunately, such compromise in the algorithm is not tolerable in the field, because sequence alignment is a part of complicated bioinformatics analysis pipelines. In such circumstances, we propose AGAThA, an exact and efficient GPU-based acceleration of guided sequence alignment. We diagnose and address the problems of the algorithm being unfriendly to GPUs, which comprises strided/redundant memory accesses and workload imbalances that are difficult to predict. According to the experiments on modern GPUs, AGAThA achieves 18.8× speedup against the CPU-based baseline, 9.6× against the best GPU-based baseline, and 3.6× against GPU-based algorithms with different heuristics.

*Corresponding author.

1 Introduction

Genome sequence analysis has largely impacted our lives, from aiding medical fields to others such as epidemiology [43], agriculture [7], and even basic science [51]. Such analysis is made possible by sequencing, which generates sequences called *reads* from strands of deoxyribonucleic acid (DNA) extracted from specimens. Although there has been a wide variety of sequencing techniques, the recently introduced third-generation sequencing (TGS) technique produces very long reads compared to its predecessors. While previous generation sequencing generated short reads of around 150~300 bps (base pairs), TGS can produce reads longer than an average of 10 kbps and higher [15, 34]. By generating longer and high-quality reads, TGS sheds light on new attributes and genomic mutations that were difficult to spot before.

However, this drastic increase in size comes at the cost of sharply increased time spent on the sequence analysis. The major essential step of the sequence analysis is read alignment, which aligns reads (i.e., *queries*) to reference genomes to find the location of each read within the whole DNA [3]. Due to the larger data size, prior read alignment algorithms struggle to process reads with acceptable low latency.

There are two de facto standard methods for aligning reads: Minimap2 [23] for longer reads from TGS, and BWA-MEM [22] for shorter reads from the previous generation.

At their cores, they form a very similar structure, based on a *guided dynamic programming* approach. Despite being highly optimized, they exhibit very long execution time even with modern multi-core CPUs. For example, [2] shows that mapping TGS reads to the entire human genome (3.2 GB) using Minimap2 can take 49 hours. To amend this, there have been several attempts to accelerate it on powerful GPUs.

However, to the extent of our knowledge, existing attempts compromise the structure [12] or target different alignment algorithms [1, 42, 57]. This is problematic because read alignment is not a stand-alone application, but an early part of complicated bioinformatics pipelines. Any change in this step would require an immense amount of verification on the entire pipeline. Therefore, the exactness of the algorithm cannot be traded off for speedup. This indicates that there is an urgent need for accelerating guided dynamic programming on modern GPU architectures.

We reveal that the difficulty in implementing the guided programming algorithm lies in the large number of random memory accesses and the severe dynamic workload imbalance caused by the guided alignment algorithm. Based on this diagnosis, we propose AGAThA[1], a GPU acceleration method for an exact and fast implementation of the guided alignment algorithm. To the best of our knowledge, our method is the first to accelerate the exact reference algorithm, achieving both exactness and speed. First, we propose an efficient scheme to calculate the termination condition of the guided alignment. The proposed scheme transforms the memory access patterns to be more sequential and coalesce better on the GPU memory hierarchy. Second, we propose a tiling scheme for the solution space to reduce the number of redundant memory accesses and unnecessary computations, which both depend on the execution order. Third, we design a method for mitigating the workload imbalance that is difficult to predict at both intra- and inter-warp levels. At the intra-warp level, we utilize a form of work stealing that is tightly coupled with the sequence alignment algorithm. At the inter-warp level, we observe a long-tail-like distribution of the workloads and devise a way to group them such that each group contains a similar amount of workload.

Our contributions can be summarized as follows:

- We propose AGAThA, the first exact GPU acceleration of the reference guided alignment. AGAThA not only exactly accelerates said algorithm but also significantly outperforms existing methods.
- We devise two schemes, rolling window and sliced diagonal, to efficiently calculate termination conditions.
- We devise subwarp rejoining and uneven bucketing to address the intra-warp and inter-warp load imbalance.
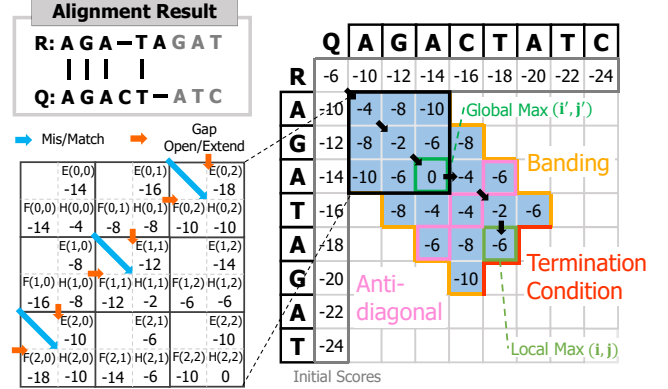

Figure 1. Guided sequence alignment.

- We evaluate AGAThA against numerous baselines to show 9.6× speedup against the best GPU baseline implementations that target guided dynamic programming, and 3.6× against other GPU-based heuristics.

2 Preliminary

2.1 Sequence Alignment

Problem Definition. The sequence alignment problem is a variant of approximate string matching. Given a pair of *reference* and *query* strings composed of five literals 'A', 'G', 'C', 'T', and 'N', it has to score how similar the two inputs are. Unlike exact string matching problems where the difference of a single character means that the pairs do not match, approximate string matching has to track more possibilities of insertion (query string has an extra character compared to the reference), deletion (query string has one less character), or a simple mismatch as depicted in the top left of Figure 1. Additionally, a gap can be defined as one or more continuous insertions/deletions. A gap has to be first initiated at a certain cell with an insertion/deletion ('gap open') and can be extended by adjacent insertions/deletions ('gap extend'). As a result, it outputs the alignment score, which represents the magnitude of similarity between the two sequences.

Dynamic Programming Approach. The sequence alignment problem is often handled with dynamic programming [36, 48]. As shown in Figure 1, it involves filling a two-dimensional score table (right) from the reference (**R**) and query (**Q**), whose computational and space complexity is $O(N^2)$. Each cell in position (i, j) represents $H(i, j)$, the best score that can be obtained by evaluating all possibilities until the i-th character of the reference string and the j-th character of the query string. $H(i, j)$ is recursively defined as:

$$H(i, j) = max\{E(i, j), F(i, j), H(i-1, j-1) + S(R[i], Q[j])\}, \quad (1)$$

$$E(i, j) = max\{H(i-1, j) - \alpha, E(i-1, j) - \beta\}, \quad (2)$$

$$F(i, j) = max\{H(i, j-1) - \alpha, F(i, j-1) - \beta\}. \quad (3)$$

$S(R[i], Q[j])$ compares the i-th reference character and the j-th query character and returns positive on a match (e.g., +2) and negative on a mismatch (e.g., −4). E and F are additional scores kept for tracking gaps, each storing
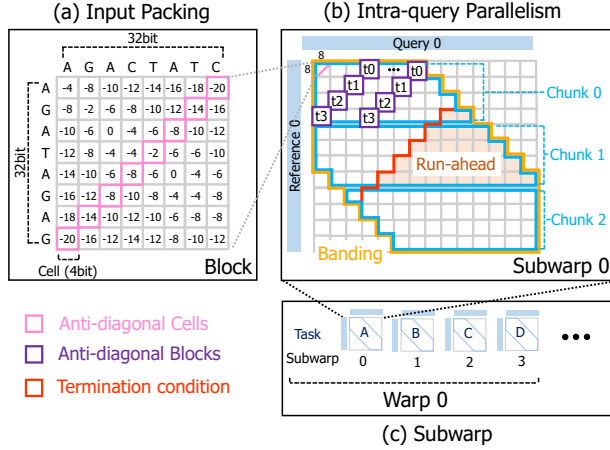
Figure 2. Existing GPU-accelerated sequence alignments.

deletions and insertions. α (e.g., 4) and β (e.g., 2) are gap opening and gap extending penalties, respectively.

From the above equations, we can find that calculating a score in the cell has dependencies on the cells from the top ($(i-1, j)$, Equation (2)), left ($(i, j-1)$ Equation (3)), and top-left ($(i-1, j-1)$, Equation (1)) values in the table, as shown in Figure 1 (bottom-left). These necessary values are referred to as *intermediate values*, and lead to the popular anti-diagonal parallelism. By defining anti-diagonal cell group (pink cells) as a group of cells that have the same index sum (e.g., $H(i-1, j)$ and $H(i, j-1)$ are on anti-diagonal $(i+j-1)$), there is no dependency between calculating the score of cells in the same anti-diagonal. Finally, the black arrows between cells represent the locations of mis/matches and gaps to reach the maximum score in the last anti-diagonal, which is equivalent to the alignment result on the top-left of Figure 1.

Guiding Strategy. For long input reads, the aforementioned alignment algorithm is known to return many false-positive matches. It might find a streak of matches with large offsets (e.g., the beginning of reference against the end of the query), or after too many mismatches. Those are often reported to be false positives even when they output high scores [23]. Because of this, the guiding strategy aims to filter false positive matches based on heuristics as shown in Figure 1.

Banding (yellow) is based on the idea that if there are too many insertions or deletions, it would be a false positive. For k-banding [13, 22, 23, 54], only a diagonal band with band width[2] k (3 in this example) is calculated and the rest are disregarded. In addition, *termination condition* (red) limits too many mismatches. If the difference between the global maximum score (dark green) and the current score (light green) becomes larger than a threshold, the termination condition is met. With some variants, a widely adopted form of termination condition [18, 23, 45] is:

$$\exists c < len(query) + len(ref), \quad \text{which satisfies} \tag{4}$$

$$i' < i, j' < j, H(i', j') - H(i, j) > Z + \beta \cdot |(i - i') - (j - j')|, \tag{5}$$

---

[2]We use the term "band width" to represent the width of the diagonal band in the score table, which is different from "memory bandwidth".

$$\text{where} \quad (i, j) = \underset{i+j=c}{\arg\max} \, H(i, j), \quad \text{(Local max.)} \tag{6}$$

$$(i', j') = \underset{i'+j'<c}{\arg\max} \, H(i', j'). \quad \text{(Global max.)} \tag{7}$$

In Equation (5), Z is an algorithm-specific and user-defined threshold, and β is the gap extension score from Equation (1). For CPUs, this technique not only reduces false positives but also increases throughput despite the overhead of checking conditions, as it can stop calculation accordingly. However, this causes a huge performance overhead on GPUs (see § 3.1), likely explaining the absence of an exact implementation.

2.2 State-of-the-art GPU Acceleration

In this section, we explain the techniques used in the state-of-the-art methods [1, 5, 42] for accelerating sequence alignment on GPUs with CUDA support.

Input Packing. Input packing [1] can help deal with memory bandwidth bottlenecks in GPU kernels. Because there are only five literals in genome sequences, four bits suffice for encoding each literal. Since GPUs typically use 32-bit words, sequences are packed with 8 literals per word. To efficiently process this, the score table is configured in units of **block**s comprising 8×8 **cell**s, which forms the smallest unit for workload distribution as depicted in Figure 2 (a).

Intra-query Parallelism. Intra-query parallelism [5, 42] is an essential technique to exploit massive parallelism on the GPU. It assigns multiple threads to an alignment task as shown in Figure 2 (b). The four threads (purple) compute four blocks concurrently, utilizing anti-diagonal parallelism (§ 2.1). Upon completing each block, the threads move one block horizontally in sync until it reaches the end of the band (rows 0 to 3). This set of blocks processed in one horizontal pass is called a *chunk*. Then, the threads move on to the next chunk (rows 4 to 7) and compute horizontally until they fill the necessary score table cells.

Subwarp. In GPU kernels, it is common to exploit parallelism in units of a warp (i.e., 32 GPU threads). However, this yields a large external fragmentation at the start and end of a chunk. Since threads calculate each block on the same anti-diagonal in sync, high-numbered threads start late and low-numbered threads end early. This can be mitigated by splitting a warp into smaller subwarps [17, 42] and assigning a task to each subwarp as in Figure 2 (c). This reduces external fragmentation at the cost of warp divergence (internal fragmentation), but the benefits surpass the penalties.

3 Motivation

3.1 Diagnosis of the Baseline Design

In this section, we reveal four issues when orthogonally implementing the exact algorithm for guided alignment on the existing GPU-accelerated baseline design (§ 2.2).

First, there is a direct problem of storing local maximum values (Equation (6)) in the memory. This causes pressure on the memory system and adds extra computation.
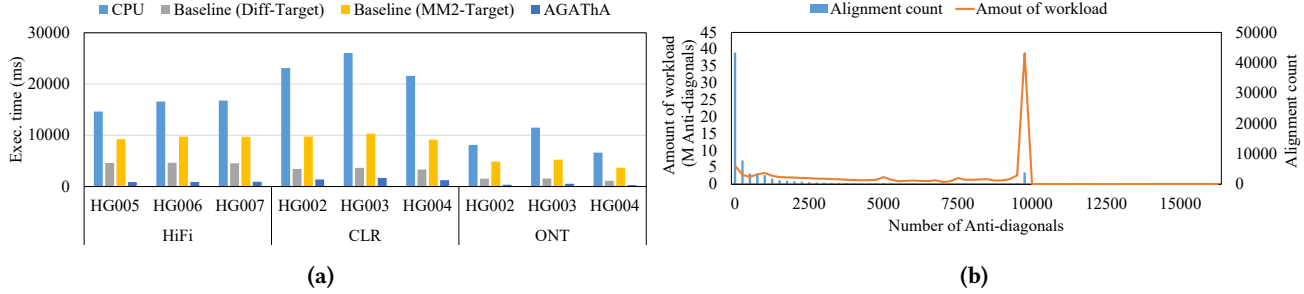
Figure 3. A motivational study. (a) represents the execution times of the existing CPU-based algorithm and two naive GPU-based alignments, and (b) represents the distribution of the accumulated workloads and alignment count for alignment tasks.

Second, there is a problem of *run-ahead processing* as depicted in Figure 2(b). The termination condition requires computing the maximum value among the cells along each anti-diagonal. However, during the horizontal progress of the threads, many anti-diagonals are not fully calculated, prohibiting the evaluation of termination conditions. For example, the termination condition at the red anti-diagonal line cannot be evaluated even after the entire chunk 1 is completed. Therefore, when the termination is performed, a huge region is unnecessarily executed.

Third, while using subwarps reduces external fragmentation, it is instead taxed with a heavy intra-warp workload imbalance. This imbalance becomes larger and harder to predict, especially with the termination condition.

Last but not least, prior work does not consider inter-warp workload imbalances. Existing approaches assign tasks to warps in the order in which the input is given. This becomes a huge issue according to our study (§ 3.2), which reveals several outliers that can cause a small portion of warps to handle all the heavy computation with long sequences.

3.2 Experimental Observation

In this section, we support our diagnosis with experimental observations. Figure 3(a) plots the execution time of a CPU-based alignment [23] ('CPU'). We compared the performance with the baseline GPU acceleration described in § 2.2 ([42]) ('Baseline (Diff-Target)'). Because this targets a different aligner from the baseline [23], we also extended it to support guiding techniques ('Baseline (MM2-Target)'). As shown in Figure 3(a), the former version of baseline enjoys 5.3× geometric mean speedup from the CPU implementation. However, when extended to the latter, it becomes 2.0×.

One cause of the slowdown is the additional overhead from tracking the local maximum values, but another cause comes from the workload imbalance stemmed by the guiding techniques, as shown in Figure 3(b). The Y-axis represents the accumulated size of all workloads of tasks that fall into the range depicted in the X-axis. Unlike most alignment tasks, certain tasks require significantly larger computations on the far right. This unique distribution works poorly with

the baseline's design of assigning tasks to subwarps in the incoming order. As revealed in § 3.1, if one subwarp is assigned a huge workload, this will cause both intra- and inter-warp imbalances that become the performance bottleneck.

4 AGAThA Design

4.1 Tracking Local Maximums with Rolling Window

To test the termination condition, calculating each cell's score also updates the local maximum of the corresponding anti-diagonal (Equation (6)). The local maximum values are to be preserved until the entire anti-diagonal is processed and tested against the termination condition (Equation (5)). As the threads progress horizontally, processing a chunk leaves many incomplete local maximums, requiring further processing. Thus, we must store the partial maximum values in the memory, which becomes a huge performance burden.

We devise a rolling window approach to temporarily store partial maximums of calculated anti-diagonals in the shared memory and periodically spill them to the global memory, as in Figure 4. To do this, we allocate a *local maximum buffer (LMB)* in shared memory, organized in a 2-D table of $3 \cdot block\_size \times num\_threads$. In this example, we use a subwarp with 4 threads ($t0$ to $t3$) and blocks with 4x4 cells (requires 12×4 table). The grid represents cells within the score table, and the purple boxes represent the anti-diagonal blocks being processed. Notice that each block spans over an equal set of seven anti-diagonals, and all threads are always on the same anti-diagonal depicted with pink. The rolling window comprises seven rows of the LMB (depicted red), representing the anti-diagonals in the current blocks' scope. Rolling window organizes each thread to keep its partial maximum values in its designated column of the window in LMB. When the window rolls down, the values for complete local max values are reduced and spilled to the *global max buffer (GMB)* in the device memory as the following:

1. (Step 1) Each thread is processing the top-left cell of a block, being on the same anti-diagonal. They calculate the cell scores into registers. The values are then written to the rolling window's first row in LMB, representing the first anti-diagonal's thread-local maximum values.
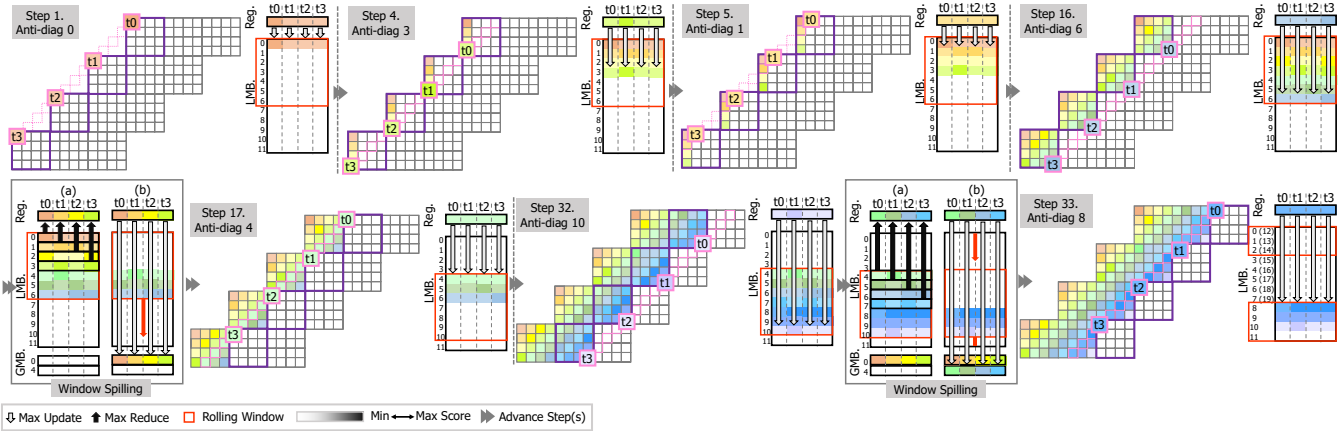
Figure 4. Tracking the anti-diagonal maximums with rolling window. Cells on the same anti-diagonal are colored with the same hue.

2. (Step 4) The threads proceed vertically inside a block. The first four processed cells belong to four different anti-diagonals, storing the values on each corresponding row within the rolling window.

3. (Step 5) Next, the threads are back to processing the second anti-diagonal. The threads compare and update the second thread-local max values in the rolling window.

4. (Step 16) All 4x4 cells of the block have been processed, which correspond to anti-diagonals 0 ∼ 6. Note that anti-diagonals 0 ∼ 3 are complete, while 4 ∼ 6 are not.

5. (Window Spilling) After step 16, a spilling step follows.
   a. Threads select one of the completed four rows in the LBM and read it to perform max-reduction using *__reduce_max_sync* warp intrinsic. Afterward, the four rows are cleared.
   b. The reduced values are updated to the GMB.

6. (Step 17) Each thread progresses horizontally onto the next block. The cells of this new block correspond to anti-diagonals 4 ∼ 10. The rolling window also 'rolls down' to target the next anti-diagonals.

7. (Step 32∼33) The procedure repeats for the next blocks.

This consumes a reasonable amount of shared memory while removing redundant memory accesses and utilizing coalescing. Note that the values from rows remaining in the window (e.g., (4 ∼ 6) in step 17) are re-accessed in the later steps. Additionally, in a special case where LMB is large enough to keep all the anti-diagonals, the spilling can be skipped (see § 4.2 for details).

4.2 Sliced Diagonal Strategy

As exemplified in § 3.1 and § 4.1, the horizontal-only progress strategy has several drawbacks on max tracking and run-ahead processing. To address these, we propose a novel tiling scheme named sliced diagonal, depicted in Figure 5. In this strategy, we partition the band into multiple *slices* (red) along the anti-diagonal direction as shown in Figure 5(a) with
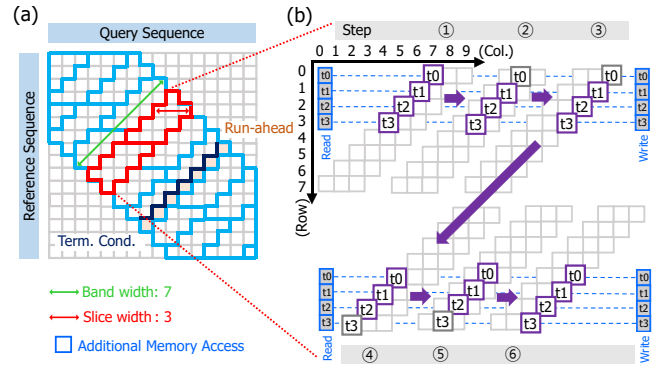
Figure 5. Sliced diagonal strategy.

the width of s (in the figure, 3 blocks). Each slice is further horizontally partitioned into chunks whose number of rows is equal to the number of threads in a subwarp (in the figure, 4 blocks). Within a slice (Figure 5(b)), each thread within a subwarp computes blocks on the same row (①∼③). Since the threads access the same row, they can keep the intermediate values using the registers. Then all threads in a subwarp move on to the next block to the next chunk to the bottom left (④), and continue until the slice is completed (⑤∼⑥). When the entire slice is finished (⑥), we check the termination condition for the completed anti-diagonals in the slice.

The proposed kernel enjoys the benefits of reduced run-ahead execution and the shared memory requirement of local max tracking. First, the amount of run-ahead execution size is greatly reduced. Unlike the baseline kernel, the run-ahead execution does not exceed $s \times Band\_width$ as again depicted with brown color in Figure 5. Second, it reduces the shared memory requirement for the rolling window. As mentioned in § 4.1, if the number of entire anti-diagonals in a slice is small enough to fit in the LMB, this can eliminate the need for global memory accesses from rolling window.
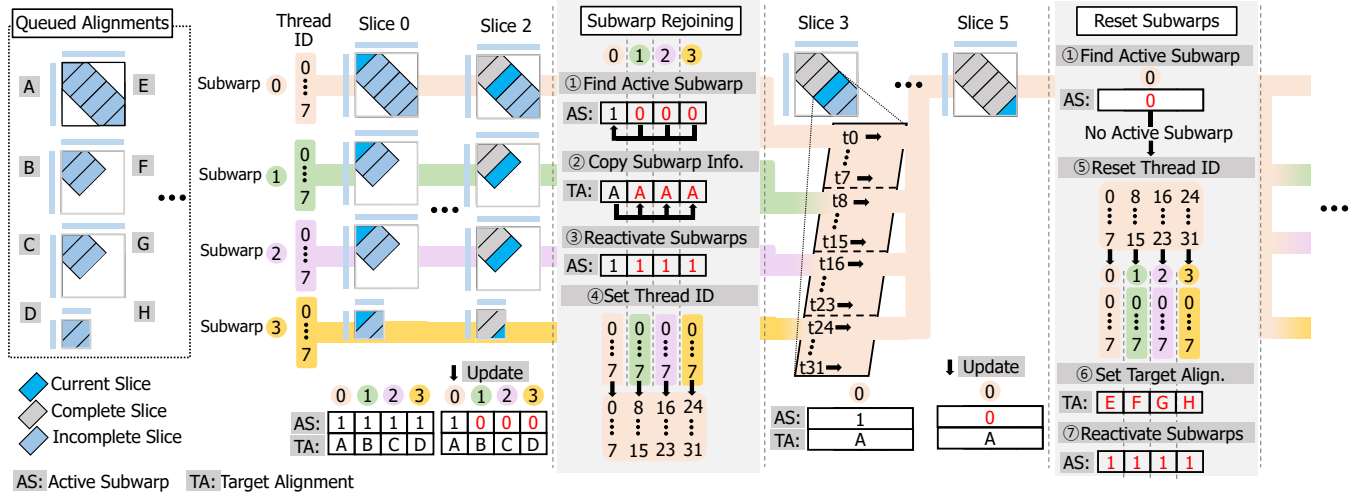
Figure 6. Subwarp rejoining.

However, there is an interesting trade-off with these overhead reductions in memory access. When a chunk in a slice starts/ends, it must read/write horizontal intermediate values (scores from $(i, j − 1)$ in Equation (3)) for the following slices as in Figure 5(b). Thus as s decreases, the number of total slices within the score table increases, leading to more memory access. After some tuning effort (details in § 5.5), we settled at $s = 3$. Note that when s is larger than the band width, the sliced diagonal kernel reduces to the baseline kernel, making sliced diagonal a generalization of the baseline.

4.3 Reducing Warp Divergence with Subwarp Rejoining

Subwarp rejoining is a scheme designed to address the warp divergence from subwarps' workload imbalance. Figure 6 shows an example warp with four subwarps of eight threads each, where only the first subwarp is assigned a large alignment task. Because those subwarps belong to a single warp, this might cause significant underutilization.

To amend this issue, we design a novel scheme called subwarp rejoining. Subwarp rejoining is a form of work stealing, but the novelty lies in that it is tightly coupled to the application such that it operates in a low overhead and fine-grained manner. In short, subwarp rejoining waits until a working subwarp finishes a slice, and rejoins idling subwarps to form a larger subwarp. The detailed steps of subwarp rejoining are as the following:

1. (Slice 0) Each subwarp starts with the first slice of its alignment task. Each subwarp maintains a flag in the shared memory for being active or not (Active Subwarp, AS) and the ID of said task (Target Alignment, TA).
2. (Slice 2) After slice 2 is completed, tasks B, C, and D are finished. The respective subwarps update AS to 0, which triggers subwarp rejoining.
3. (Subwarp Rejoining) works as the following:

a. The deactivated subwarps search AS to find an active subwarp (subwarp 0).
b. The deactivated subwarps copy information from the active subwarp's task in TA.
c. The subwarps set their AS flags back to active.
d. Subwarps are merged by adjusting local thread IDs using *__match_any_sync* warp intrinsic.
4. (Slice 3) Subwarp 0 (now the entire warp) computes the remainder of slice 3 with all 32 threads.
5. After task A is completed, AS is updated.
6. (Reset Subwarps) As no active subwarp remains according to AS, the subwarps are re-split to the original sizes and each fetches a new task.

The key of subwarp rejoining is synchronizing the subwarps at slice boundaries. This indicates another trade-off with the slice width s, where a wide slice would cause longer subwarp idle time, and the contrary would cause too frequent overhead of checking for subwarp rejoining feasibility.

### 4.4 Workload Balancing with Uneven Bucketing

Uneven bucketing is the last piece of AGAThA designed to reduce the inter-warp workload imbalance. In the baseline method, the alignment tasks are assigned to warps in an arbitrary order (i.e., sequential). However, in the existence of extra-long workloads, as depicted in Figure 3(b), some warps will receive much longer workloads than others, which causes the performance to be dominated by the slowest warp.

To prevent this from happening, we use uneven bucketing as illustrated in Figure 7 to distribute long sequences to different warps, flattening out the workload per warp. The method is straightforward to implement:

1. Sort the given sequences to pick the longest $1/N$ sequences where N is the number of subwarps per warp.
2. Redistribute the sorted sequences such that one long sequence is assigned per warp.

Table 1. Performance Models

Design	Model
Baseline	$\widetilde{MAX}_{Warps}(MAX_{Subwarps}(Cells \times (\frac{1}{Comp.TP} + \frac{AR_{Anti}\quad+AR_{Inter}\quad+AR_{Term}}{Mem.TP})))$
+RW	$\widetilde{MAX}_{Warps}(MAX_{Subwarps}(Cells \times (\frac{1}{Comp.TP} + \frac{AR_{Anti}(\downarrow)+AR_{Inter}\quad+AR_{Term}}{Mem.TP})))$
+RW+SD	$\widetilde{MAX}_{Warps}(MAX_{Subwarps}(Cells(\downarrow) \times (\frac{1}{Comp.TP} + \frac{AR_{Anti}(\downarrow)+AR_{Inter}(\uparrow)+AR_{Term}(\downarrow)}{Mem.TP})))$
+RW+SD+SR	$\widetilde{MAX}_{Warps}(\widetilde{AVG}_{Subwarps}(Cells \times (\frac{1}{Comp.TP} + \frac{AR_{Anti}\quad+AR_{Inter}\quad+AR_{Term}}{Mem.TP})))$
+RW+SD+SR+UB	$\widetilde{AVG}_{Warps}(\widetilde{AVG}_{Subwarps}(Cells \times (\frac{1}{Comp.TP} + \frac{AR_{Anti}\quad+AR_{Inter}\quad+AR_{Term}}{Mem.TP})))$
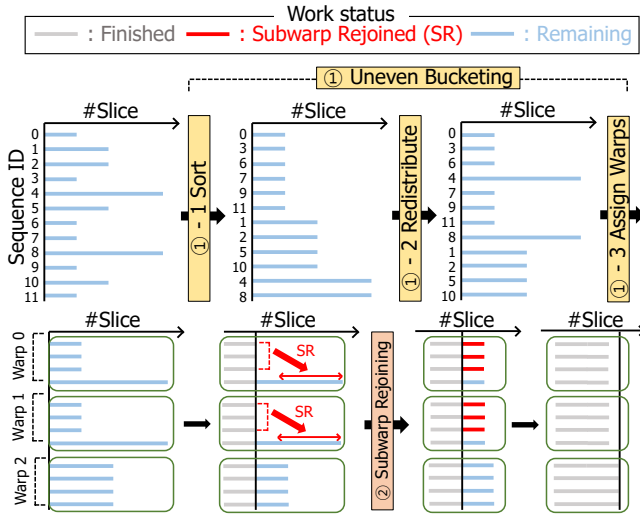
Figure 7. Uneven bucketing used with subwarp rejoining.

Uneven bucketing owes to subwarp rejoining on handling the uncertainty coming from the dynamic nature of the termination condition. If the long task continues without termination, the large workload can be automatically redistributed with subwarp rejoining. If termination does occur, subwarp rejoining can still reduce the overall execution time by rejoining the terminated subwarp to other active subwarps.

4.5 Performance Modeling

We present a simple performance model for the latency of each scheme of AGAThA in Table 1. Within a subwarp, the latency is proportional to the total number of cells in the score table, simplified as below:

$$Cells = Antidiags \times Band\_width + Runahead. \quad (8)$$

Then we model its corresponding cost for computation and memory access. All cells in the scoreboard are processed at a certain computational throughput ($Comp.TP$), and require memory access at memory throughput ($Mem.TP$). There are three parameters that model the portion of cells that access memory: memory access ratio for storing the anti-diagonal max values (AR_{Anti}), managing intermediate

values (AR_{Inter}), and checking the termination condition (AR_{Term}). At the baseline, these ratios can be approximated as $1 : (1/8) : (1/Band\_width)$, respectively. Then, a combination of the subwarp latencies models the warp latency, and a combination of the warp latencies models the total latency.

In the baseline, the subwarps have no interaction, so the longest warp dominates the overall latency ($MAX_{Subwarps}()$). Similarly, in the existence of extremely long queries, the longest warp will dominate the execution time ($\widetilde{MAX}_{Warps}()$) where $\widetilde{MAX}(\cdot)$ denotes a function dominated by maximum.

By applying **rolling window (RW)**, the kernel can use shared memory to greatly reduce the number of memory accesses for anti-diagonal max tracking. This is depicted by reduced AR_{Anti} in the model. Next, by adding **sliced diagonal (SD)**, we can mainly reduce the amount of run-ahead processing, leading to an overall decrease in $Cells$. We can additionally decrease both AR_{Anti} and AR_{term} by using an optimal slice length. As discussed in § 4.2, it slightly increases AR_{inter}, but the benefit outweighs the penalty. Applying **subwarp rejoining (SR)** changes $MAX_{Subwarps}()$ close to the average ($\widetilde{AVG}_{Subwarps}()$) by letting subwarps help others within the warp. Finally, **uneven bucketing (UB)** has the effect of reducing the straggler warps and changes the maximum dominated latency $\widetilde{MAX}_{Warps}()$ close to their average $\widetilde{AVG}_{Warps}()$.

5 Evaluation

5.1 Experimental Setup

We evaluated AGAThA on a server with NVIDIA RTX-A6000 GPU and AMD EPYC 7313P 16-Core Processor with 64 GB of RAM. It runs on Ubuntu 20.04 with CUDA version 11.7 and driver version 525.60.13.

For real-world datasets, one reference and nine query datasets were used. For reference, GRCh38 [44], an up-to-date assembly of the human genome was used that has 3.1G base pairs (i.e., literals). For the nine query datasets, we used data from the 'Genome in a Bottle' project [37]. The datasets can be grouped into three categories by the sequencing techniques used to generate the dataset. The first category consists of HiFi HG (human genome) 005~007, which the PacBio

HiFi [55] sequencing technique was used to obtain reads from the ChineseTrio. The second category contains CLR 002~004 generated by the PacBio CLR [40] sequencing technique from the AshkenazimTrio. The last category holds reads from the same target, but the reads were extracted with the ONT [52] sequencing technique. We picked the first 50,000 reads from each dataset, and ran them through the pre-computing steps [23] to obtain the final datasets for alignment. We mainly use Minimap2 [23] as the reference algorithm, but we also test the feasibility of AGAThA against BWA-MEM [22] (§ 5.9). We used Minimap2's preset parameters for each dataset category.

5.2 Baselines

The GPU-based baselines chosen for the evaluation are as follows:

- **Manymap [12]** is a GPU alignment based on Minimap2. However, Manymap applies an inexact interpretation of the termination condition. In addition, it allows aligning only one sequence at a time, so we fixed it to accept multiple different reads in parallel using CUDA streams.
- **GASAL2 [1]** is a GPU alignment with input packing and inter-query parallelism. Out of the multiple kernels implemented, we use the banding kernel in GASAL2.
- **SALoBa [42]** utilizes intra-query parallelism. We applied banding heuristic that gives further speedup, similar to GASAL2.
- **LOGAN [57]** is an algorithm that implements its own guiding algorithm. It adjusts the band width during score table filling after calculating each anti-diagonal as a form of guided alignment.

For fair comparison, we measure the performance of the baselines in two versions. One is measured in the version that they originally target (Diff-Target), and another is extended with faithful optimization efforts to provide output equal to the reference algorithm (MM2-Target). This represents a scenario where those libraries are used to accelerate Minimap2 in the field. We fixed the provided termination condition for Manymap and newly implemented it in GASAL2/SALoBa. Since LOGAN targets a different guiding algorithm, we only report the unmodified performance for LOGAN. Note that Manymap is the only baseline intended as a direct replacement of Minimap2's algorithm. Others target different algorithms, thus not implementing all of the heuristics we focus on.

5.3 Performance Comparison

In this section, we evaluate the performance of AGAThA using real-world sequencing datasets. Figure 8 reveals the performance comparisons of AGAThA against the baselines, normalized to the performance of Minimap2 on a CPU. For the baselines, the blank bars represent the speed of the originally targeted algorithms (Diff-Target), while the solid bars
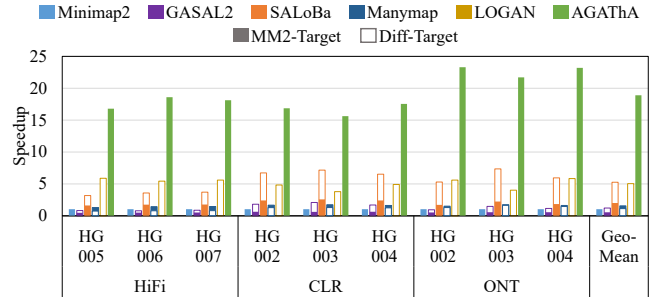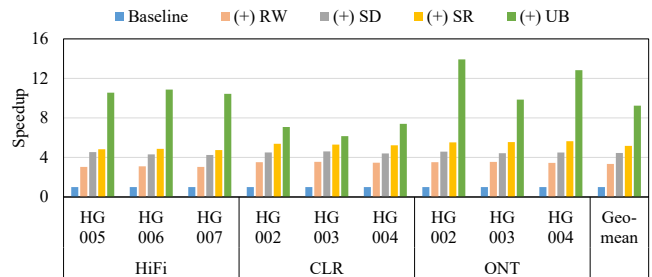

Figure 8. Performance comparison.

Figure 9. Ablation study.

represent the speed of implementations with the reference guiding algorithm as their target (MM2-Target).

Against all baselines, AGAThA clearly outperforms them significantly. Over Minimap2, AGAThA showed 18.8× geometric mean speedup. While AGAThA shows more than 16× speedup for all datasets, it was the largest on the HiFi dataset (22.6×).

Between the Minimap2-targeted baselines, SALoBa was the fastest but was 9.6× slower than AGAThA. Next, Manymap and GASAL2 were 12.1× and 36.6× slower than AGAThA, respectively. This version of GASAL2 was even slower than Minimap2. Also, Manymap is the only version that benefits (albeit slightly) from implementing the guided alignment algorithm. We suspect that this is from the fact that Manymap is already filling the score table by computing each anti-diagonal, removing the run-ahead processing entirely.

Among baselines that do not target Minimap2, SALoBa was also the fastest but was still 3.6× slower than AGAThA. SALoBa's speedup comes from albeit naive, banding reducing the computational workload, and other GPU acceleration techniques such as subwarps were used. Additionally, LOGAN's performance closely follows SALoBa. One cause could be that LOGAN maintains a gap score that is less expensive in both computation and memory. Manymap was the slowest with 1.1× geometric mean speedup.

5.4 Ablation Study

To demonstrate the advantage of how AGAThA achieves its speedup, we conduct an ablation study as in Figure 9. For the baseline, we use the naive exact implementation of the
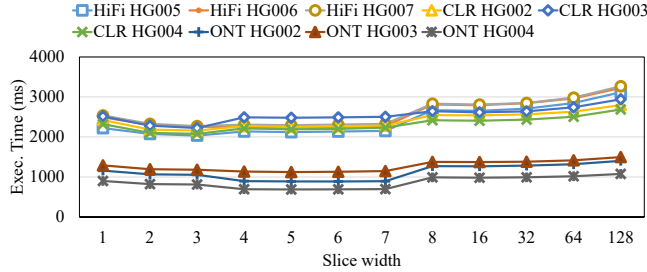
Figure 10. Slice width sensitivity study.

Figure 11. Effect of the workload balancing techniques.

Figure 12. Workload distribution from workload balancing.

guiding algorithm, without the proposed techniques. By using rolling window for max tracking (RW), AGAThA already achieves an average of 3.1× on the HiFi datasets and 3.5× on the rest. This speedup comes from decreasing the number of global memory accesses and using max-reduce to parallelize calculating the anti-diagonal maximum. Sliced diagonal (SD) further optimizes rolling window by additional speedup of 1.4× for the HiFi datasets, and 1.3× for the other two datasets, respectively. It optimizes rolling window by reducing the run-ahead execution and only using an essential amount of shared memory to reduce global memory access. In addition, subwarp rejoining (SR) increases speedup by an additional 1.1× for the HiFi datasets, 1.2× for the rest. It removes the warp divergence coming from subwarps receiving different-sized workloads. Finally, uneven bucketing (UB) boosts the alignment performance by approximately 2.2× more on the HiFi and ONT datasets, and 1.3× more on CLR datasets. Uneven bucketing spreads the few extremely large workloads to multiple warps to solve the inter-warp imbalance problem with the aid of subwarp rejoining.

5.5 Sensitivity Study on Slice Width

The slice width is an important parameter that can highly affect the performance of the sliced diagonal kernel. In Figure 10, we change the slice width from 1 up to 128 blocks. There is an overall decreasing trend from 1 to 4, flattening around 5 to 16, and an increasing trend as the slice width gets larger. The first decrease comes from reducing the number of memory accesses for intermediate values, and the increasing trend at the end comes from the growing amount of run-ahead processing. Among this overall trend, we can see small jumps after slice widths 3 and 7. This is because it is possible to use bitwise & operation with these widths instead of modulo operation which is known to be slow on GPUs. While there were both similar speedups at widths 3 and 7, we chose 3 as our main target. This is because it uses less shared memory (and saves space for implementing subwarp rejoining), and the speedup for the longer datasets HiFi and CLR were higher for width 3. The upper hand of width 3 on long datasets stems from the fact that the effect of memory access for anti-diagonal max scores and run-ahead execution is more important than short datasets.
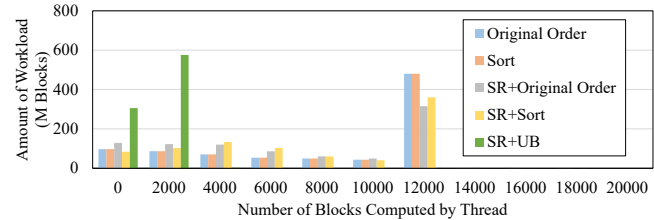
### 5.6 Subwarp Rejoining and Uneven Bucketing

In this section, we study the impact of subwarp rejoining and uneven bucketing by observing the speedup and distribution of allocated workload per thread. We compare them with sorting, which we believe is simple and intuitive for reducing workload imbalances. Figure 11 shows the performance comparisons, where the 'Original order' represents AGAThA only with rolling window and sliced diagonal applied. If we apply sorting to the workloads by the number of anti-diagonals, there was approximately 1.06× speedup in geometric mean for the entire dataset ('Sort'). This is from the reduced warp divergence by allocating similar-sized score tables to the subwarps within the same warp. However, applying subwarp rejoining to the baseline ('SR + Original Order') shows a speedup of 1.17×. The result shows that subwarp rejoining is more effective in reducing warp divergence than sorting. This is because sorting cannot adapt to the dynamic behavior of termination, while subwarp rejoining can.

If we apply subwarp rejoining to the sorted sequences ('SR + Sort'), the speedup is still 1.17× in geometric mean, similar to only applying subwarp rejoining. Finally, with subwarp rejoining and uneven bucketing, the speedup is 2.22× on average for all datasets. This shows how subwarp rejoining and uneven bucketing working in unison is the best approach in reducing workload divergence both within and between warps.

Figure 12 shows how the combination of subwarp rejoining and uneven bucketing redistributes the workload evenly to the subwarps. The Y axis shows the accumulated number of blocks, representing the amount of workload. On the 'Original Order', much of the work is performed on subwarps whose number of initially assigned blocks per thread is around 12,000. While 'SR+Original Order' and 'SR+Sort' do reduce the work from the heavily loaded subwarps, their impact is limited. Contrarily, we can see that subwarp rejoining
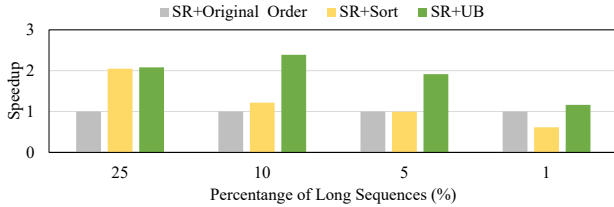
**Figure 13.** Performance on datasets with different long sequence percentages.
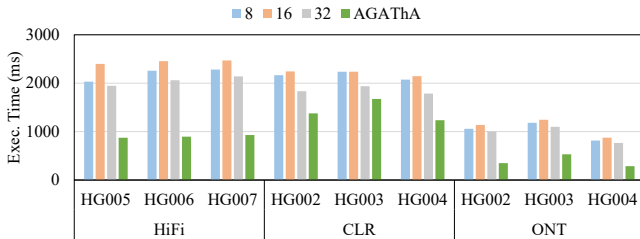
Figure 14. Sensitivity study on subwarp size.

Figure 15. Hardware flexibility of AGAThA.

and uneven bucketing together shift the entire graph to the left, which represents how the large amounts of workloads that were allocated to a few subwarps are spread out to many other subwarps. This shows the underlying mechanism of how uneven bucketing is such a powerful acceleration tool with our data distribution.

To better understand why uneven bucketing shows significant speedup, we compare the performance of uneven bucketing on generated datasets in Figure 13. We generated datasets by varying the percentage of long sequences (4096 bp) against short sequences (128 bp). To isolate the impact of uneven bucketing, we used 'SR+Original Order' as the baseline and compared 'SR+Sort' and 'SR+UB'. Even with a similar speedup at 25%, uneven bucketing always outperformed sorting for every percentage, showing the highest speedup of 2.39× at 10%. Notably, the speedup of sorting peaks at 25% and continues to drop as the percentage decreases, even becoming slower than the original ordering by 0.61×. The main reason for this drastic slowdown is that a few warps with long sequences become the main bottleneck. Contrarily, uneven bucketing is still faster than the original ordering because it distributes the long sequences to different warps and can fully utilize subwarp rejoining. Note that the percentage of alignments on the far right peak in Figure 3(b) ranged between 5∼20% for all datasets.

5.7 Sensitivity Study on Subwarp Size

We conducted a sensitivity study on the effect of different subwarp sizes on execution time. Figure 14 shows the execution time for different subwarp sizes 8, 16, and 32 (full warp) compared to the final AGAThA. Due to the nature of subwarps, warp divergence increases as the subwarp size decreases. One could think it would be faster to use the full warp per alignment instead of subwarps. For the kernel with
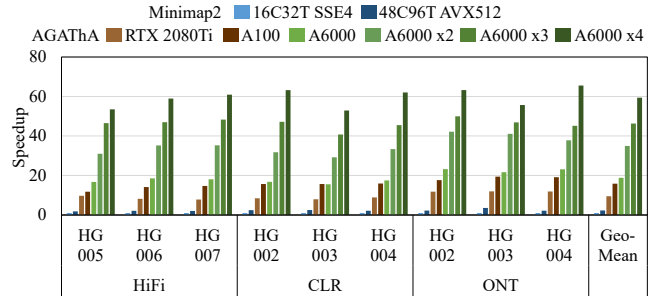
rolling window and sliced diagonal only, it is true that using the full warp is faster than using subwarps, on average 10%. However, this is easily outpaced by the final version of AGAThA, which uses designs such as subwarp rejoining and uneven bucketing that must have a subwarp to be implemented. Additionally, there are slowdowns at 16 threads per subwarp. This can be explained by the increased amount of idling threads at the start and end of a chunk with the remaining warp divergence from the usage of subwarps.

5.8 Hardware Flexibility of AGAThA

In the present section, we demonstrate the performance of AGAThA on various hardware environments in Figure 15.

Comparison with a Stronger CPU Baseline. In addition to the default CPU baseline implemented with SSE4.1 support that runs on a 16-core 32-thread processor ('16C32T SSE4'), we also experimented with a stronger CPU baseline with AVX512 support using optimized implementation from [18] on a 48-core 96-thread environment (2 × Xeon Gold 6442Y Processor, '48C96T AVX512'). Overall, the stronger baseline was 2.30× faster in geometric mean compared to the default one. Even with the stronger CPU baseline, AGAThA still showed a significant 8.19× geometric mean speedup with a single GPU.

Sensitivity on GPU Types. We also tested AGAThA with RTX 2080Ti and A100 to verify the applicability on other GPUs. As RTX 2080Ti does not support warp-reduce functions from recent GPUs, we replaced them with shared memory access. AGAThA provided stable 9.49× and 15.84× speedup over the CPU baseline in geometric mean, respectively. Although A100 is perceived as a higher grade than A6000, A6000 performs better due to having a larger cuda core count.

Scalability on Number of GPUs. Additionally, we extend AGAThA to multiple GPUs by distributing equal numbers of alignment tasks to each GPU and making each GPU process them. AGAThA showed almost linear scalability and achieved 59.38× geometric mean speedup with four GPUs over the CPU baseline, which is close to linear compared to 18.83× speedup from a single GPU. We expect AGAThA to perform even better by supporting workload balancing among different GPUs similar to uneven bucketing.
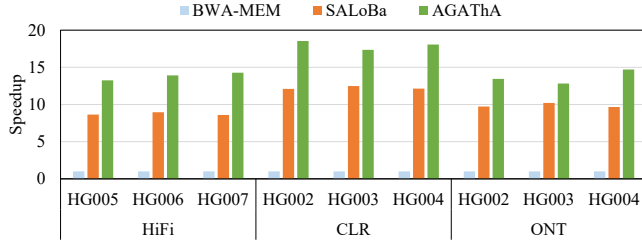
Figure 16. Result of applying AGAThA to BWA-MEM.

5.9 Applying AGAThA to BWA-MEM

Finally, we evaluate AGAThA on BWA-MEM [22], an algorithm for older generations that is still widely used. We demonstrate that our schemes can be applied to guided alignment algorithms other than Minimap2. We compare the runtime of BWA-MEM on CPU with SALoBa and AGAThA by applying BWA-MEM's guided alignment. We chose SALoBa, the fastest baseline from § 5.3. Figure 16 shows that AGAThA also has the speedup gap above SALoBa. The speed gap is smaller than on Minimap2 mainly due to the default band width and termination threshold being significantly smaller, effectively reducing the amount and imbalance of task workloads. However, AGAThA still achieves a significant speedup of 15× compared to BWA-MEM on a CPU.

6 Discussion

Applying DPX to AGAThA. The new NVIDIA Hopper architecture [9] introduces a new instruction called Dynamic Programming extension (DPX) [39]. It can accept two or three integers and compute ReLU or min/max comparison and is said to accelerate sequence alignments up to 4.4× [38]. We expect DPX to be seamlessly integrated into our kernel and help AGAThA thrive even more because DPX enhances the computational speed, and AGAThA mainly addresses the issues from memory bandwidth bottleneck.

Different Bucketing Parameters. While uneven bucketing largely contributes to our speedup, we see more potential in this design. For example, if we predict exactly when the termination condition is met before execution, then the kernel could remove most of the remaining workload imbalance. We would like to explore this possibility in future work.

7 Related Work

GPU Accelerations of Sequence Alignments. There is a line of work on accelerating sequence alignment with GPUs, using OpenGL library [25, 26], or earlier version of CUDA [6, 27, 30, 31, 33]. In addition, some target single-pair alignment and utilize intra-query parallelism [10, 11, 46, 47]. To be integrated into the read mapping algorithm, multiple alignments have to be performed on relatively shorter partial pieces of sequence. SOAP [24] and CUSHAW family [28, 29, 32] are early ones on such approach, and GASAL2 [1] introduces on-GPU input packing. SALoBa [42] further optimizes on

GPU memory hierarchy. However, while they are compared with existing reference algorithms [22, 23], none have implemented the exact guiding algorithms.

Guided Alignment. Minimap2 and BWA-MEM are regarded as golden standards, but they are not the only algorithms with a guided dynamic programming approach. For example, banding is implemented on other variant algorithms [18, 45]. On GPUs, [21] is a GPU-based alignment library with banding, and F5C [14] adaptively decides the direction of the band for every iteration. The termination condition has been first introduced in Blast [4], called *X-Drop*. However, it was found to penalize single long gaps too much. Because of this, it has been amended to be *Z-drop* on later work as in Equation (5) by Minimap2 [23]. It has been accelerated using SIMD instructions [18], but not on GPUs.

Balancing Workload on GPUs. Workload balancing for GPUs is a classic problem studied for decades [8, 16, 19, 49, 50, 53, 58]. [56] exploits software-level control of scheduling on streaming multiprocessors. [20] suggests dynamic task-to-thread assignment, and [59] provides a pipelined parallel programming framework on GPU for job scheduling. AGAThA also applies a type of workload balancing, but does so in deep relation with the algorithm and the observed real-world data distribution.

8 Conclusion

We propose AGAThA, a GPU-accelerated sequence alignment software. It is the first exact GPU implementation of the guided sequence alignment algorithm. To address the challenges the guiding step imposes, we propose a new method to track anti-diagonal maximums, a tiling strategy, a dynamic workload balancing, and a workload distribution strategy. The evaluation shows that AGAThA significantly outperforms the baselines.

Data Availability Statement

The artifact of AGAThA is available at [41]. The most recent version is kept updated at https://github.com/readwrite112/AGAThA. The artifact contains a sample dataset, the whole code of AGAThA, and some evaluation-related code. For the details of the artifact, please refer to Appendix A.

References

[1] Nauman Ahmed, Jonathan Lévy, Shanshan Ren, Hamid Mushtaq, Koen Bertels, and Zaid Al-Ars. 2019. GASAL2: A GPU accelerated sequence alignment library for high-throughput NGS data. *BMC bioinformatics* 20 (2019), 1–20.

[2] Mohammed Alser, Joel Lindegger, Can Firtina, Nour Almadhoun, Haiyu Mao, Gagandeep Singh, Juan Gomez-Luna, and Onur Mutlu. 2022. From molecules to genomic variations: Accelerating genome analysis via intelligent algorithms and architectures. *Computational and Structural Biotechnology Journal* 20 (2022), 4579–4599.

[3] Mohammed Alser, Jeremy Rotman, Dhrithi Deshpande, Kodi Taraszka, Huwenbo Shi, Pelin Icer Baykal, Harry Taegyun Yang, Victor Xue, Sergey Knyazev, Benjamin D Singer, et al. 2021. Technology dictates algorithms: recent developments in read alignment. *Genome biology* 22, 1 (2021), 249.

[4] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. 1997. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic acids research* 25, 17 (1997), 3389–3402.

[5] Muaaz G Awan, Jack Deslippe, Aydin Buluc, Oguz Selvitopi, Steven Hofmeyr, Leonid Oliker, and Katherine Yelick. 2020. ADEPT: A domain independent sequence alignment strategy for GPU architectures. *BMC bioinformatics* 21, 1 (2020), 1–29.

[6] Jacek Blazewicz, Wojciech Frohmberg, Michal Kierzynka, Erwin Pesch, and Pawel Wojciechowski. 2011. Protein alignment algorithms with an efficient backtracking routine on multiple GPUs. *BMC bioinformatics* 12, 1 (2011), 1–17.

[7] Abhishek Bohra, Uday Chand Jha, Ian D Godwin, and Rajeev Kumar Varshney. 2020. Genomic interventions for sustainable agriculture. *Plant Biotechnology Journal* 18, 12 (2020), 2388–2405.

[8] Sanjay Chatterjee, Max Grossman, Alina Sbîrlea, and Vivek Sarkar. 2011. Dynamic task parallelism with a GPU work-stealing runtime system. In *LCPC Workshop*. Springer, Fort Collins, CO, USA, 203–217.

[9] Jack Choquette. 2023. NVIDIA Hopper H100 GPU: Scaling Performance. *IEEE Micro* 43 (2023), 9–17.

[10] Edans Flavius de Oliveira Sandes, Guillermo Miranda, Xavier Martorell, Eduard Ayguade, George Teodoro, and Alba Cristina Magalhaes Melo. 2016. CUDAlign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in GPU clusters. *IEEE Transactions on Parallel and Distributed Systems* 27, 10 (2016), 2838–2850.

[11] F de O Edans, Guillermo Miranda, Alba CMA de Melo, Xavier Martorell, and Eduard Ayguadé. 2014. CUDAlign 3.0: Parallel biological sequence comparison in large GPU clusters. In *CCGrid*. IEEE, Chicago, IL, USA, 160–169.

[12] Zonghao Feng, Shuang Qiu, Lipeng Wang, and Qiong Luo. 2019. Accelerating long read alignment on three processors. In *ICPP*. ACM, Kyoto, Japan, 1–10.

[13] Daichi Fujiki, Shunhao Wu, Nathan Ozog, Kush Goliya, David Blaauw, Satish Narayanasamy, and Reetuparna Das. 2020. SeedEx: A Genome Sequencing Accelerator for Optimal Alignments in Subminimal Space. In *MICRO*. IEEE, Athens, Greece, 937–950.

[14] Hasindu Gamaarachchi, Chun Wai Lam, Gihan Jayatilaka, Hiruna Samarakoon, Jared T Simpson, Martin A Smith, and Sri Parameswaran. 2020. GPU accelerated adaptive banded event alignment for rapid comparative nanopore signal analysis. *BMC bioinformatics* 21 (2020), 1–13.

[15] Alice Maria Giani, Guido Roberto Gallo, Luca Gianfranceschi, and Giulio Formenti. 2020. Long walk to genomics: History and current approaches to genome sequencing and assembly. *Computational and Structural Biotechnology Journal* 18 (2020), 9–19.

[16] Kshitij Gupta, Jeff A Stuart, and John D Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *InPar*. IEEE, San Jose, CA, USA, 1–14.

[17] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. In *PPoPP*. ACM, San Antonio, TX, USA, 267–276.

[18] Saurabh Kalikar, Chirag Jain, Md Vasimuddin, and Sanchit Misra. 2022. Accelerating minimap2 for long-read sequencing applications on modern CPUs. *Nature Computational Science* 2, 2 (2022), 78–83.

[19] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. 2015. Efficient warp execution in presence of divergence with collaborative context collection. In *MICRO*. ACM, Waikiki, HI, USA, 204–215.

[20] Farzad Khorasani, Bryan Rowe, Rajiv Gupta, and Laxmi N Bhuyan. 2016. Eliminating intra-warp load imbalance in irregular nested patterns via collaborative task engagement. In *IPDPS*. IEEE, Chicago, IL, USA, 524–533.

[21] Matija Korpar and Mile Šikić. 2013. SW#–GPU-enabled exact alignments on genome scale. *Bioinformatics* 29, 19 (2013), 2494–2495.

[22] Heng Li. 2013. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. arXiv:1303.3997 [q-bio.GN]

[23] Heng Li. 2018. Minimap2: Pairwise alignment for nucleotide sequences. *Bioinformatics* 34, 18 (2018), 3094–3100.

[24] Chi-Man Liu, Thomas Wong, Edward Wu, Ruibang Luo, Siu-Ming Yiu, Yingrui Li, Bingqiang Wang, Chang Yu, Xiaowen Chu, Kaiyong Zhao, et al. 2012. SOAP3: Ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics* 28, 6 (2012), 878–879.

[25] Weiguo Liu, Bertil Schmidt, Gerrit Voss, Andre Schroder, and Wolfgang Muller-Wittig. 2006. Bio-sequence database scanning on a GPU. In *IPDPS*. IEEE, Rhodes Island, Greece, 1–8.

[26] Yang Liu, Wayne Huang, John Johnson, and Sheila Vaidya. 2006. GPU accelerated smith-waterman. In *ICCS*. Springer, Reading, UK, 188–195.

[27] Yongchao Liu, Douglas L Maskell, and Bertil Schmidt. 2009. CUDASW++: Optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC research notes* 2, 1 (2009), 1–10.

[28] Yongchao Liu, Bernt Popp, and Bertil Schmidt. 2013. High-speed and accurate color-space short-read alignment with CUSHAW2. arXiv:1304.4766 [q-bio.GN]

[29] Yongchao Liu, Bernt Popp, and Bertil Schmidt. 2014. CUSHAW3: Sensitive and accurate base-space and color-space short-read alignment with hybrid seeding. *PloS one* 9, 1 (2014), e86869.

[30] Yongchao Liu and Bertil Schmidt. 2015. GSWABE: Faster GPU-accelerated sequence alignment with optimal alignment retrieval for short DNA sequences. *Concurrency and Computation: Practice and Experience* 27, 4 (2015), 958–972.

[31] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. 2010. CUDASW++ 2.0: Enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC research notes* 3, 1 (2010), 1–12.

[32] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. 2012. CUSHAW: A CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform. *Bioinformatics* 28, 14 (2012), 1830–1837.

[33] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. 2013. CUDASW++ 3.0: Accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC bioinformatics* 14, 1 (2013), 1–10.

[34] Tuomo Mantere, Simone Kersten, and Alexander Hoischen. 2019. Long-read sequencing emerging in medical genetics. *Frontiers in genetics* 10 (2019), 426.

[35] NCBI. 1982. GenBank. https://www.ncbi.nlm.nih.gov/genbank/, visited 2024-01-15.

[36] Saul B Needleman and Christian D Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology* 48, 3 (1970), 443–453.

[37] NIST. 2012. Genome in a Bottle. https://www.nist.gov/programs-projects/genome-bottle, visited 2024-01-16.

[38] NVIDIA. 2022. Boosting Dynamic Programming Performance Using NVIDIA Hopper GPU DPX Instructions. https://developer.nvidia.com/blog/boosting-dynamic-programming-

performance-using-nvidia-hopper-gpu-dpx-instructions/, visited 2024-01-16.

[39] NVIDIA. 2022. NVIDIA Hopper GPU Architecture Accelerates Dynamic Programming Up to 40x Using New DPX Instructions. https://blogs.nvidia.com/blog/2022/03/22/nvidia-hopper-accelerates-dynamic-programming-using-dpx-instructions/, visited 2024-01-15.

[40] PacBio. 2023. PacBio - Sequence with Confidence. https://www.pacb.com/, visited 2023-04-21.

[41] Seongyeon Park. 2024. *readwrite112/AGAThA: AGAThA: Fast and Efficient GPU Acceleration of Guided Sequence Alignment for Long Read Mapping*. AISys. https://doi.org/10.5281/zenodo.10462237

[42] Seongyeon Park, Hajin Kim, Tanveer Ahmad, Nauman Ahmed, Zaid Al-Ars, H Peter Hofstee, Youngsok Kim, and Jinho Lee. 2022. SALoBa: Maximizing Data Locality and Workload Balance for Fast Sequence Alignment on GPUs. In *IPDPS*. IEEE, Lyon, France, 728–738.

[43] Julian Parkhill and Brendan W Wren. 2011. Bacterial epidemiology and biology-lessons from genome sequencing. *Genome biology* 12 (2011), 1–7.

[44] NCBI RefSeq. 2019. GRCh38.p13. https://www.ncbi.nlm.nih.gov/data-hub/genome/GCF_000001405.39/, visited 2024-01-16.

[45] Harisankar Sadasivan, Milos Maric, Eric Dawson, Vishanth Iyer, Johnny Israeli, and Satish Narayanasamy. 2023. Accelerating Minimap2 for accurate long read alignment on GPUs. *Journal of biotechnology and biomedicine* 6, 1 (2023), 13–23.

[46] Edans Flavius de O Sandes and Alba Cristina MA de Melo. 2011. Smith-Waterman alignment of huge sequences with GPU in linear space. In *IPDPS*. IEEE, Anchorage, AK, USA, 1199–1211.

[47] Edans Flavius O Sandes and Alba Cristina MA de Melo. 2010. CUD-Align: Using GPU to accelerate the comparison of megabase genomic sequences. In *PPoPP*. ACM, New York, NY, USA, 137–146.

[48] Temple F Smith, Michael S Waterman, et al. 1981. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197.

[49] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. 2012. Softshell: dynamic scheduling on gpus. *ACM Transactions on Graphics* 31, 6 (2012), 1–11.

[50] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-based scheduling of dynamic workloads on the GPU. *ACM Transactions on Graphics* 33, 6 (2014), 1–11.

[51] Yutaka Suzuki. 2020. Advent of a new sequencing era: long-read and on-site sequencing. *Journal of Human Genetics* 65, 1 (2020), 1–1.

[52] Oxford Nanopore Technologies. 2018. PromethION. https://nanoporetech.com/products/promethion, visited 2024-01-16.

[53] Stanley Tzeng, Anjul Patney, and John D Owens. 2010. Task management for irregular-parallel workloads on the GPU. In *HPG*. ACM, Saarbrucken, Germany, 29–37.

[54] Ze-Gang Wei, Shao-Wu Zhang, and Fei Liu. 2020. smsMap: mapping single molecule sequencing reads by locating the alignment starting positions. *BMC bioinformatics* 21, 1 (2020), 1–15.

[55] Aaron M Wenger, Paul Peluso, William J Rowell, Pi-Chuan Chang, Richard J Hall, Gregory T Concepcion, Jana Ebler, Arkarachai Fungtammasan, Alexey Kolesnikov, Nathan D Olson, et al. 2019. Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. *Nature biotechnology* 37, 10 (2019), 1155–1162.

[56] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *ICS*. ACM, Newport Beach, CA, USA, 119–130.

[57] Alberto Zeni, Giulia Guidi, Marquita Ellis, Nan Ding, Marco D Santambrogio, Steven Hofmeyr, Aydın Buluç, Leonid Oliker, and Katherine Yelick. 2020. LOGAN: High-performance GPU-based x-drop long-read alignment. In *IPDPS*. IEEE, New Orleans, LA, USA, 462–471.

[58] Eddy Z Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. 2010. Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In *ICS*. ACM, Tsukuba, Ibaraki, Japan, 115–126.

[59] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. 2017. Versapipe: a versatile programming framework for pipelined computing on GPU. In *MICRO*. ACM, Cambridge, MA, USA, 587–599.

A Appendix

We provide AGAThA's source code, additional code for setup and execution, and a sample dataset as an example for the input format. For the most recent version of AGAThA's description, please refer to the up-to-date artifact link in GitHub.

A.1 Artifact Summary

- **Dataset**: A sample of HiFi HG005 dataset.
- **Runtime environment**: Ubuntu 20.04 or higher.
- **Hardware**: Multicore x86_64 CPU with one or more NVIDIA GPUs.
- **Metrics**: Execution time.
- **Code license**: Apache-2.0 license.
- **Archived DOI**:
  https://doi.org/10.5281/zenodo.10462237
- **Up-to-date artifact**:
  https://github.com/readwrite112/AGAThA

A.2 Description

A.2.1 How to access. Please access the artifact via the archived DOI [41] or the up-to-date artifact link.

A.2.2 Hardware dependencies. Requires multicore x86_64 CPU with at least a single NVIDIA GPU. The artifact was tested on NVIDIA RTX A6000 and NVIDIA GeForce RTX 4090, both with NVIDIA compute capability 86 (sm86).

A.2.3 Artifact structure. The artifact's structure can be represented as the following:

```
|-- AGAThA
|   |-- src
|   |   `-- kernels
|   `-- test_prog
|-- dataset
|-- docker
|-- misc
`-- output
```

- `AGAThA/src`: source code for AGAThA.
- `AGAThA/test_prog`: test program to use AGAThA's kernels for sequence alignment.
- `dataset`: includes sample input dataset.
- `docker`: scripts and Dockerfile for launching docker.

- `misc`: miscellaneous code for outputting AGAThA's kernel execution time.
- `output`: directory for outputs such as alignment score and kernel execution time.

A.2.4 Setup. We recommend building and launching a docker image with the following scripts.

```
$ cd docker
$ bash build.sh
$ bash launch.sh
```

AGAThA's source code can be built using the following command lines.

```
$ cd AGAThA
$ bash build.sh
```

A.2.5 Datasets. A sample of the HiFi HG005 dataset used in this work is provided in `dataset/`. Other (custom) datasets can be used as input, as long as there is a reference and query file, and both are formatted as `.fasta` files. The input files should both follow the format below:

```
>>> 1
ATGCN...
>>> 2
TCGGA...
```

Given the two input files, AGAThA will output the alignment score between a pair of sequences from each file. Note that each input file should have an equal number of reference and query strings. `.fasta` files can be download from various sources such as GenBank [35] or projects such as 'Genome in a Bottle' [37].

A.2.6 Running AGAThA. Using `AGAThA.sh` script, the following options can be used for AGAThA.

```
-a  the match score
-b  the mismatch penatly
-q  the gap open penalty
-r  the gap extension penalty
-z  the termination threshold
-w  the band width in the score table
```

This script stores the alignment scores in `output/score.log`, and the total kernel execution millisecond time is stored in `output/time.json`.