FastZ: Accelerating Gapped Whole Genome Alignment on GPUs

Sree Charan Gundabolu
Purdue University
West Lafayette, Indiana, USA
sgundabo@purdue.edu

T. N. Vijaykumar
Purdue University
West Lafayette, Indiana, USA
vijay@ecn.purdue.edu

Mithuna Thottethodi
Purdue University
West Lafayette, Indiana, USA
mithuna@purdue.edu

ABSTRACT

Recognizing the importance of whole genome alignment (WGA), the National Institutes for Health maintains LASTZ, a sequential WGA application. As genomic data grows, there is a compelling need for scalable, high-performance WGA. Unfortunately, high-sensitivity, 'gapped' alignment which uses dynamic programming (DP) is slow, whereas faster alignment with ungapped filtering is often less sensitive. We develop *FastZ*, a GPU-accelerated, gapped WGA software which matches gapped LASTZ in sensitivity. *FastZ* employs a novel *inspector-executor* scheme in which (a) the *lightweight* inspector elides DP traceback except in common, extremely short alignments, where the inspector performs limited, *eager* traceback to eliminate the executor, and (b) *executor trimming* avoids unnecessary work. Further, *FastZ* employs *register-based cyclic-buffering* to drastically reduce memory traffic, and *groups DP problems by size* for load balance. *FastZ* running on an RTX 3080 GPU and our multicore implementation of LASTZ achieve 111x and 20x speedups over the sequential LASTZ, respectively.

CCS CONCEPTS

• **Applied computing → Bioinformatics**.

ACM Reference Format:
Sree Charan Gundabolu, T. N. Vijaykumar, and Mithuna Thottethodi. 2021. FastZ: Accelerating Gapped Whole Genome Alignment on GPUs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3458817.3476202

1 INTRODUCTION

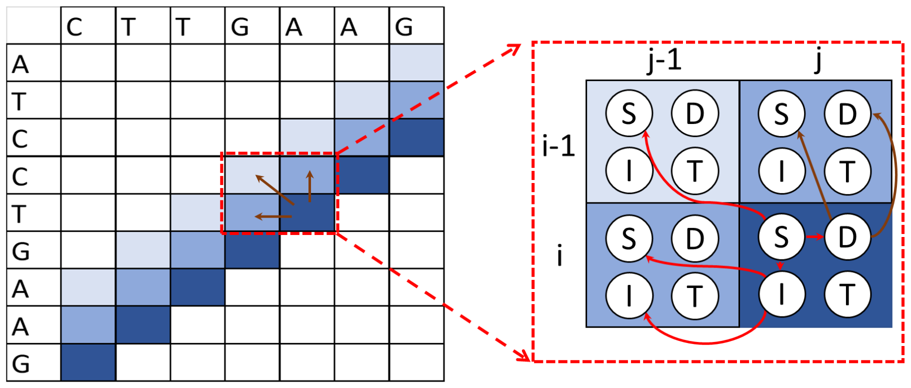
Gene sequence alignment [2, 9, 11, 22, 34] is one of the basic computational kernels of genomics. Such alignment is used in many contexts including genome assembly (from fragmented reads) and comparative genomics (comparing whole genomes of organisms). Specifically, comparative genomics examines the alignment between whole genomes of two or more organisms to answer many key evolutionary and genetic questions [10]. Reflecting the broad interest in such alignment, the National Institutes of Health (NIH) maintains servers for whole genome alignment (WGA), specifically LASTZ [11], as a computational service for the genomics community [24]. Our focus is WGA and not genome assembly (e.g., [1, 14, 33, 36]).

Genomic data continues to grow with (1) increasing numbers of fully-sequenced genomes, (2) genome assemblies of more complex life forms (*e.g.*, the genome of *D. simulans* has orders of magnitude more base pairs than that of *Salmonella enterica*, a simpler life form), and (3) faster, cheaper ways to capture genomic data [13, 20, 26]. As such, there is a compelling need for scalable, high-performance WGA. Further, there is a well-understood trade-off of sensitivity for performance. For instance, SegAlign [8] proposes GPU optimizations for the version with *ungapped* filtering, which is faster but finds fewer, high-scoring alignments than the *gapped* version which includes insertions and deletions. Because our goal is high-performance WGA *without* trading off sensitivity, we consider only the *gapped* version of LASTZ.

LASTZ employs a 'seed-and-extend' approach wherein short sequences that are exact matches (i.e., 'seeds') are extended with gaps, if necessary, to form longer, higher-scoring matches. The local alignment algorithm used in LASTZ for the seed extension [9] is a variant of the Smith-Waterman algorithm [34] which is a dynamic programming (DP) algorithm for assigning scores to matches. Recent work [37] has focused on acceleration of gapped WGA using custom ASICs (or FPGAs at a lower performance point), which remains unavailable for most genomics users until research proposals become products. In this paper, we show that off-the-shelf GPUs can achieve high-performance, gapped WGA. To that end, (1) we identify the key workload characteristics and performance bottlenecks, and (2) based on those insights we develop *FastZ*, a GPU-accelerated, drop-in replacement for gapped LASTZ. There are also older GPU-accelerated Smith-Waterman implementations for gapped extension (e.g., [38]) against which we compare.

The WGA workload can be characterized in two ways. First, there are a large number of seed extensions that must be computed. Individual seed-extensions are independent of one another and can be computed in parallel (although some work-reduction optimizations are possible if they are computed in sequential order). The distribution of work across these seed extensions is highly skewed. A vast majority of seed extensions have little work because they result in low-scoring, short alignments. For example, over 97% of seed sites result in alignments no longer than 128 base pairs. In contrast, the high-scoring alignments can be orders of magnitude longer (e.g., tens of thousands of base pairs).

Second, the Smith-Waterman computation accounts for most of the execution time (e.g., $> 99\%$) of gapped LASTZ. The DP scoring matrix is a key data structure that is populated progressively with optimal solutions to smaller problems which are then used to identify optimal solutions to larger problems. Figure 1 shows the Smith-Waterman DP matrix for the two sequences shown along the row and column. The recurrences on the right fully define the



$$I_{i,j} = max\begin{cases} I_{i,j-1} + s_e \\ S_{i,j-1} + s_o + s_e \end{cases}$$

$$D_{i,j} = max\begin{cases} D_{i-1,j} + s_e \\ S_{i-1,j} + s_o + s_e \end{cases}$$

$$S_{i,j} = max\begin{cases} I_{i,j} \\ D_{i,j} \\ S_{i-1,j-1} + s_{X_i,Y_j} \end{cases}$$

Figure 1: Dependencies and parallelism in Smith-Waterman

computation for each matrix cell. Each matrix position (i,j) holds information about the optimal score of a partial alignment of the two sequences up to and including the i^{th} ad j^{th} base pair, respectively. For each cell, the algorithm tracks the overall score (S), the score assuming a gap insertion (I) and deletion (D), and traceback pointers (T), which enable the reconstruction of the precise alignment.

Such DP algorithms have several key characteristics, some of which are relevant to GPUs.

(1) The DP algorithm computes the scores and traceback pointers for a *much larger* alignment space beyond the point of the *optimal* alignment to make sure that no higher-scoring alignment is missed. For example even though more than 97% of the alignments are shorter than 128 base pairs, more than 90% of searches explore alignments as long as 5700 base pairs (including gaps).

(2) The DP computation performs only a modest amount of work per byte accessed (e.g., four comparisons and five additions requiring five cell reads and six cell writes, as seen in the recurrences in Figure 1). When parallelized, the lopsided compute-to-bandwidth ratio results in a memory-bandwidth bound algorithm. Darwin [36], Darwin-WGA's [37] predecessor, emphasizes memory footprint but does not mention memory bandwidth-boundedness.

(3) Unlike traceback data, which must be preserved till the end of the DP algorithm to reconstruct the alignment, the scores are read fewer than five times soon after being written and then are dead.

Left unaddressed, the memory accesses along the DP matrix diagonals would degrade memory bandwidth because the discontiguous accesses cannot be coalesced. However, there are well-known mitigations (e.g., Feng *et al.* [38]) which transform the layout to ensure that the diagonal elements are contiguous and can be coalesced, Nevertheless, the low memory-to-compute ratio remains a challenge.

FastZ employs two high-level strategies for *memory* to address this challenge. First, the optimal alignment length, and hence the sizes of the DP and traceback matrices, are unknown a priori, and at the same time dynamic memory allocation in GPU kernels is slow [12, 18, 23, 35, 40]. Consequently, *FastZ* employs the well-known *inspector-executor approach* [30] where a lightweight *inspection* of the (larger) alignment search-space quickly determines the optimal alignment lengths without any dynamic memory allocation for the DP matrix; the *executor* completely computes the (much smaller) actual alignment via memory allocation at kernel launch for traceback using the alignment lengths from the inspector. Second, *FastZ* employs additional *memory* optimizations (1) to identify and eliminate a large fraction of the DP matrix accesses in both the inspector and executor, and (2) to elide global memory accesses to the traceback matrix in the common case of short alignments.

FastZ's key contributions are as follows.

- We observe that the traceback state is needed only for the optimal alignment and not for the larger alignment search space. Accordingly, the inspector elides traceback state tracking to be lightweight, with one exception (eliding the DP matrix allocation is next). Thus, the inspector not only eliminates unnecessary memory accesses but also reduces the memory footprint enabling more parallelism (i.e., more threads).
- The exception is the common case of extremely short alignments (e.g., 16-base pairs or fewer in length) where the inspector performs limited, *eager* traceback to entirely eliminate the executor. Eager traceback eliminates approximately 80% of all DP computation from the executor. Further, the traceback state is small enough (e.g., limited to 16x16) to fit in the GPU's Shared Memory or L1 cache, and does not generate memory traffic.
- Using the optimal alignment information from the inspector, *FastZ* employs *executor trimming* to compute the DP matrix and traceback *only* for the much smaller optimal alignment.
- As stated above, the DP matrix scores are needed only temporarily for computing other matrix elements. Accordingly, *FastZ* significantly reduces the DP matrix memory traffic by using a *cyclic-use-and-discard* buffering scheme of producing values for one diagonal by using the values in the two preceding diagonals in a cyclic pattern. Moreover, the three-diagonal state is small enough to be held in registers; thus eliminating a vast majority (97%) of memory accesses, and

any dynamic memory allocation for the DP score matrix. This optimization is used in both the inspector and executor.

- Finally, because of the variation in alignment lengths, intermingling short alignments with long alignments in the same GPU kernel leads to load imbalance and under-utilization. *FastZ* groups seed extension tasks into size bins based on the alignment lengths (known from the inspector) to mitigate such load imbalance.

We implement and evaluate *FastZ* on Nvidia's RTX 3080 (Ampere), QV100 (Volta), Titan X (Pascal) GPUs. Our implementation produces identical (or occasionally longer) alignments as NIH's gapped LASTZ. *FastZ* and our multicore implementation of LASTZ, which require no custom hardware, achieve 111x and 20x speedups over (the sequential) LASTZ, respectively.

2 BACKGROUND AND IMPLICATIONS FOR *FASTZ*

Whole genome alignment typically proceeds as a three-stage computation. In the first stage, a lightweight exact-match search identifies short sequences (19 base pairs long) to serve as seed sites. Because the seed search results in numerous seeds, the second stage filters the seed sites to obtain a shorter list of promising seed sites. Finally, the third stage extends the filtered seed sites using the Smith-Waterman dynamic programming (DP) algorithm for sequence alignment with affine gap penalties [9].

2.1 Baseline LASTZ optimizations

LASTZ supports both 'gapped' alignment, which allows for gap insertions and deletions, and 'ungapped' alignment, which looks for exact matches at the filtering stage. In the final stage, both versions use gapped extension. However, in ungapped filtering, many seeds that lead to high-scoring alignments may be dropped resulting in lower sensitivity, as also recognized in other work [37]. For brevity, we refer to LASTZ with ungapped filtering as 'ungapped LASTZ' (even though it uses gapped seed extension *after* the filtering stage). The more sensitive, gapped version is preferred for its higher quality results (more, longer, higher-scoring alignments) whereas the less sensitive, (but faster) ungapped version is preferred when run-time is the primary constraint. Figure 2 illustrates this tradeoff by comparing the alignments found by the ungapped and gapped variants of LASTZ when comparing subsequences of *C. elegans* and *C. briggsae* genomes that each contain one million seeds. Each alignment is shown as a point in the scatter-plot based on the alignment length (in number of base pairs, X-axis) and alignment score (Y-axis). The gapped version finds more, longer, higher-scoring alignments as visible in the graph. For example, the gapped version finds more than twice as many alignments with score exceeding 10,000 than the ungapped version (41 versus 17).

The recently-proposed SegAlign [8] accelerates the ungapped extension as part of the filtering whereas *FastZ* accelerates the gapped version for high-quality alignments. In the gapped version of LASTZ, which is a sequential implementation, more than 99% of the time is spent in the DP component, 1% of which is for traceback analysis. For example, we profiled an alignment of the first chromosome of *C. elegans* and *C. briggsae* using LASTZ (the high-sensitivity variant without the ungapped filtering) using AMD μProf v3.4 profiler. We found that one function (`ydrop_one_sided_align`) that performs the DP computation accounts for over 99.75% of the execution time. Thus, the DP component is our focus.

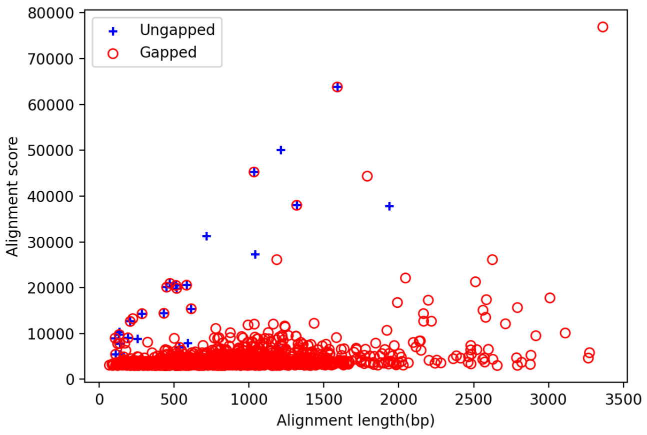
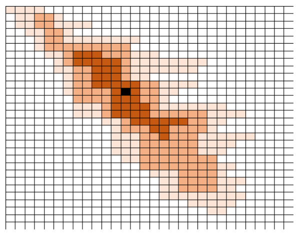


Figure 2: Gapped versus ungapped alignments

The default operation of LASTZ's gapped Smith-Waterman does not evaluate all the cells of the DP matrix. Where possible, LASTZ reduces the compute work without affecting the quality or length of the alignment. Figure 3(a) illustrates the operation of the DP scoring matrix as a score heat map (darker shades imply higher scores) where only a fraction of the full matrix is explored. In evaluating the matrix in the horizontal dimension (along rows), if the score of a cell drops below a threshold (relative to the highest observed score), the rest of the cells on that row are pruned. In the vertical dimension, if *all* the scores of a row fall below the threshold, subsequent row exploration is also pruned. Though this pruning of later cells, captured in the `y-drop` algorithm, fundamentally relies on LASTZ's sequential nature and is not amenable *fully* to parallel implementations, both *FastZ* and Darwin-WGA achieve a conservative approximation wherein most of the areas below the threshold are pruned even in the parallel implementations.

Furthermore, in the filtering stage, Darwin-WGA employs a heuristic based on Banded Smith-Waterman [7] which limits the search space to a bounded band around the diagonal of a fixed width, where the optimal alignment is likely to be found. Many insertions and deletions would mean straying away from the diagonal which is likely to be sub-optimal. However, the optimal solution may not always be found within the band. In this paper, we pursue the optimal solution and not the potentially suboptimal banded approach (explained in Section 3.4). In addition, LASTZ terminates an ongoing seed extension upon reaching a previously-discovered alignment because it is not profitable to combine the prior and current alignments; if it were, the prior alignment would have expanded to include the current one. Because this work reduction also relies on LASTZ's sequential nature, *FastZ*, being parallel, forgoes this optimization while still being significantly faster.

After the alignment search concludes, the traceback stage computes the alignment (including gaps) from the highest-scoring cell as shown in Figure 3(b). Consequently, the traceback space is much smaller than the full search space. *FastZ* exploits this difference in its inspector-executor approach.


(a) Alignment search: Score heatmap

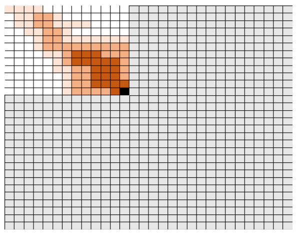

(b) Traceback of highest scoring alignment

Figure 3: Smith-Waterman operation

2.2 Memory-boundedness, Dependencies and Parallelism

From the gapped Smith-Waterman recurrences shown in Figure 1, we make four key observations.

(1) In a single seed extension, consider the computation-to-memory-access ratio for an arbitrary cell of the DP matrix (say i, j), as shown in Figure 1. The operation requires five memory reads to previously computed cells, five additions and four comparisons (for the *max* operator in Figure 1), and three memory writes (accesses to $S_{i,j}, I_{i,j}, D_{i,j}$ scoring matrices). Further, to reconstruct eventually the alignment via traceback, there are three additional writes for the traceback pointers in each matrix. The overall ratio of memory accesses to compute operations (9) to memory accesses (11) is less than 1. Thus, the workload becomes memory-bound with more parallelism; and compute acceleration will not offer any further speedups unless the memory-bandwidth bottleneck is ameliorated.

(2) In extending a single seed, the (*intra-seed parallelism*) lies along the diagonals (Figure 1). As discussed in Section 1, the irregular accesses along the diagonal is a known problem for which there are well-known data layout transformations of the DP matrix to make the accesses contiguous (e.g., [38]). The layout transformations provide a one-to-one mapping of the original i, j coordinate of a matrix cell to a new i', j' coordinate such that the elements originally along an anti-diagonal lie along rows in the transformed coordinate system. For example, Figure 4 shows such a transformation using $i' = i + j$ and $j' = j$ as the transformation functions. Parallel accesses of locations that are along the anti-diagonal in the logical DP matrix (see shaded cells in Figure 4) are in physically contiguous rows in the transformed layout (and thus can be coalesced on modern GPUs). Such transformed layouts require some padding in the corners, which results in an increase in memory footprint. However, the small increase in memory is more than offset by the improved access pattern. This transformation is necessary for performance but far from sufficient.

(3) Although the footprint seems large at first glance ($M \times N$ matrices for sequences of length M and N, respectively), the shallow dependencies in the DP matrix ensures that only a small fraction of the DP matrix needs to be preserved. For example, in Figure 1, when computing the dark shaded cells, only the two adjacent diagonals (lighter shaded) are needed. *FastZ* exploits this observation via cyclic-use-and-discard buffering within registers to elide most memory accesses.

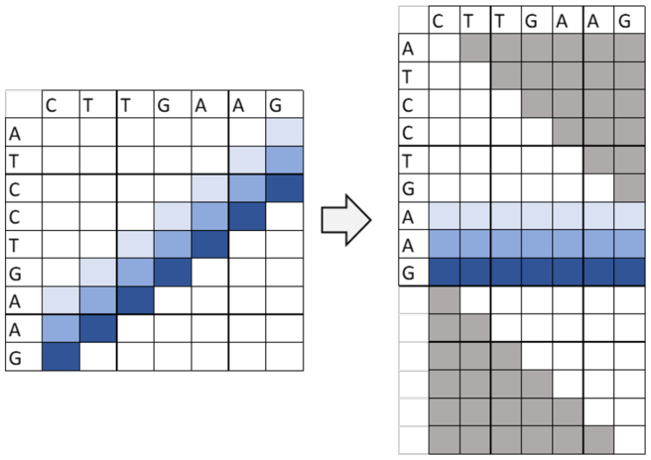


Figure 4: Transformed layout for improved memory behavior

(4) The traceback pointers must be preserved to reconstruct the alignment. However, *FastZ*'s inspector elides traceback altogether. *FastZ*'s executor preserves the traceback only for the highest-scoring alignment (and not the larger search space) avoiding superfluous traceback state.

2.3 Previous work

Darwin-WGA [37], an ASIC accelerator design, employs a specialized systolic array to exploit the parallelism along the DP matrix diagonal. *FastZ*, on the other hand, employs the previously-proposed data-layout transformation to achieve coalescing accesses on off-the-shelf GPUs. Darwin-WGA uses hardware acceleration for seed filtering, banded alignment, and traceback. Specifically for banded-alignment, Darwin-WGA uses a heuristic approach based on Banded Smith-Waterman [7]. In contrast, *FastZ* uses exact filtering, as we discuss later in Section 3.4.

Finally, Darwin-WGA uses hardware tiling (inherited from Darwin [36]) wherein the scoring matrix is not spilled to memory except for an inter-tile overlap region, which reduces the memory bandwidth demand. By using fixed hardware tiles, Darwin-WGA avoids memory allocation in the accelerator while the host may handle allocation for the overlap regions, In contrast, *FastZ* employs cyclic-use-and-discard to elide most memory accesses to the DP score matrices and elides traceback in inspector.

Darwin-WGA's **(and *FastZ*'s)** parallel implementations, cannot exploit LASTZ's work reduction based on previously-found alignments (i.e., the final outputs of previously computed seed extensions) determined sequentially in LASTZ but concurrently in Darwin-WGA and *FastZ*. Instead, Darwin-WGA achieves work-reduction via some score-based heuristics which may change the obtained alignments. In contrast, *FastZ* gives up some work reduction to avoid changing the alignment boundaries while still achieving significant speedups.

GPU acceleration for a single Smith-Waterman DP computation has been proposed [38]. However, the WGA problem is composed of millions of DP computation instances (one for each seed extension). The previous work exploits the limited single-problem parallelism whereas *FastZ* focuses on the significant multi-problem parallelism.

3 *FASTZ*

The traditional one-pass execution used in WGA algorithms interacts poorly with GPUs due the wide distribution of alignment lengths. On one hand, allocating memory for the worst case alignment lengths reduces parallelism since fewer seed extensions can be accommodated within the available memory. On the other hand, dynamic memory allocation on GPUs is slow [12, 18, 23, 35, 40]. To address this problem, *FastZ* replaces the one-pass approach with a two-pass inspector-executor approach [30].

FastZ's design is driven by the two goals, each of which has its own complications. The first goal is to split the functionality into a lightweight *inspector* that gathers run-time information to reduce significantly the amount of work and memory footprint in the subsequent *executor* phase. The second goal is to improve GPU memory performance and maximize parallelism, while operating within the GPUs' memory footprint and bandwidth limits. The rest of this section describes *FastZ*'s design to achieve these two goals.

3.1 Inspector-executor design of *FastZ*

Recall from Section 2.1 that the alignment searches a large search-space around short alignments to ensure that nearby, high-scoring alignments are not missed. Recall that Figure 3(a) shows the scores as a heatmap. The inspector explores the larger search-space to find the cell with the highest score (black cell in Figure 3(a)). The executor then performs a full computation up to (and including) the highest-scoring cell but completely avoids the computations in the rest of the search space, as shown in Figure 3(b).

FastZ's approach fundamentally requires the inspector to be lightweight; if the inspector is as heavyweight as the full computation, the additional executor would make the performance strictly worse.

3.1.1 **Inspector**. To make the inspector lightweight, *FastZ* reduces the inspector's memory bandwidth demand and footprint, which indirectly improves parallelism by enabling more concurrent DP problems. Specifically, the inspector leaves the heavy work of capture of the traceback state to the executor, with one exception (explained next). This lightweight inspector is our first contribution. The inspector precisely identifies the DP matrix cell which yields the optimal alignment. This information is crucial for the executor.

In the inspector, *FastZ* assigns a single seed extension DP task to a warp. The lanes within a warp effectively compute the independent cells along a DP matrix diagonal. However, because of the diagonal layout transformation, the elements of a diagonal are contiguous (see Section 2.2). While the degree of this intra-DP parallelism, especially for the common case of short alignments, is lower than what a GPU can exploit, the much higher degree of inter-DP parallelism is more than sufficient to keep a GPU busy (e.g., a million seed extensions). Accordingly, multiple seed extensions (DP problems) are concurrently run (as warps within a threadblock as well as multiple threadblocks) to exploit GPU multithreading. This simple strategy captures both intra- and inter-DP problem parallelism.

The load balancing optimization of binning the DP problems by alignment lengths obtained from the inspector is available only for the executor and not the inspector. Fortunately, because the inspector is lightweight, load imbalance is far less severe in the inspector and is alleviated further by well-known techniques, such as CUDA *streams* [25].

3.1.2 Avoiding executor redundancy in the common case via eager traceback. As attractive as the inspector-executor approach is in reducing unnecessary memory traffic and increasing parallelism, the approach does incur redundant computation for each seed-extension. To minimize the redundancy *without* worsening memory traffic, *FastZ* entirely eliminates the executor in the common case of extremely short alignments via limited, *eager traceback* in the inspector. This inspector traceback exception is our second contribution.

To achieve this traceback, *FastZ* augments the inspector to track a small (16×16) traceback matrix. The limited case of any extremely short alignment, that falls entirely within this range, can immediately perform the traceback; thus entirely avoiding the executor. Alignments that exceed this range will be reevaluated in the executor stage. One may think that the short alignments caught by this technique would be uninteresting because they would be low scoring alignments anyway. However, LASTZ and *FastZ* perform left and right extensions of any seed site separately before combining the two for the final extension. Thus, a short left (right) alignment cannot be eliminated a priori as it may yield a high-scoring final alignment after combining.

Eager traceback is particularly attractive due to its high benefits and low costs. The benefits are high because a significant majority of seed extensions ($> 80\%$) are short and avoid redundant computation in the executor. The costs are low because the 16x16 traceback state is small enough to fit in GPUs' L1 cache (or Shared Memory). Thus, the inspector's memory traffic reduction is not degraded.

3.1.3 **Executor**. *FastZ* employs *executor trimming* where, using the precise information of the optimal alignment, the executor computes the DP matrix and tracks traceback information only for the much shorter optimal alignment and not the larger search space explored in the inspector (our third contribution). On its own, the traceback state is dwarfed by the score matrices. As such, one may think that this improvement is minor. However, *FastZ* also reduces the memory accesses to the score matrices in both the inspector and executor, as we discuss in the next subsection.

Another benefit of the inspector's information is that the executor can statically allocate the DP and traceback matrices according to the exact size needed. Because most seed extensions are small, the executor achieves a huge reduction in memory footprint. More importantly, given the limited memory on GPUs, precise allocation enables *FastZ* to pack many more seed extensions into one kernel, thus maximizing parallelism. The executor stage inherits many of the inspector's optimizations such as the parallelism model (i.e., one seed extension per warp). However, there are key differences mainly due to the traceback state which is not captured by the inspector but by the executor. We now focus on the traceback state.

The traceback state, which records the choices made at each cell, is consumed only after the DP matrix is computed fully. During the DP computation, traceback state is written, but never read. To minimize this write memory bandwidth demand of writing traceback data (within the constraint that traceback data *cannot* be discarded like scoring data), *FastZ* employs two further optimizations.

First, we observe that the traceback data for any given DP matrix need not be a full byte. Traceback data records the identity of the maximum among the various score choices (see Figure 1). The recurrences for the matrices *I*, *D*, and *S* select the maximum among 2, 2, and 3 choices respectively, which can be identified using 1, 1, and 2 bits, respectively. Because modern systems disallow sub-byte access, we compress the trace state from all three scoring matrices into a single byte.

Second, to achieve cache block-level aggregation of trace write-back, we consolidate the trace of multiple cells to fill a cache block in Shared Memory which is written to memory in one single write. While naively writing to the L1 cache may also achieve such coalescing, GPUs' small L1 caches are prone to unpredictable evictions which may prevent such coalescing. *FastZ*'s approach of consolidation in Shared Memory deterministically avoids such failures.

Traceback Parallelism. Despite being small (less than 1%), the traceback can limit *FastZ*'s net speedup to 50× (Amdahl's Law limit assuming a speedup of around 100x for the main DP computation except the traceback). Indeed, because *FastZ* speeds up the DP computation by as much as 100×, the Amdahl's Law limit is a real problem for *FastZ*. Unlike the DP computation, the traceback of a given seed extension has no internal parallelism. As such, the only available parallelism is inter-seed parallelism – the traceback of one seed extension is independent of other seed extensions.

Given that *FastZ*'s parallelism employs one warp per seed extension for the DP computation, we use one thread of the same warp to perform the traceback computation after the DP computation is complete. With this structure, *FastZ* effectively exploits inter-seed traceback parallelism which is captured both by multithreading and execution on different SMs. Because of the presence of multiple warps per SM, the degree of multithreading (and the resultant latency hiding) remains the same as that for the DP computation.

One may think that the loss of intra-warp parallelism is a setback for traceback. However, the purpose of traceback parallelization is merely to push the Amdahl's law limit to a point where the DP component speedup is not dampened. For this limited purpose, fully-multithreaded inter-SM parallelism (which can vary from 28-way on a Titan X(Pascal), 68-way on the RTX 3080(Ampere) to 80-way on a Volta (QV100)) is more than adequate. A 1% component when parallelized 28-ways is under 0.04%, which pushes the Amdahl's Law limit away from the DP component's speedup of 100×.

3.2 Minimizing memory bandwidth demand via *Cyclic Use-and-Discard*

Ideally, we want to minimize writing the DP scores back to memory. Recall from Section 2.2 that the live state is limited to three consecutive diagonals of data because whenever one diagonal of cells in the matrix are fully computed, there is no further use of cells that are at the third-previous diagonal and beyond. The live state being limited to three diagonals of DP matrix data is not enough to avoid severe cache pollution due to the scan-like access pattern of the large matrix. With parallel execution of multiple problems, the number of scans are amplified; and large numbers of large scans result in unnecessary spills to (and reloads from) memory. To address this challenge, *FastZ* uses *cyclic use-and-discard buffers* that can hold three diagonals of the DP matrix, as shown in Figure 5 (our fourth contribution). In

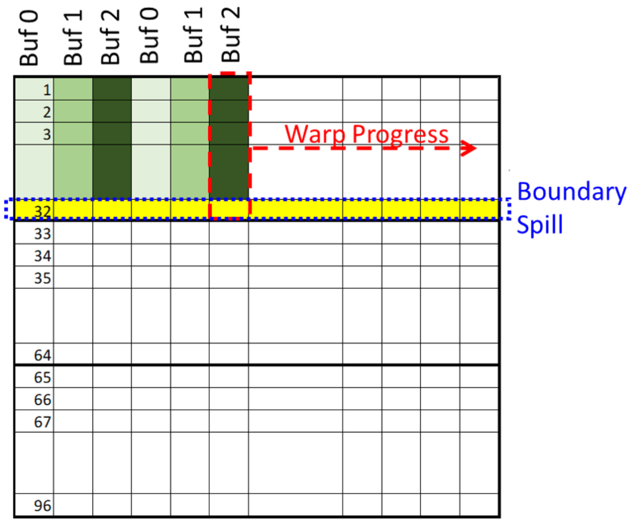


Figure 5: *FastZ*'s Cyclic Use-and-discard Buffer Management

Figure 5, the original DP matrix diagonals have been transformed into rows as shown in Figure 4, but shown transposed as columns to match the familiar DP row-major traversal. *FastZ* cyclically reuses the three buffers by overwriting (and hence discarding) the oldest diagonal as a new diagonal is computed.

Because the scores are not needed (only the position of the maximum score is remembered) even for the final solution, *FastZ* uses cyclic use-and-discard optimization for both the inspector and executor phases. For the executor, the elimination of the scoring matrix writes reduces the memory bandwidth demand by 92%. (The remaining 8% accounts for the traceback state which *must* be written back to memory in order to reconstruct the alignment.)

FastZ's strategy for parallelization on GPUs directly impacts buffer management. Because *FastZ* parallelizes only a warp-wide strip of a given DP matrix computation (Section 3.1.1), the cyclic buffers hold warp-wide strips of three original diagonals at a time (Figure 5). Consequently, the strip boundary issues must be addressed. Specifically, the boundary cells of the strip cannot be discarded as they hold live state that will be needed when the computation proceeds to the next strip in the diagonal. As such, the state will have to be preserved and written out to memory to be re-loaded later for the next strip of the same diagonals. The other, non-boundary cells may be discarded. Because such boundary conditions affect only one cell out of the 32 cells handled in a warp, the bandwidth reduction is effectively more than 96% (=31/32).

We considered both GPU Shared Memory and GPU registers as the physical locations for the cyclic use-and-discard buffers. The aggregate state for the three diagonals for all the threads that can be scheduled on a single Streaming Multiprocessor (SM) exceeds Shared Memory capacity of current GPUs. For example, 2 thread-blocks each with 64 warps of 32 threads, each requiring 36 bytes (3 scores of 4 bytes each), corresponds to 144 KB of Shared Memory storage. In contrast, the per-thread storage of 36 bytes can be accommodated easily in the register space of each CUDA thread.

Finally, our parallelization requires each thread in a warp to access live state produced by other threads (neighboring DP matrix cells), which can be a challenge when using thread-private registers. Fortunately, CUDA includes fine-grained register exchange instructions that allow for the necessary cross-thread communication. As such, *FastZ* houses the cyclic use-and-discard buffers in registers.

3.3 Load-balancing via inspector-determined alignment length-binning

In addition to optimizing memory traffic, *FastZ* exploits information from the inspector to achieve load balance on the GPU (our fifth contribution). The GPU's bulk-synchronous approach, wherein a kernel completes only when the threadblocks on all the SMs complete, can cause load imbalance when long alignments are intermingled with short alignments within the same kernel. Accordingly, *FastZ* uses the precise knowledge of the *length* of the alignment (from the inspector) to bin the seed extensions by length. *FastZ* bundles the seed extensions in each bin into its own kernel which ensures load balance. We use four bins with bin boundaries at 512x512, 2048x2048, 8192x8192, and 32,768x32,768 respectively. An optimal alignment found at DP matrix cell i, j is placed in the smallest bin in which the alignment is contained. Our benchmarks did not need larger bins, but one could add bins using a similar 4x scaling factor if needed.

3.4 Implementation

GPU performance can be fragile without careful tuning. To that end, *FastZ* employs other well-known optimizations.

Streams. As mentioned in Section 3.1, each seed extension takes a variable amount of time even in the inspector. (The length-binning optimization cannot be used at the inspector stage as the alignment lengths are unknown.) To achieve good scheduling of thread blocks to SMs, we use CUDA streams [25]. Each stream's kernels launch asynchronously with respect to other streams; thus preventing long-running kernels on one SM from blocking execution on other SMs.

Work Reduction in Parallel Implementations. Recall from Section 2.1 that LASTZ performs two work-reduction optimizations: (1) pruning when the score along a row or column falls below a threshold, and (2) termination upon reaching previous alignments. For the first optimization, because *FastZ* cannot use the score information that is being produced concurrently without significant communication overhead, *FastZ* uses previously-completed scores along the row or column at the cost of some extra work (if the completed score is already below the threshold, it is safe to stop the exploration). Though conservative, this approximation achieves good pruning. LASTZ's sequentially-dependent second optimization is infeasible in *any* parallel implementation. As such, each seed extension in *FastZ* (like those in Darwin-WGA) uses only score-based work reductions which do not depend on other alignments. The benefits of inter-seed parallelism outweigh the costs of the modest loss in work reduction. Consequently, *FastZ* explores the same or a strict superset of basepairs as LASTZ, resulting in the same or occasionally longer alignments (at most 0.005% of alignments across all benchmarks with a median length increase of 5.5x).

Control divergence. The recurrence computation uses the *max* operator which is implemented using data-dependent branch instructions, likely leading to control flow divergence. However, the control divergence is limited to only a few paths each with only a few instructions. For example, computing the maximum of two operands (for the I and D matrices) requires at most two control paths, each with only a few instructions (e.g., 2-4). Even though SIMT execution has to execute all possible control paths, the overhead is limited.

We implemented *FastZ* in CUDA as a kernel launched by LASTZ. In our prototype, LASTZ identifies seeds which are extended by *FastZ*. We ensure correctness by comparing unmodified LASTZ's alignments with ours.

Multicore Implementation. As a comparison, we also develop a variant of LASTZ that exploits multiprocess parallelism on multicores. Our implementation partitions the set of seeds where each partition runs in a sequential process. *FastZ*'s innovations are not relevant for multicore execution. *FastZ*'s inspector-executor innovation is needed for slow GPU dynamic memory allocation, not for multicores; Multicores do not have enough CPU registers for *FastZ*'s cyclic use-and-discard buffers. *FastZ*'s eager traceback decreases inspector-executor's overheads, which is not relevant without inspector-executor. Finally, *FastZ* uses size binning to mitigate within-kernel load imbalance in GPUs, which is non-existent in multicores. Even the well-known layout transformation (Figure 4) is not relevant to multicores which do not have SIMT's warp parallelism and traverse the DP matrix in a memory-friendly row-wise order (i.e., no need for anti-diagonal parallelism).

Table 1: Genomes

Common Name	Species	Basepairs
Nematodes	*C. elegans* (*chr1*)	15,072,434
	C. briggsae (*chr1*)	15,455,979
	C. elegans (*chr2*)	15,279,421
	C. briggsae (*chr2*)	16,627,154
	C. elegans (*chr3*)	13,783,801
	C. briggsae (*chr3*)	14,578,851
	C. elegans (*chr4*)	17,493,829
	C. briggsae (*chr4*)	17,485,439
	C. elegans (*chr5*)	20,924,180
	C. briggsae (*chr5*)	19,495,157
Fruit flies	*D. melanogaster* (*chr2R*)	25,286,936
	D. pseudoobscura (*chr2*)	30,794,189
Mosquitoes	*A. albimanus* (*chrX*)	12,318,379
	A. atroparvus (*chrX*)	17,503,697
	A. gambiae (*chrX*)	24,393,108

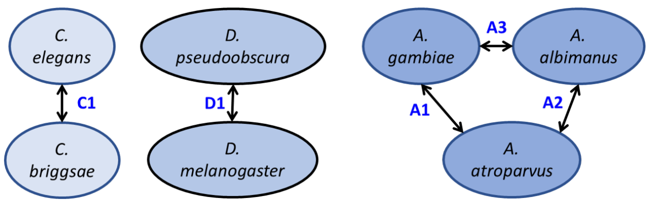


Figure 6: Pairs used for whole genome alignment

4 METHODOLOGY

Genomes and pairwise alignment: In our evaluation, we use the genomes of seven species shown in Table 1, including two nematodes, two fruit flies, and three mosquitoes. Table 1 includes the number of base pairs in each species' genome in specific chromosomes. As shown in Figure 6, while we perform alignments across all chromosomes of *C. elegans* and *C. briggsae*, we align selected individual chromosome pairs from among the fruit fly and mosquito genomes.

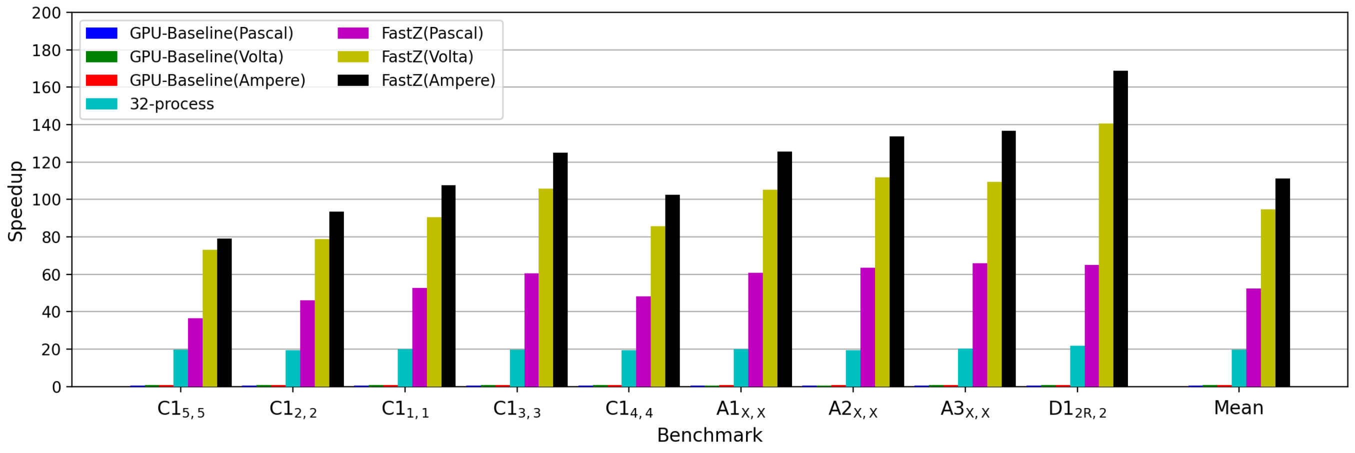


Figure 7: *FastZ* Performance

We use the pair labels shown in Figure 6 to refer to the specific chromosome pair being aligned. For example, the pairwise alignment of chromosome *chr1* of *C. elegans* and *C. briggsae* genomes is labeled $C1_{1,1}$. Similarly, we perform a total of nine pairwise alignments – five between nematodes ($C1_{j,j}$ where $j = 1..5$), one among fruit flies ($D1_{2R,2}$), and three among mosquitos ($A1_{X,X}$, $A2_{X,X}$, and $A3_{X,X}$). To achieve manageable run-times for the sequential version, we report the performance of seed extension for a million seed sites. We choose a random seed site from the entire chromosome and use a million seed sites from the immediate vicinity of the chosen seed site. This choice preserves seed-site density so that LASTZ maximally benefits from its work-reducing optimizations (Section 2.1).

Baseline Configurations: While we compare *FastZ* against the publicly-available LASTZ sequential version, we also include a multi-process-based multicore implementation and a GPU implementation in the comparison. Our multicore version uses coarse-grained, inter-seed parallelism across cores; each process performs LASTZ's default DP computation for a subset of the seeds. We also developed a GPU implementation of the Feng *et al.* scheme [38] for a single Smith Waterman computation that exploits intra-seed parallelism (Section 2.3). The GPU implementation (1) exploits the data layout transformation for improved memory coalescing behavior, and (2) uses synchronization across warps to preserve diagonal-to-diagonal dependencies. All GPU-based implementations use CUDA Runtime version 11.0.

***FastZ* performance:** *FastZ*'s execution time includes the inspector stage (without any traceback data collection) and the executor stage (with full traceback). The executor uses four bins for load balancing, as described in Section 3.3.

Evaluation platforms: We evaluate *FastZ* on three GPUs: (1) an Nvidia Titan X (Pascal) GPU with 28 streaming multiprocessors (SMs) and 12 GB of memory, (2) a V100 GPU with 80 SMs and 32 GB of memory, and (3) an RTX 3080 (Ampere) GPU with 68 SMs and 10 GB of memory. We also evaluate LASTZ (the original sequential version and our multi-process version) on an AMD Ryzen 3950x with 16 high-performance cores, and 32 GB of memory.

5 RESULTS

5.1 Performance

Figure 7 shows the speedup over the sequential LASTZ baseline (Y-axis) for each of the benchmarks (groups of bars on the X-axis). In addition, the rightmost group of bars shows the mean speedups over all the benchmarks. The benchmarks are ordered left to right by decreasing load-balancing bin4 counts (Table 2, discussed later) to show performance trends clearly. In each group, the individual bars (from left to right) show the speedup achieved by the GPU Baseline (parallelizing single Smith Waterman DP [38] described in Section 2.3) on the three GPU configurations (Pascal, Volta, and Ampere), 32-process multicore configuration, and finally *FastZ* on the three GPUs.

The first three bars (GPU Baseline on the Pascal, Volta and Ampere GPUs) are barely visible because the baselines achieve slowdowns (18% to 43% slower) relative to sequential LASTZ across all the benchmarks. The GPU baseline's inadequate parallelism due to handling only one seed extension at a time, and the costly inter-SM synchronization needed for parallelizing one seed extension over multiple SMs are the reasons behind the slowdowns.

The multicore configuration with 32-process parallelism achieves 20x mean speedup over all benchmarks (with similar speedups for individual benchmarks). Although the seed extensions can be trivially parallelized across 32 processes, the speedup does not scale up linearly to 32x due to memory bandwidth limits. Recall from Section 3.4 that multicores cannot benefit from *FastZ*'s innovations; hence our techniques cannot improve these speedups.

FastZ achieves significantly higher speedups across all benchmarks with overall mean speedups of 43x, 93x, and 111x on the Pascal, Volta, and Ampere generation GPUs, respectively. The trend shows that speedups are higher for later generation GPUs. The Pascal generation TitanX GPU in particular is also handicapped by its fewer SMs (28) compared to significantly more SMs for the Volta V100 (80) and Ampere RTX3080 (68) GPUs. Further, to put *FastZ*'s and the multicore's speedups in perspective, we note that the Titan X, for example, has 3584 1-wide lanes running at 1 GHz with a 3-MB L2 cache compared to the 16-core multicore with 4-wide superscalar cores running at 3,5 GHz and a 64-MB L3 cache. While the GPU

Table 2: Alignment length distribution of 1 million seeds

Benchmark	**Eager Traceback**	**Bins**			
		1	2	3	4
$C1_{5,5}$	776453	222663	651	208	25
$C1_{2,2}$	771776	227211	810	187	16
$C1_{1,1}$	757731	240979	1101	177	12
$C1_{3,3}$	764269	234331	1225	165	10
$C1_{4,4}$	759240	240040	603	114	3
$A1_{X,X}$	816817	182879	240	62	2
$A2_{X,X}$	814634	185014	300	50	2
$A3_{X,X}$	819519	179978	426	76	1
$D1_{2R,2}$	812578	187408	13	1	0

has more memory bandwidth, *FastZ* cuts the bandwidth demand and does not use GPU's bandwidth, whereas the CPU has more cache benefiting the memory-bound LASTZ.

Finally, for a given GPU, *FastZ*'s speedups vary across the benchmarks due to differences in the optimal alignment lengths and the number of long alignments. More, shorter alignments result in higher speedups because shorter alignments naturally induce less load imbalance in the inspector which dominates *FastZ*'s execution time, as discussed next.

5.2 Execution Time Breakdown

Figure 8 shows the breakdown of *FastZ*'s normalized execution time (Y-axis, stacked bars) for our benchmarks (X-axis) on the Ampere GPU. For each benchmark, the execution time is broken down into three components: *inspector*, *executor*, and *other*. As expected, inspector is the largest component accounting for around two-thirds (and as high as 79%) of the execution time for most benchmarks. The executor accounts for a smaller fraction (approximately 10%). Although more heavy-weight than the inspector, the executor is invoked only for a small subset of seed extension problems that survive eager-traceback. Finally, the remainder (*other* in Figure 8) accounts for other work such as reading the anchor points, sequence files, allocating memory, copying all these over to the GPU, reading final alignments, sorting the anchors into bins based on alignment length and copying the eager traceback-surviving anchor points for the executor. Note that the *other* component is a much smaller fraction for sequential LASTZ. It is only after *FastZ*'s acceleration, which vastly reduces the execution time of the inspector and executor, that the *other* component is seen as a non-negligible fraction.

The differences in execution time breakdown among individual benchmarks is entirely explained by the same reason that explains benchmark speedup differences for a given GPU in Figure 7 – the number of long alignments. Table 2 shows the distribution of the alignment lengths for our 1 million seeds in each benchmark. The lengths are binned as per our load balancing bins from Section 3.3: upto 16 base pairs in eager traceback, 16-512 base pairs in bin1, 512-2K base pairs in bin2, 2K-8K base pairs in bin3, and 8K-32K base pairs in bin4. The table shows that 75-80% of the alignments are 16 base pairs or fewer. A vast majority of the remaining alignments fall within bin1. The rest of the bins are quite small. However, bin4's long alignments, though the least, are the primary reason for the differences in speedups (Figure 7) and execution time breakdown (Figure 8), The lower the bin4 count for a benchmark (e.g., $D1_{2R,2}$), the smaller the inspector and executor run time components in Figure 8 and the lower the inspector load imbalance, and the higher the benchmark speedup in Figure 7.

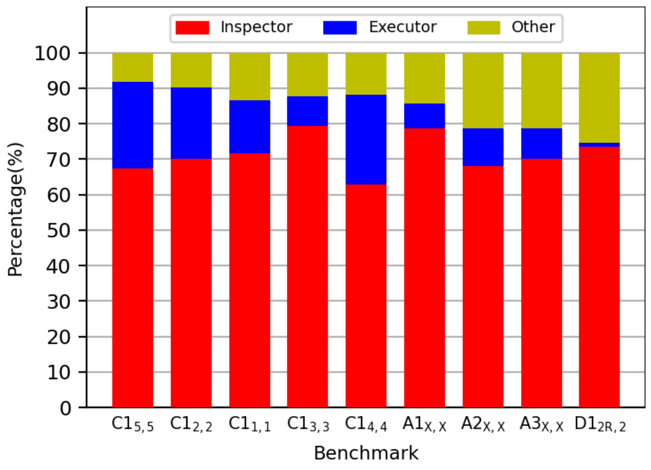


Figure 8: Execution time breakdown (Ampere GPU)

5.3 Isolating the impact of *FastZ*'s optimizations

Figure 9 isolates the impact of the individual optimizations of *FastZ*. To that end, we start with a variant of *FastZ* that uses the inspector-executor approach with length-binned load balancing and the light-weight inspector. We do not include a configuration that excludes load balancing which would result in high slowdowns due not only to load imbalance but also to per-problem memory allocation in the absence of per-bin allocation. We then *progressively add* one by one *cyclic use-and-discard* buffer management, *eager traceback* for short alignments, and *executor trimming*. Finally, we also isolate the impact of CUDA streams. Figure 9 shows the speedup (Y-axis) of these progressively composed schemes (X-axis, individual bars) for our three GPUs (X-axis, groups of bars). Progressive addition of optimizations implies that any bar in Figure 9 includes *all* the optimizations of the bars to its left (within the same group). The penultimate bar with all optimizations is *FastZ*. To isolate the impact of streams, the last bar shows *FastZ* using a single CUDA stream (*FastZ*-single-stream) whereas all the previous bars use 32 streams.

For each GPU, we show the mean speedup across all benchmarks as the results are qualitatively similar for each individual benchmark. In each case, the following common observations hold. First, *FastZ*'s load balancing on its own, yields small slowdowns to modest speedups – between 8% slowdown (on the Pascal) and 2.8x speedup (on the Ampere) over the sequential LASTZ baseline. Second, each optimization is valuable. The addition of cyclic use-and-discard buffers (green bars) boosts the speedup to 4.7x, 6.1x, and 17x on the Pascal, Volta and Ampere architectures, respectively. Eager traceback for seed extensions whose optimal alignment lies within a 16x16 DP matrix further boosts performance to 15x, 21x, and 46x on the Pascal, Volta and Ampere GPUs, respectively. The penultimate bar, which adds executor trimming wherein the executor trims its own execution footprint based on the inspector's knowledge of the optimal alignment, is *FastZ*; and achieves speedups of 43x, 93x, and 111x on the Pascal, Volta and Ampere GPUs, respectively. Finally, using a single stream reduces the mean performance by 1.7x, 1.7x, and 2.4x on the Pascal, Volta, and Ampere GPUs, respectively.

Because we progressively add *FastZ*'s optimizations, the later optimizations improve performance on top of a higher-performing base.

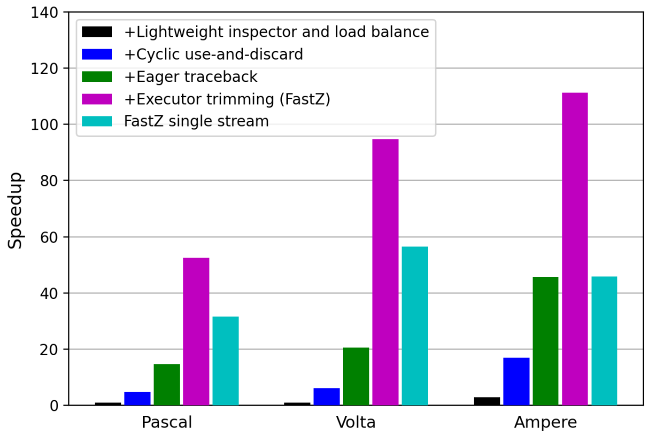


Figure 9: Isolating the Impact of *FastZ*'s optimizations

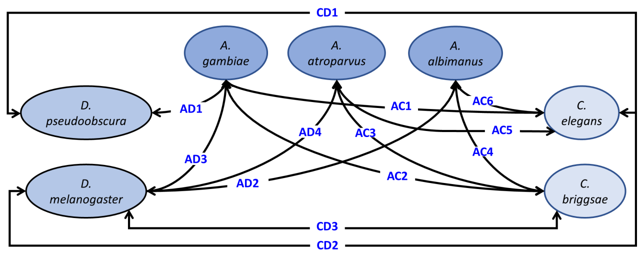


Figure 10: Cross-genus pairs for whole genome alignment

Improvements on a larger base create the optical illusion that the later optimizations offer higher speedups (e.g., a 50x to 100x jump visually looks larger than a 2x to 4x jump, even though the speedup increment in the two cases is the same factor of 2). Analyzing the relative speedups, we found that no single optimization is responsible for a large fraction of *FastZ*'s gains. Rather, the contributions of lightweight inspector and load balancing, cyclic-use-and-discard, eager traceback, and executor trimming are somewhat comparable (1.4x, 5.8x, 3x, and 3.4x mean speedups across the GPUs).

5.4 *FastZ*'s performance for dissimilar genomes

Thus far, we have focused on genome alignments within the same genus (e.g., among nematodes, mosquitoes, and fruit flies), where the genomes are expected to align quite well. To understand *FastZ*'s performance on dissimilar genome alignments, we evaluate *FastZ*'s speedups over LASTZ for several cross-genus comparisons as shown in Figure 10. As before, the benchmark names are derived from the edge labels in Figure 10 with the chromosome numbers specified in the subscript. We verified that these comparisons are indeed dissimilar as no alignment falls in the two largest size bins.

Figure 11 shows the speedups (Y-axis) over LASTZ for each of our benchmarks (X-axis) on Ampere. The rightmost bar shows the mean speedup across all benchmarks. We observe that *FastZ*'s speedups are higher for dissimilar alignments (mean speedup 137x) than those for the similar alignments (mean speedup 111x). Dissimilar genomes have fewer high-score alignments which results in relatively more time being spent in the (faster) inspector compared to similar genomes whose longer alignments require relatively more time to be spent in the (slower) executor.

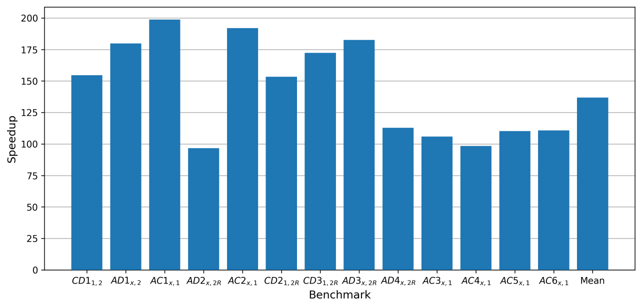


Figure 11: *FastZ* performance on Ampere for dissimilar (cross-genus) sequence alignment

6 DISCUSSION

Multi- GPU/node extension: *FastZ*'s approach lends itself to multi-GPU (and if necessary, multi-node) acceleration because the seeds can be partitioned easily. As such, each partition can be assigned to different GPUs and/or nodes for parallel execution. We defer multi-GPU/multi-node implementations for future work.

Remaining bottlenecks: *FastZ* reduces the number of memory accesses significantly; but to understand the impact of such memory access reduction, we consider the three phases of the application separately. First, in the inspector phase, *FastZ* requires only 3 FP words (12 bytes) of output (the S, D, and I scores) for every 32×9 (9 = 4 comparisons + 5 additions) operations of a GPU warp, yielding an operational intensity of 24 ops/byte. (Recall from Section 3.2 that only the boundary lane writes the scores back to memory.)

Second, in the executor phase, *FastZ* requires one byte of traceback state for every cell of the DP matrix, which results in an additional 32 bytes (beyond the 12 bytes needed for the scores) accessed per $32 \times 9 = 288$ operations of a GPU warp for an operational intensity of 6.5 (= 288/(12 + 32)) ops/byte.

The nominal peak compute- and memory-bandwidth of the RTX 3080 GPU are 29.77 TFlops/s and 760 GB/s, yielding a threshold operational intensity of 39 ops/byte. This nominal intensity has to be derated for the maximum achievable compute bandwidth due to branch-divergence (arising from the *max* operator in the Smith-Waterman recurrence). Examining the instruction-level overheads yields a derating factor of 2.56, because the 9 operations expand to 23 operations under SIMT divergence. Thus, the true threshold is 39/2.56 = 15.2 ops/byte which means that the inspector is slightly compute-bound and the executor is slightly memory-bound. Note that *FastZ* would have been deeply memory-bound without our optimizations (roughly 0.75 (= 9/12) op/byte in inspector and 0.69 (= 9/13) op/byte in executor).

Finally, because of the sequential nature of the traceback phase, *FastZ* leverages only inter-SM parallelism (and not warp-parallelism). This stage remains memory-bound but is a small fraction of execution time (< 0.2%). Recall that the traceback is parallelized only to prevent it from becoming an Amdahl's law bottleneck.

7 RELATED WORK

Acceleration of sequence alignment: There are software-only multicore libraries which focus on the core Smith-Waterman functionality (e.g., Parasail [4]). However, we focus on the full WGA application rather than on the Smith-Waterman kernel. As such *FastZ* and Parasail are not comparable in functionality. Moreover, *FastZ* exploits GPU-scale parallelism (with hundreds of concurrent threads) compared to Parasail's multicore-scale parallelism (tens of threads).

LOGAN [39] implements a subset of Smith-Waterman functionality which does not include traceback. It is not comparable to *FastZ* or LASTZ which produce alignments and not just scores. Finally, LOGAN minimizes memory footprint by reusing memory for the anti-diagonals in the DP matrix, but writes out each anti-diagonal to memory and thus does not reduce memory bandwidth of the scoring matrices – a key optimization of *FastZ.*

CUDAlign [31] eschews the traditional *seed-filter-extend* alignment and instead performs a computationally-expensive global Smith-Waterman on the extremely long sequences. This approach results in tens of petabytes of DP matrix state and requires hundreds of GPUs to compute. While impressive as a big-iron computing achievement, CUDAlign is not a practical option for most users. In contrast, *FastZ* accelerates practical alignment based on the seed-extend framework that is accessible to practitioners.

SegAlign [8], which we have discussed previously in Section 2, accelerates only the seed-and-filter stages of LASTZ. Because SegAlign targets ungapped extension in the filter stage, its acceleration applies only to the lower-sensitivity, variant of LASTZ with ungapped filtering. In contrast, *FastZ* targets gapped extensions like the high-sensitivity variant of LASTZ.

On the hardware front, Darwin [36] and Darwin-WGA [37] are two recently-proposed accelerators that rely on hardware-based tiled execution of the Smith-Waterman algorithm. There are other FPGA-based Smith-Waterman acceleration proposals as well [5, 6, 29]. *FastZ* shows that much of the advantage of such custom hardware accelerators can be captured entirely in software (e.g., eliminating DP matrix reads/writes) running on commodity GPUs.

Inspector-Executor approach: The *inspector-executor* is a general dynamic optimization to exploit run-time information that is not known at compile time [30]. The approach has found uses in many areas such as graph/tree traversals [19] and sparse linear algebra kernels [16]. *FastZ* uses the approach to address similar challenges. WGA on a GPU can benefit from run-time information like alignment length (e.g., to minimize unnecessary traceback tracking via executor trimming, and to improve load balancing by binning by length).

Other applications of Smith-Waterman sequence alignment: Smith-Waterman and its variants are used in several sequence alignment applications including de novo and reference-guided genome assembly from long [14, 33] and short reads [3, 15, 17, 27, 28, 32]. MUMmer4 [21], which uses Banded Smith-Waterman, has been parallelized for multicores. As such, MUMmer4 achieves speedups comparable to our multicore implementation of LASTZ, which is significantly slower than *FastZ* on GPUs. Moreover, *FastZ* implements full Smith-Waterman and not the banded variant which does not guarantee optimal alignments (Section 2.1).

Some of *FastZ*'s techniques that optimize a single seed extension are directly applicable in the above contexts. For example, the inspector-executor approach, and cyclic use-and-discard buffer management are applicable to any Smith-Waterman application. However, some of *FastZ*'s other techniques depend on specific application characteristics such as the distribution of alignment lengths and the number of independent alignments/seed-extensions. For example, our eager traceback, is driven by the observation that 75%-80% of the seed extensions result in extremely short alignments. If other applications differ in those characteristics, those specific aspects of *FastZ* may have to be revisited and potentially redesigned. This limitation is not unique to *FastZ*; existing sequence alignment software packages are also specialized for specific use cases.

8 CONCLUSION

Whole genome alignment (WGA) is an important application for comparative genomics, with NIH-supported servers offering (WGA) as a service. Unfortunately, researchers have to choose between high performance at the cost of low sensitivity, ungapped alignment, or highly sensitive, gapped alignment that is slow. Prior attempts to overcome this dilemma have proposed either GPU-based solutions that are only modestly better, or have proposed ASIC accelerators which will not be available to practitioners till they are commercially produced.

FastZ fills this void and achieves high speedups on off-the-shelf GPUs for high-sensitivity, gapped alignment using the Smith-Waterman dynamic programming (DP) algorithm. The challenges are (1) the workload is memory bandwidth-bound, (2) the final alignments for different seeds vary widely in length, with short alignments being a vast majority, requiring dynamic memory allocation which is slow in GPUs, and (3) the algorithm searches through longer matches to find the shorter optimal alignment. *FastZ*'s five key innovations are based on the well-known inspector-executor approach. (1) The inspector stage discovers the optimal alignment by searching longer matches but remains lightweight by eliding DP traceback with the following exception. (2) To eliminate the redundancy between the inspector and executor in the common case of extremely short alignments, *FastZ* employs limited, *eager traceback* in the inspector without degrading the inspector's memory traffic efficiency while completely removing those alignments from the executor. (3) An efficient, 'trimmed' executor stage computes the DP matrix and tracks traceback data only for the shorter optimal alignment determined by the inspector. Both inspector and executor avoid dynamic memory allocations. (4) *FastZ*'s design employs a novel cyclic use-and-discard buffer management strategy which drastically shrinks the DP matrix footprint and eliminates a majority of DP matrix memory accesses in both stages by fitting the buffers in registers. (5) Finally, *FastZ* bins the executor tasks by their alignment lengths as a load balancing measure to minimize GPU under-utilization arising from intermingling long and short alignments in a single GPU kernel.

We implemented and evaluated *FastZ* on Nvidia's Titan X (Pascal), V100 (Volta) and RTX 3080 (Ampere) GPUs. Our implementation, which requires no custom hardware and produces identical (or occasionally longer) alignments, achieves 43x, 93x, and 111x speedups over (the sequential) LASTZ on the Pascal, Volta and Ampere GPUs, respectively. In comparison, our multicore implementation of LASTZ on a 16-core AMD Ryzen 3950x achieves 20x over LASTZ. *FastZ*'s high speedups on commodity, off-the-shelf hardware has the potential to help accelerate progress in genomics.

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 (10 2019). https://doi.org/10.1186/s12859-019-3086-9

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. 1990. Basic local alignment search tool. *Journal of Molecular Biology* 215 (1990), 403–410.

[3] Jacek Blazewicz, Wojciech Frohmberg, Michal Kierzynka, and Pawel Wojciechowski. 2013. G-MSA — A GPU-based, fast and accurate algorithm for multiple sequence alignment. *J. Parallel and Distrib. Comput.* 73, 1 (2013), 32–41. https://doi.org/10.1016/j.jpdc.2012.04.004 Metaheuristics on GPUs.

[4] Jeff Daily. 2016. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics* 17 (12 2016). https://doi.org/10.1186/s12859-016-0930-z

[5] Lorenzo Di Tucci, Kenneth O'Brien, Michaela Blott, and Marco D. Santambrogio. 2017. Architectural optimizations for high performance and energy efficient Smith-Waterman implementation on FPGAs using OpenCL. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 716–721. https://doi.org/10.23919/DATE.2017.7927082

[6] Wael Abou El-Wafa, Asmaa G. Seliem, and Hesham F. A. Hamed. 2016. Hardware Acceleration of Smith-Waterman Algorithm for Short Read DNA Alignment Using FPGA. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 2. 604–605. https://doi.org/10.1109/COMPSAC.2016.127

[7] Norman E. Gibbs, William G. Poole, Jr., and Paul K. Stockmeyer. 1976. A Comparison of Several Bandwidth and Profile Reduction Algorithms. *ACM Trans. Math. Softw.* 2, 4 (Dec. 1976), 322–330. https://doi.org/10.1145/355705.355707

[8] Sneha D. Goenka, Yatish Turakhia, Benedict Paten, and Mark Horowitz. 2020. SegAlign: A Scalable GPU-Based Whole Genome Aligner. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 39, 13 pages. https://doi.org/10.5555/3433701.3433752

[9] Osamu Gotoh. 1982. An improved algorithm for matching biological sequences. *Journal of Molecular Biology* 162, 3 (1982), 705 – 708. https://doi.org/10.1016/0022-2836(82)90398-9

[10] Ross C Hardison. 2003. Comparative Genomics. *PLOS Biology* 1, 2 (11 2003). https://doi.org/10.1371/journal.pbio.0000058

[11] Robert S. Harris. 2007. *Improved Pairwise Alignment of Genomic Dna*. Ph.D. Dissertation. Pennsylvania State University, USA.

[12] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W. Hwu. 2010. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. In *2010 10th IEEE International Conference on Computer and Information Technology*. 1134–1139. https://doi.org/10.1109/CIT.2010.206

[13] Illumina [n.d.]. Illumina: Next Generation Sequencing. https://www.illumina.com/science/technology/next-generation-sequencing.html.

[14] Abdul Rafay Khan, Muhammad Tariq Pervez, Masroor Ellahi Babar, Nasir Naveed, and Muhammad Shoaib. 2018. A Comprehensive Study of De Novo Genome Assemblers: Current Challenges and Future Prospective. *Evolutionary Bioinformatics* 14 (2018), 1176934318758650. https://doi.org/10.1177/1176934318758650 PMID: 29511353.

[15] Petr Klus, Simon Lam, Dag Lyberg, M. Cheung, G. Pullan, Ian McFarlane, G. Yeo, and B. Lam. 2011. BarraCUDA - a fast short read sequence aligner using graphics processing units. *BMC Research Notes* 5 (2011), 27 – 27.

[16] Christopher D. Krieger and Michelle Mills Strout. 2012. A Fast Parallel Graph Partitioner for Shared-Memory Inspector/Executor Strategies. In *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*.

[17] W. Langdon, Brian Yee Hong Lam, J. Petke, and M. Harman. 2015. Improving CUDA DNA Analysis Software with Genetic Programming. *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* (2015).

[18] C. Lauterbach, Q. Mo, and D. Manocha. 2010. gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries. *Computer Graphics Forum* 29, 2 (2010), 419–428. https://doi.org/10.1111/j.1467-8659.2009.01611.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01611.x

[19] Jianqiao Liu, Nikhil Hegde, and Milind Kulkarni. 2016. Hybrid CPU-GPU scheduling and execution of tree traversals. 1–2. https://doi.org/10.1145/2851141.2851174

[20] Glennis Logsdon, Mitchell Vollger, and Evan Eichler. 2020. Long-read human genome sequencing and its applications. *Nature Reviews Genetics* 21 (06 2020). https://doi.org/10.1038/s41576-020-0236-x

[21] Guillaume Marçais, A. Delcher, A. Phillippy, Rachel Coston, S. Salzberg, and A. Zimin. 2018. MUMmer4: A fast and versatile genome alignment system. *PLoS Computational Biology* 14 (2018).

[22] 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. https://doi.org/10.1016/0022-2836(70)90057-4

[23] David M. Nicol. 1990. *Inflated speedups in parallel simulations via malloc ()*. National Aeronautics and Space Administration, Langley Research Center.

[24] NIH: LASTZ on Biowulf [n.d.]. NIH HPC: LASTZ on Biowulf. https://hpc.nih.gov/apps/LASTZ.html

[25] nvidia [n.d.]. CUDA Toolkit Documentation (Stream Management). https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html

[26] pacbio [n.d.]. PACBIO: Single Molecule, Real-Time (SMRT) Sequencing. https://www.pacb.com/smrt-science/smrt-sequencing/.

[27] Jacopo Pantaleoni and Nuno Subtil. [n.d.]. NVBIO: nvBowtie Sequence Aligner. ([n. d.]). http://nvlabs.github.io/nvbio/nvbowtie_page.html

[28] Ren. 2019. GPU accelerated sequence alignment with traceback for GATK HaplotypeCaller.

[29] Enzo Rucci, Carlos Garcia, Guillermo Botella, Armando De Giusti, Marcelo Naiouf, and Manuel Prieto Matias. 2018. SWIFOLD: Smith-Waterman implementation on FPGA with OpenCL for long DNA sequences. *BMC Systems Biology* 12 (11 2018). https://doi.org/10.1186/s12918-018-0614-6

[30] J. H. Saltz, R. Mirchandaney, and K. Crowley. 1991. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.* 40, 5 (1991), 603–612. https://doi.org/10.1109/12.88484

[31] 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. https://doi.org/10.1109/TPDS.2016.2515597

[32] M. Schatz, Cole Trapnell, A. Delcher, and A. Varshney. 2007. High-throughput sequence alignment using Graphics Processing Units. *BMC Bioinformatics* 8 (2007), 474 – 474.

[33] Korbinian Schneeberger, Stephan Ossowski, Felix Ott, Juliane D. Klein, Xi Wang, Christa Lanz, Lisa M. Smith, Jun Cao, Joffrey Fitz, Norman Warthmann, Stefan R. Henz, Daniel H. Huson, and Detlef Weigel. 2011. Reference-guided assembly of four diverse Arabidopsis thaliana genomes. *Proceedings of the National Academy of Sciences* 108, 25 (2011), 10249–10254. https://doi.org/10.1073/pnas.1107739108

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

[35] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg. 2012. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *2012 Innovative Parallel Computing (InPar)*. 1–10. https://doi.org/10.1109/InPar.2012.6339604

[36] Yatish Turakhia, Gill Bejerano, and William J. Dally. 2018. Darwin: A Genomics Co-Processor Provides up to 15,000X Acceleration on Long Read Assembly. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 199–213. https://doi.org/10.1145/3173162.3173193

[37] Y. Turakhia, S. D. Goenka, G. Bejerano, and W. J. Dally. 2019. Darwin-WGA: A Co-processor Provides Increased Sensitivity in Whole Genome Alignments with High Speedup. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 359–372.

[38] Shucai Xiao, Ashwin Aji, and Wu Feng. 2009. On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit. In *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*. 26–33. https://doi.org/10.1109/ICPADS.2009.110

[39] 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 *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 462–471. https://doi.org/10.1109/IPDPS47924.2020.00055

[40] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. 2008. Real-Time KD-Tree Construction on Graphics Hardware. *ACM Trans. Graph.* 27 (12 2008), 126. https://doi.org/10.1145/1457515.1409079