SegAlign: A Scalable GPU-Based Whole Genome Aligner

Sneha D. Goenka[§*], Yatish Turakhia[‡*], Benedict Paten[‡], Mark Horowitz[§]

[§]*Stanford University*
[‡]*University of California, Santa Cruz*
gsneha@stanford.edu, yturakhi@ucsc.edu, bpaten@ucsc.edu, horowitz@ee.stanford.edu

***Abstract*—Pairwise Whole Genome Alignment (WGA) is a crucial first step to understanding evolution at the DNA sequence-level. Pairwise WGA of thousands of currently available species genomes could help make biological discoveries, however, computing them for even a fraction of the millions of possible pairs is prohibitive – WGA of a single pair of vertebrate genomes (human-mouse) takes 11 hours on a 96-core Amazon Web Services (AWS) instance (c5.24xlarge).**

This paper presents SegAlign – a scalable, GPU-accelerated system for computing pairwise WGA. SegAlign is based on the standard seed-filter-extend heuristic, in which the filtering stage dominates the runtime (e.g. 98% for human-mouse WGA), and is accelerated using GPU(s). Using three vertebrate genome pairs, we show that SegAlign provides a speedup of up to 14× on an 8-GPU, 64-core AWS instance (p3.16xlarge) for WGA and nearly 2.3× reduction in dollar cost. SegAlign also allows parallelization over multiple GPU nodes and scales efficiently.

***Index Terms*—Whole Genome Alignment, Graphics Processing Unit (GPU), Comparative Genomics, Apache Spark.**

I. Introduction

The field of comparative genomics relies on comparing the genomes of different species. Pairwise whole genome alignment (WGA) is a computational technique that allows us to compare entire genome sequences of a pair of species – a target and a query. WGA is a fundamental operation in comparative genomics that enables us to study evolution at the DNA sequence level [1]. WGAs are primarily used for identifying *homologous* sequences, i.e., sequences that share a common ancestry in the genomes of different species [2]. This can further be applied to (i) identify where certain functional elements, such as genes and regulatory sequences, known in the genome of one species, are present in the genome of another species [3], [4], as well as to (ii) discover novel sequences of unknown function that are highly conserved across species genomes, and therefore, are likely to be biologically significant [5]. WGAs have also been used in phylogenetics [6], [7], and for reconstructing the genomes of extinct species [8], [9].

We are now entering an era of large-scale comparative genomics – nearly a thousand vertebrate species have already been sequenced and assembled (see Figure 1), and sequencing of thousands of more species is currently underway [11]–[13].

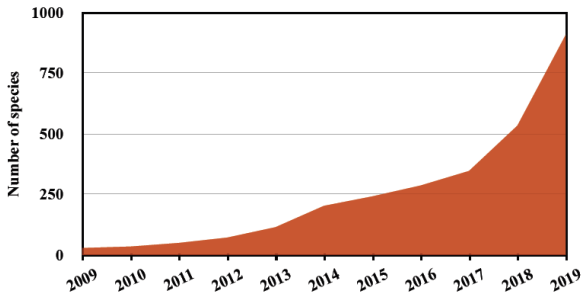


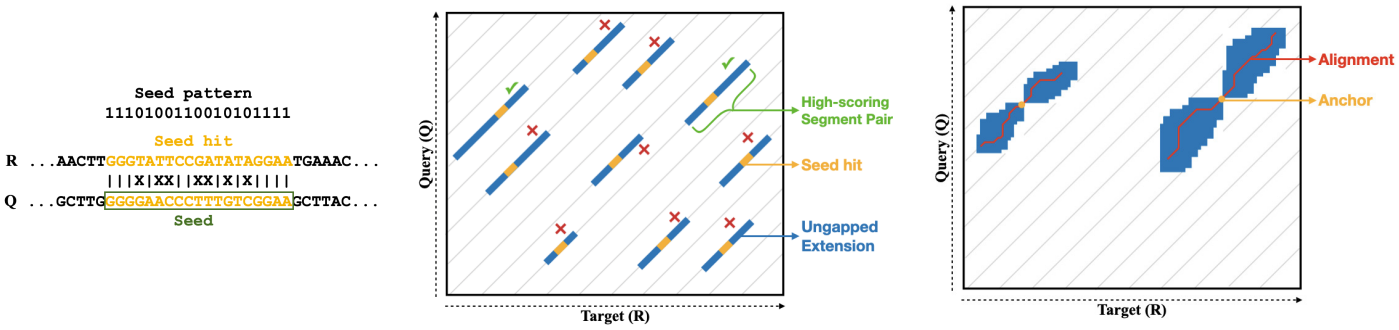
Fig. 1: Cumulative number of vertebrate genome assemblies (one assembly considered per species) available in the NCBI genome database [10] by year.

In fact, the recently proposed Earth BioGenome Project [13] aims to sequence the genomes of all living animal and plant species on earth in the next ten years. This wealth of genomic data would allow us to study evolution at an unprecedented scale and resolution, but also would pose serious computational challenges. Using privately-shared data, we estimated that over 440 CPU-years were required for approximately 1,400 WGA pairs (i.e., an average of 115 CPU-days per pair) of large genomes (>1G bases) computed by the UCSC Genome Browser group [14] (one of the largest centers supporting comparative genomics research) to date. Another recent work that created a multiple vertebrate genome alignment of over 600 species required around 200 CPU-years of computation [15], that took 3 months of wall-clock time on an 800-core AWS cluster, with around 90% of the time spent on performing pairwise WGA. We anticipate that in a few years, when thousands of new vertebrate genomes would be sequenced each year, it would become impossible even for large compute clusters to keep up with the sequencing rate using the currently available software tools.

In this paper, we describe *SegAlign* – a scalable GPU-based pairwise whole genome aligner. SegAlign is based on the standard *seed-filter-extend* paradigm [16], and provides large improvements in both speed and cost compared to LASTZ [17], the current state-of-art software tool for cross-species WGA. This paper makes the following contributions:

* These authors contributed equally to this work

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00©2020 IEEE



(a) (b) (c)

Fig. 2: An illustration of the seed-filter-extend heuristic used in LASTZ. (a) An example of the seed hit with a given seed and the default seed pattern in LASTZ, where 1 is an exact match and 0 is a don't care at that position. (b) A dot plot (not to scale) where the seed hits are shown in yellow, and the right and left gap-free extensions in blue. Only forward strand hits and extensions are shown with diagonals drawn in grey. The discarded segment pairs are marked with a red cross-mark and the high-scoring segment pairs (HSPs) with a green check-mark. (c) A dot plot for the gapped extension stage using the HSPs from (b) with the region in which the dynamic programming algorithm computes the scores shown in blue and the resulting alignment path in red.

1) We provide a GPU implementation to accelerate the bottleneck seeding and filtering stages of LASTZ (referred to as the *seed-and-filter* stage subsequently in the paper). In particular, the filtering stage has a large number of data-dependent instructions, and we propose a scheme to parallelize it efficiently for the SIMT (single instruction, multiple threads) architecture that is used in GPUs [18]. On one NVIDIA V100 GPU, we obtain 180× higher throughput on the seed-and-filter stage compared to a single-core of a modern Intel Xeon processor. We demonstrate that our implementation has high compute and memory bandwidth utilization on a GPU.
2) We extend our GPU-accelerated seed-and-filter stage to a complete seed-filter-extend tool, called SegAlign, where the extension is performed using LASTZ on CPU. SegAlign is developed using Intel's TBB library [19], which helps in efficiently scheduling and load balancing the workload on multiple CPU cores and GPU devices. We evaluated SegAlign on AWS instances using three vertebrate genome pairs and found that on p3.16xlarge instance (8 GPUs, 64 logical cores (called vCPUs)), SegAlign provides 12-14× higher performance than a parallel execution of LASTZ on the c5.24xlarge instance (96 vCPUs). Additionally, SegAlign provides nearly 2.3× savings in cost.
3) We further extend SegAlign to a multi-node implementation using the Apache Spark framework [20]. We evaluated this implementation on a cluster of GPU nodes created using AWS Elastic MapReduce (EMR) framework [21] and found that our implementation scales efficiently, with a strong scaling efficiency of around 93% and a weak scaling efficiency of around 97.8%.
4) We compared SegAlign's output to LASTZ's output for human and mouse WGA and found that SegAlign not only recovers every alignment discovered by LASTZ, but also finds a few additional alignments. SegAlign supports all output formats used by LASTZ along with a wide variety of command line options. To that end, we believe that SegAlign could serve as a drop-in replacement for LASTZ in a large number of comparative genomics pipelines.

To understand how SegAlign works, Section II provides the needed background on the WGA algorithm and NVIDIA's GPU execution model. Section III then describes the implementation details of SegAlign at both the system and the seed-and-filter kernel level. It also presents the details about SegAlign's multi-node parallelization technique using Apache Spark. Section IV describes the experimental methodology used to evaluate our system, and Section V presents the results of this evaluation. Section VI shows how SegAlign compares to related work, and Section VII concludes the paper and describes future directions for this work.

II. Background

A. Whole genome alignment algorithm (LASTZ)

Whole genome alignment (WGA) is a fundamental tool in comparative genomics to study evolution and understanding the genome function. Its primary objective is to identify the set of corresponding subsequences between the whole genome sequences of a species pair that are well-conserved (i.e. have high similarity), and therefore, are likely to serve a biological function [22], [23]. At the core of WGA algorithms is sequence alignment.

The Smith-Waterman (SW) algorithm [24] is a classic algorithm for *local* sequence alignment in genomics. It is based on dynamic programming and has a space and time complexity of $\mathcal{O}(L_R L_Q)$, where L_R and L_Q are the lengths of the input sequences. While aligning whole genomes, the

input sequences could get as long as billions of base-pairs (bp), making it intractable to compute WGA with the SW algorithm. Hence, the most common whole genome aligners, like LASTZ [17], use heuristics based on the *seed-filter-extend* paradigm, made popular by BLAST [16].

The seed-filter-extend pipeline of LASTZ computes local alignments between the target (R) and the query (Q) genome sequences in 3 stages – 1) *Seeding* 2) *Filtering* and 3) *Extension*.

Seeding. The seeding stage reduces the search space for the alignments from the entire genomes to short, localized, exact matches between the target and the query genome, known as seed hits, as shown in Figure 2. For increased sensitivity, seed hits are based on a seed pattern instead of exact matches [25], e.g. Figure 2a. A seed pattern requires a match only at some fixed positions in the seed hit.

A seed is a subsequence of length k (also known as a *k-mer*) in the query genome. Based on the seed pattern, the corresponding *k-mer* position(s) are searched along the target genome to determine the seed hits as shown in Figure 2a. Since k is small, a look-up table is created that lists all possible *k-mers* along with their positions in the target genome to reduce the computation needed for this step.

Since the seed hits span only 10s of bp, there is a high probability of finding randomly occurring matches. Hence, the seeding stage ends up with a high false-positive rate. For efficiency, most of these false-positives need to be filtered before extending them using the most compute-intensive dynamic programming step.

Filtering. The filtering stage sieves out a majority of the false-positives by extending the short matches to 100s of bp in an inexpensive, gap-free manner. The seed hit is extended along both right and left sides of the diagonal, as shown in the dot plot in Figure 2b. A dot plot is a graphical representation of a relationship, such as an alignment, between two genome sequences. Specifically, a dot (i, j) represents a base-pair at position i in the target sequence and position j in the query sequence, and a diagonal refers to a set of points with a constant difference $(i - j)$.

During each extension, the score for the base-pair at every position is calculated using the substitution score matrix, W. The cumulative score, along with the maximum score, is calculated at each position. The extension is stopped in each direction when the cumulative score falls below the maximum score by more than the X-drop value, H_x. This is known as the X-drop condition. The sequence between the maximum positions of the left extension and the right extension results in a segment pair. The score of the segment pair constructed is the sum of the maximum scores along the right and left extensions. If this score crosses the filtering threshold (H_f), the segment pair is a high-scoring segment pair (HSP), which is sent as an input to the extension stage to generate the final alignment.

LASTZ incorporates two additional filters in this stage: (i) in order to reduce the total time spent on the filtering, it skips a number of seed hits using a diagonal hashing strategy [17]; (ii) LASTZ filters out low-complexity HSPs that have a score close to the filtering threshold based on the HSP's entropy value.

Extension. The extension stage uses dynamic programming to extend the HSP to construct the final alignments that can span tens of thousands of bp as depicted in Figure 2c. The alignments with a score above the extension threshold (H_e) are written to the output.

The gapped extension step for an HSP is much more compute-heavy than a gap-free extension for the seed hit. However, the filtering ends up being the slowest stage in this pipeline since the input to it is orders of magnitude higher in size as compared to the input to the extension stage.

B. GPU execution model

NVIDIA GPUs consist of a set of Streaming Multiprocessors (SMs). During a kernel execution on the GPU, multiple *thread-blocks* are allocated to an SM. A group of 32 consecutive *threads* constitutes a *warp* — the basic unit for scheduling execution and memory accesses within a thread-block. All the threads in a warp execute the instructions in a SIMT fashion [26].

In order to maximize the utilization and efficiency of the SM, several features of the GPU execution model need to be considered: *a*) The kernel should be programmed to avoid divergent branches within a warp to utilize the SM cores efficiently. A warp remains active and keeps the compute resources busy, even if it has only a single thread active. *b*) Enough thread-blocks and warps should be provisioned to avoid under-utilization of the SM in case of long wait times within the warp execution. *c*) Global memory requests within a warp should be contiguous to utilize high DRAM bandwidth of GPUs using global memory coalescing. *d*) On-chip shared memory should be used to store small, frequently accessed data shared among the threads in a block. *e*) In order to reduce shared memory utilization, *shuffle* (`__shfl_sync()`) instructions should be used as a means for fast, inter-thread communication, within a warp.

III. Implementation Details

SegAlign uses Intel TBB library's [19] *flowgraph* utility to exploit the parallelism in the pipeline for maximum compute resource utilization while adhering to the dependencies between the stages. Figure 3 provides an overview of the stages in SegAlign. Details of the pipeline stages will be described in the subsections that follow.

In ❶, the SegAlign software reads the target (R) and the query (Q) genome sequences and loads them into the CPU DRAM. Each genome consists of multiple chromosomes (when chromosome-level assembly is not available, these are sometimes referred to as contigs, scaffolds, etc. but for brevity, we always refer to them as chromosomes). WGA for a pair of genomes consists of the combined set of alignments for each chromosome pair, that are independent of each other. For a given chromosome pair, an index table and a seed position table [27] are constructed for the target chromosome,

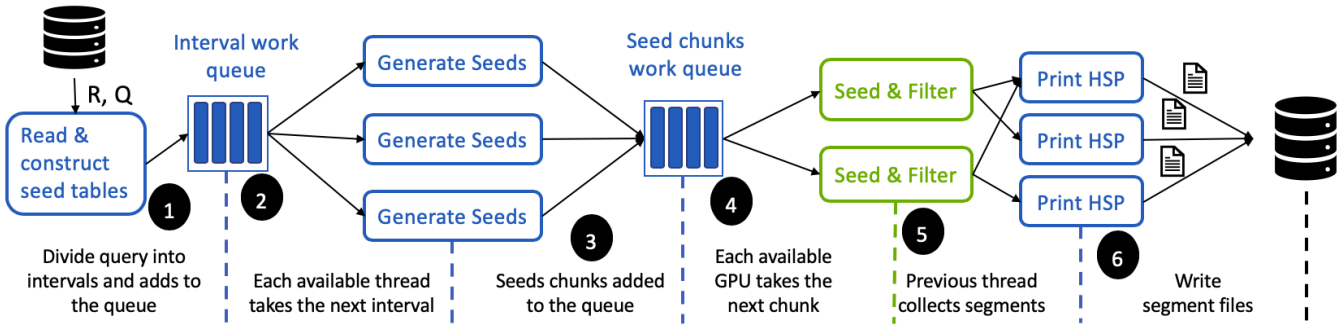


Fig. 3: Pipeline stages in SegAlign for seeding and filtering. The stages executed on CPU are marked by blue boxes and the ones executed on GPU are in green boxes.

for efficient seed position lookup in the seeding stage. In order to reduce the possible redundant computation of generating the seed position and the index table for a target chromosome multiple times, SegAlign computes the alignments for all query chromosomes before moving onto the next target chromosome. After the construction, the target chromosome sequence and the tables are copied into the GPU DRAM. In stage ❷, the query chromosome is divided into multiple intervals of size L_i (= 10Mbp by default), and each interval is added to the work queue. TBB allocates one query interval to a CPU thread. As a result, multiple threads can execute stage ❸ in parallel, as depicted by the multiple branches in Figure 3. In SegAlign, the maximum number of threads that can be spawned or the maximum number of such parallel branches is limited by the number of CPU cores on the machine. In ❸, each CPU thread breaks down the query interval further, into multiple chunks of size L_c (= 250Kbp by default). These chunks are added to a work queue in ❹, where each thread waits on a GPU for the seed-and-filter stage. The maximum number of parallel seed-and-filter stages in ❺ is the number of GPUs on the machine. Each GPU receives a chunk of seeds, finds the corresponding seed hits in the seeding stage, filters the seed hits, and generates a vector of high-scoring segment pairs (HSPs) as the output. Once the CPU thread receives this vector of HSPs for every chunk in its query interval, it combines them and prints them to a segment file in stage ❻. As soon as these files are generated, they are added to a work queue and a LASTZ process is spawned to complete the extension stage and write the alignments to the specified files.

A. Seed Position table construction

Since the construction of the seed position table is on the critical path of our pipeline (step ❶, Figure 3), we optimize this component using parallel algorithms. Our parallel construction method requires that it make two sweeps of the target sequence. First, we initialize (4^k+1) entries in the index table with zeros, where k is the size of the seed. Second, our algorithm divides the target sequence into smaller intervals and makes the first sweep of the target intervals in parallel using a parallel for loop. For each valid seed found in this sweep, the corresponding entry in the index table is incremented by 1

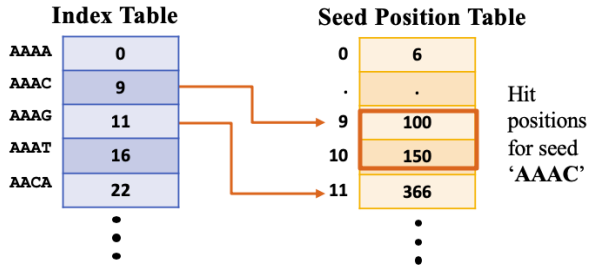


Fig. 4: An example seed position table showing a seed hit lookup pattern for an example seed 'AAAC'.

using an atomic add instruction. Now, the index table stores the seed counts for each seed. Next, a parallel prefix scan of the index table updates it to store the ending address for each seed in its corresponding entry. Finally, the second parallel sweep takes place in which the position of each valid seed is stored in the position table corresponding to address in the index, while it simultaneously decrements the address by 1 using an atomic instruction. At the end of this step, the index table stores the starting address for each seed in its corresponding entry, similar to Figure 4. Overall, this implementation results in 2-3× faster seed position table construction time on an 8-core processor than a naive, sequential implementation of the same algorithm.

B. Seed-and-Filter stage design on GPU

Seeding. The input to the seed-and-filter stage consists of a vector of the query seed position and the corresponding seed index. The number of seed hits per seed is variable. So, in order to allocate only the required amount of memory to store the seed hit positions, a GPU array, N_{hit}, is created to store the number of hits per seed in the input vector. A preprocessing GPU kernel is launched to calculate the number of seed hits per seed. The kernel allocates one seed per GPU thread. Each thread reads the position table addresses from the index table for its seed and the next. As shown in Figure 4, the difference in addresses provides the number of hits for the seed. Subsequently, the *inclusive_scan* utility from NVIDIA's

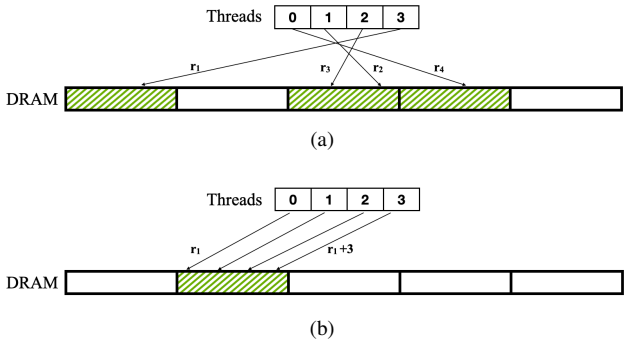


Fig. 5: (a) Assigning a single seed hit per thread results in a random memory access pattern for the thread warp. (b) Coalesced memory access pattern resulting from assigning a single seed hit per thread warp.

Thrust library [28] is used for the fast, cumulative sum of the number of hits. Based on the total number of hits, another GPU array, A_{hit}, is created to store the seed hit positions. The cumulative sum in N_{hit} ensures that the hits for a given seed are written to continuous indices in A_{hit}.

Next, the seeding kernel is launched and one seed is allocated per GPU thread. Each thread reads the target positions for the given seed from the position table and populates A_{hit} with the target positions and the common query position of the seed. These seed hits now need to be filtered using gap-free extension.

Filtering. Seed hits can be extended independently, making it an embarrassingly parallel problem. A naive implementation of the filtering stage would allocate one seed hit per GPU thread. It means that all the threads within a warp extend different seed hits. However, this choice results in an implementation that is slower than CPU due to multiple inefficiencies – (i) Consecutive seed hits considerably vary in positions over the target and the query genomes. Each thread would request bases from randomly distributed memory addresses, as depicted in Figure 5a. As described in Section II-B, uncoalesced memory accesses significantly reduce effective global memory bandwidth. (ii) Each seed hit within a warp is extended to varying lengths since the X-drop termination condition is dynamic and based on the genome sequences. This divergent execution within a warp causes poor utilization since many threads have to wait on some longer-running threads to complete.

SegAlign addresses these shortcomings by allocating one seed hit per warp and not per thread. However, gap-free extension has multiple data-dependencies which make it hard to parallelize. SegAlign extends a seed hit by the number of threads in a warp rather than extending it by one base-pair at a time. This change enables SegAlign to leverage the data-locality within seed hit extension and exploit the high GPU DRAM bandwidth. While a similar approach has been proposed in cuBLASTP [29], the differences between SegAlign and cuBLASTP will be highlighted in Section VI.

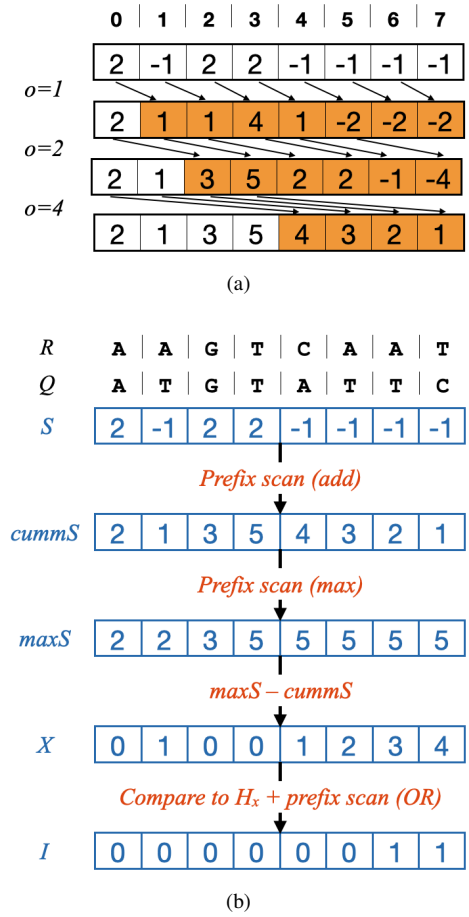


Fig. 6: Steps involved in the extension stage within a single partition. (a) Prefix scan with the add operation for an example warp size of 8. The registers updated at each stage are highlighted in orange, with arrows showing the communication between them through the `__shfl_up_sync` instruction. (b) The flow of operations in ungapped extension within the GPU kernel.

Each warp extends the seed hit along the right and left diagonal as a series of successive partitions until the X-drop criterion is satisfied.

While computing a partition, every thread in the warp requests data from consecutive positions in the target and the query sequences, as shown in Figure 5b, which results in coalesced global memory accesses. Each thread-block maintains the substitution matrix, W, in its shared memory. Each thread maintains a local register for base-pair substitution score (S), cumulative score ($cummS$), maximum score ($maxS$), X-drop measure (X), and a flag (I) to indicate whether the X-drop condition has been satisfied up to this position.

We will be describing the gap-free extension process within a partition with the help of Figure 6b. For the given example, the warp size is 8, and the substitution matrix is based on a simple constant match (score = +2), constant mismatch (score = -1) model. Each thread in a warp reads the target and

query base from the DRAM and calculates S based on the substitution matrix. Instead of a naive sequential reduction, $cummS$ is calculated using parallel prefix scan using CUDA's warp-level primitives [26], as shown in Figure 6a. Each thread reads the local register of another thread at a distance *offset*, o within the warp, using (`__shfl_up_sync`). Similarly, $maxS$ is calculated for each position using the parallel prefix scan technique with the *max* operation in place of the *add* operation used for $cummS$. Subsequently, X is concurrently calculated by the threads as the difference between the $maxS$ and $cummS$ values. X is compared to the X-drop threshold, H_x, and the results are written to the indicator, I register. A parallel prefix scan with the *or* operation ensures that the I register of the last thread of the warp will indicate if the X-drop condition has been satisfied in the current partition. The seed hit extension along a diagonal is terminated once the above condition has been met. If not, $cummS$ and $maxS$ of the last thread in the warp are stored in the shared memory for the next partition in the same direction. Since each thread in the warp is executing the same instruction, there are hardly any divergent branches (except at the ends) which leads to a high SM utilization.

Once the extensions are completed, the target and query segments between the maximum scoring positions of the extensions in each direction constitute a segment pair of length L_{sp}. The score of the segment pair, H_{sp}, is the sum of the maximum scores from extensions along the right and the left diagonal. If H_{sp} exceeds the filtering threshold, H_f, it qualifies for the extension stage as a high-scoring segment pair (HSP). Just like LASTZ, if H_{sp} is close to H_f, an additional entropy-based filtering step is added. More specifically, for such a segment pair, H_{sp} is multiplied by the Shannon entropy [30] computed using the probability vector of individual bases (A, C, G, T) in the segment pair.

An output is generated for each seed hit with that consists of: (i) a flag to indicate if the corresponding segment pair is an HSP, and (ii) accompanying data about the HSP (target start position, query start position, length, score). In the case of large genome pairs, such as human and mouse, only about 1 in 10,000 seed hits result in an HSP. As such, transferring back the output for all the segment pairs results in a high GPU-CPU communication overhead. For example, when the seed-and-filter stage was profiled for the human-mouse WGA, it showed that 18% of the time was spent on GPU to CPU memory transfer. Hence, a new array is created to efficiently gather only the output for HSPs. When the seed-and-filter stage was profiled following this optimization, it showed that the memory transfer overhead reduced to 1% of the total runtime. Also, since multiple seed hits may result in the same HSP, the *unique_copy* function from NVIDIA's Thrust library is used to remove the redundancies and transfer only the unique HSPs to the CPU. While this step adds a 5% overhead to the filtering kernel, the resulting file sizes are typically reduced by about 5×.

C. Extension stage using LASTZ

Once the HSPs for an interval are written to a segment file, a new LASTZ process is spawned to extend the HSPs to alignments. The number of concurrently LASTZ processes is limited to the number of cores to avoid (i) context-switching overhead if the rate of starting LASTZ processes is significantly higher than the rate of its completion, (ii) running out of RAM that leads to some failed LASTZ jobs.

D. Load balancing

In order to maximize the utilization of the GPU and the CPU resources, SegAlign pipelines the GPU-accelerated seed-and-filter stage with LASTZ's CPU-based extension stage, over different chromosome pair alignments. Before computing the alignment for all the pairs with a given target, the seed lookup tables and target chromosome sequences are copied to the GPU DRAM. Subsequently, each query chromosome is copied to the GPU DRAM, and once the seed-and-filter stage is completed for all its intervals, the next query chromosome begins. Concurrently, a new LASTZ process is spawned off for each interval of the previously completed query chromosome.

As the number of GPUs increases, the query chunks produced for one query chromosome are not enough to keep all the GPUs busy, especially towards the last few chunks, where most GPUs are idle and only a few GPUs are busy. This can get particularly pronounced with the high tail latencies of a few chunks (such as those containing genomic repeats) containing a large number of alignments. To overcome this, a double buffering technique is used for query chromosomes in the GPU. At any given time, there are two query chromosome sequences available in the GPU. The chunks for both the chromosomes can be added to the work queue in stage ❹ (Figure 3) simultaneously. As soon as all the chunks for one of the two chromosomes complete, the next query chromosome is written to the GPU DRAM, and the corresponding intervals are scheduled for seed generation and subsequently, the seed-and-filter stage on GPU. In this way, there is enough work available to keep the GPUs busy till all the query chromosomes are filtered for a target chromosome.

SegAlign also concatenates multiple smaller target and query chromosomes till the concatenated length for each exceeds a certain threshold (500 Mbp). This is because if the chromosomes are too small (e.g. for mitochondrial DNA or a highly fragmented genome assembly), the GPU utilization may suffer because of: (i) low number of seed hits between the chromosome pairs, and (ii) higher CPU-GPU memory overhead. This optimization requires an additional small dictionary of the chromosome start and stop positions in the concatenated sequences to be maintained in software to ensure that the segment files are generated with correct coordinates.

E. Multi-node implementation

Up to this section, we described the single node implementation of SegAlign with single and multiple GPUs. SegAlign has further been developed to compute WGA over multiple GPU nodes using the MapReduce technique.

Species (common) name	Assembly name	Size (Gbp)
Homo Sapiens (Human)	hg38	3.08
Mus musculus (Mouse)	mm10	2.72
Gallus gallus domesticus (Chicken)	galGal6	1.05
Taeniopygia guttata (Zebra Finch)	taeGut2	1.02

TABLE I: List of vertebrate species used in our study with their assembly name and size.

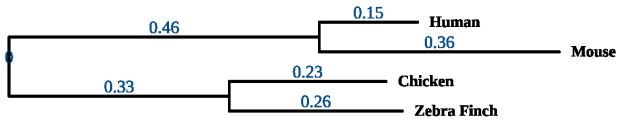


Fig. 7: Phylogenetic tree of the four vertebrate species with branch lengths in substitutions/site.

Task granularity is one of the most critical factors determining the performance of a distributed system. As described earlier, computing alignments for chromosome pairs are independent of each other, and this could be a possible strategy for distributing the parallel and independent tasks. However, as discussed earlier, because of the varying chromosome lengths, the number of HSPs and consequently, the number of extensions vary considerably between chromosome pairs. As a result, this strategy shows poor performance scaling due to high tail latency of the extension phase, e.g. the longest human-mouse chromosome pair takes 14 min while the median is only 1 min. Hence, SegAlign decouples seed-and-filter and extension stages into two different MapReduce phases. For the seed-and-filter phase, the tasks are distributed such that filtering a chromosome-pair constitutes a task. Each pair, in turn, generates a number of segment files, each with a fixed maximum limit on the number of HSPs (currently set to 10K). Subsequently, in the extension phase, a single task corresponds to gapped extension for HSPs in a segment file using LASTZ that takes fewer than a couple of minutes to complete, thus maintaining a low tail latency.

IV. Methodology

A. Species and genome data

We downloaded the latest genome assemblies of four vertebrate species (human, mouse, chicken, and zebra finch) from the UCSC database (http://hgdownload.cse.ucsc.edu/goldenpath/) for performing whole genome alignments. We pre-processed all assemblies to only include nuclear chromosomes, and removed the mitochondrial DNA, unplaced, unlocalized and alternate haplotype sequences from these assemblies. Table I provides the names of the four vertebrate genome assemblies used in our study, along with their sizes after pre-processing. We used the genome assemblies to perform three different whole genome alignments (human-mouse, human-chicken, chicken-zebra finch), which allowed evaluation for different combinations of genome sizes and evolutionary distance. Figure 7 provides the phylogenetic tree for the four species as used in the UCSC 100-way MULTIZ alignment (http://hgdownload.cse.ucsc.edu/goldenpath/hg38/multiz100way/), with branch lengths scaled to average substitutions per genomic site (a measure of evolutionary distance) [31].

B. Software baseline

We used LASTZ [17] version 1.04.03 in its default setting as our software baseline for WGA. Since LASTZ does not have a multi-threaded mode, we parallelized LASTZ on a multi-core system by first dividing the target and query genomes into smaller chunks of 10Mbp size, with successive chunks overlapping by 10Kbp, and then performing LASTZ alignments for each pair of target-query chunks using the `parallel` utility in Linux. For performance evaluation, we used an AWS c5.24xlarge instance, consisting of 96 vCPUs and costing $4.08/hour, since it provided us with the best LASTZ performance among all the other AWS instances. We used the `perf` utility in Linux for profiling LASTZ, and used the `--nogapped` option when evaluating the seed-and-filter throughput of LASTZ.

C. SegAlign Single-node evaluation

We implemented SegAlign's GPU-accelerated seed-and-filter stage in CUDA (version 10.2) and the remaining software pipeline from reading the target and query sequences to writing HSPs in output files in C++ and compiled using `g++` (version 7.5.0) at `-O4` optimization level. We used a bash script to monitor the HSPs being generated by above, and to spawn off a new LASTZ process, if an idle CPU core is found. In order to perform the alignment extension for the HSPs, the segment file is provided as input using the `--segments` option in LASTZ. This ensured that seeding, filtering and extension stages were pipelined while keeping all cores busy. We evaluated single-node speedup and costs for computing the whole genome alignments in comparison to LASTZ using 3 AWS GPU instances: p3.2xlarge (8 vCPUs, 1 GPUs, $3.06/hr), p3.8xlarge (32 vCPUs, 4 GPUs, $12.24/hr), and p3.16xlarge (64 vCPUs, 8 GPUs, $24.48/hr), with varying number of NVIDIA V100 GPUs and logical cores. The performance across different species pairs was evaluated on the p3.16xlarge instance. We also compared the output of human-mouse alignments produced by LASTZ and SegAlign using the R dotplot functionality as described in the LASTZ documentation (http://www.bx.psu.edu/~rsharris/lastz/README.lastz-1.04.03.html).

D. Multi-node evaluation

SegAlign's seed-and-filter and extension phases were implemented on a cluster of multiple GPU nodes using Apache Spark (version 2.4.4). While we also plan to extend SegAlign's support for workflow management systems like Cromwell [32], we have found that the Spark framework is extremely efficient and convenient for our use case. We implemented two Python scripts, one for the seed-and-filter phase and one for the extension phase. For the seed-and-filter phase, the script divides the input target and query

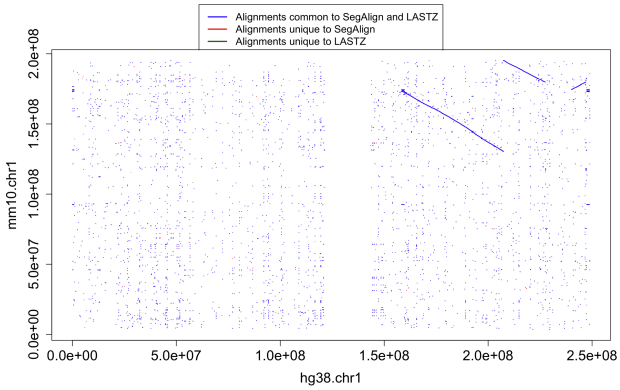


Fig. 8: Dot plot comparing SegAlign and LASTZ alignments between the chromosome 1 of the human and mouse genomes. SegAlign recovers every LASTZ alignment (i.e. no green alignments are found).

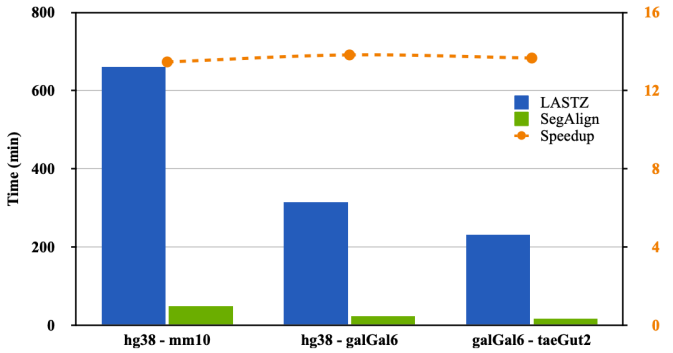


Fig. 9: LASTZ and SegAlign runtime comparison for different vertebrate WGA. Speedup resulting from SegAlign is shown in orange.

genomes into their respective chromosomes and then generates a bath of jobs which are specified using Spark's Resilient Distributed Datasets (RDDs). Each job consists of a target-query chromosome pair to be filtered and the corresponding segment files to be uploaded to a bucket in AWS Simple Storage Service (S3). The jobs are then distributed to the slave nodes using a mapper function and the output segment file paths are collected by the master node. Once the seed-and-filter phase completes, another script generates a set of jobs in a similar manner, where each job specifies a segment file to be extended using LASTZ.

We used AWS Elastic MapReduce (EMR) [21] to create a GPU cluster and evaluate our scaling results. The cluster used an m5.xlarge instance (4 vCPUs, $0.096/hr) as the master node and p3.2xlarge instances for the slave nodes. We used the YARN cluster manager which was configured with one executor core, dynamic allocation disabled and 13GB driver memory. For the seed-and-filter phase, the number of executors was set to `N` for an `N`-node cluster along with 40GB executor memory. The above settings ensured that a single SegAlign process executed on, and owned the GPU device on a slave node at a time. Similarly, for the extend phase, where the tasks need to be distributed one per available vCPU, the number of executors was set to 8`N` for an `N`-node cluster along with 5GB executor memory.

For SegAlign's weak scaling analysis, we aligned the human genome with artificial genomes containing multiple copies of mouse chromosome 1, in which for an `N`-node cluster, the mouse chromosome 1 was replicated in the query genome `N`-times.

V. Results

A. SegAlign WGA Comparison to LASTZ

The dot plot in Figure 8 presents a qualitative view of the alignments generated by SegAlign and how it compares to the alignments generated by LASTZ. The alignments common to SegAlign and LASTZ are represented by blue dots in the figure. Unique alignments generated only by SegAlign are shown as red dots and the ones unique to LASTZ in green. The plot did not have any green dots, implying that SegAlign finds all the alignments that LASTZ does.

SegAlign provides more alignments than LASTZ because it does not incorporate LASTZ's diagonal hashing strategy [17]. This strategy requires seed hits to be processed sequentially, which would be detrimental to the performance of the SegAlign's parallel seed hit extension on GPU. While we are considering incorporating a parallel hashing strategy in the future, the dot plot in Figure 8 shows that the concordance between the two alignments is already very high.

B. Single-node Analysis

The plot in Figure 9 shows the runtime of LASTZ and SegAlign for the 3 species pairs described in Section IV-A. As the product of the genome lengths of the species pair decreases, the number of seed hits decreases, causing an overall reduction in the WGA runtime for both LASTZ and SegAlign. SegAlign reduces the WGA computation time from the order of hours to minutes. Figure 9 also shows that SegAlign's speedup as compared to LASTZ remains within the close range of 13.5× to 14× even as the evolutionary distance between the species varies widely.

An analysis of SegAlign's filtering stage showed that on a single NVIDIA V100 GPU, it provides a throughput of up to 305 million seed hits/sec, which is around 180× the throughput provided by LASTZ on a single core of a modern Intel Xeon processor. Figure 10a shows the speedup and cost improvement for seed-and-filter stage in human-mouse WGA across different AWS instances. A similar speedup and cost-comparison are done for the end-to-end pipeline of SegAlign and the results are shown in Figure 10b. In both cases, the speedup increases almost linearly with the number of GPUs in the instances mentioned in Section IV-C. Efficient load-balancing using TBB and techniques described in Section III-D results in high CPU and GPU utilization over the complete pipeline. As described in Section III-D, as the number of GPUs increases, their utilization tapers during the

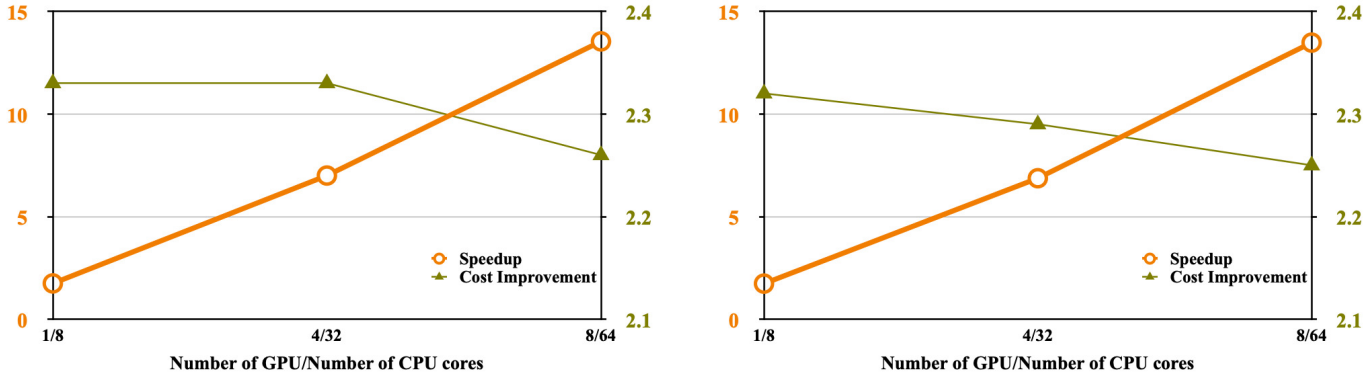


Fig. 10: Speedup and cost analysis for human-mouse WGA on different AWS instances for (a) SegAlign's seed-and-filter stage, and (b) complete SegAlign pipeline.

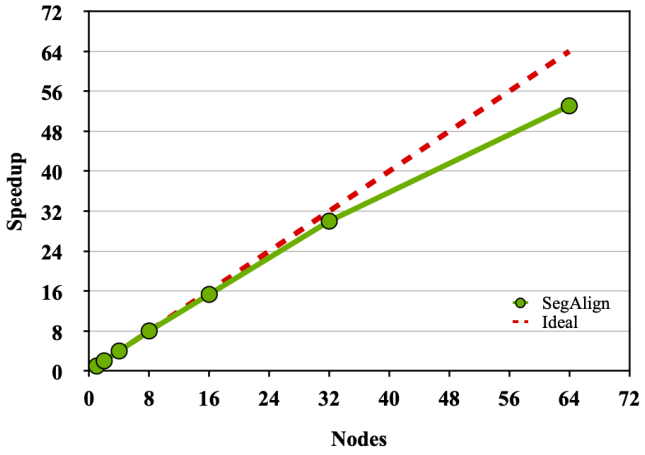


Fig. 11: Strong scaling analysis for SegAlign's multi-node implementation for human-mouse WGA.

tail-end of the execution of the last few query intervals for a particular target chromosome. The cost of the AWS instances are proportional to the number of GPUs and since the scaling is not perfect with the number of GPUs, the instance with a single GPU is found to be most cost-efficient for the end-to-end pipeline. Overall, the cost improvements lie within a close range of 2.25-2.33× of LASTZ for both, seed-and-filter stage, and the end-to-end pipeline. We verified with the p3dn.24xlarge (96 vCPUs, 8 GPUs, $31.212/hr) instance that increasing the number of CPU cores with 8 GPUs does not have an effect on the speedup. This is because the pipeline is not CPU-bound with the ratio of 8 CPU cores per GPU.

C. Multi-node Scaling Analysis

Figure 11 shows strong scaling analysis for SegAlign. SegAlign achieves a strong scaling efficiency of 93% with 32 nodes, which drastically reduces to 82% at 64 nodes. The efficiency starts tapering off at 64 nodes mostly due to a parallel slack – the entire WGA completed in 13 min, of which the last 2 min were spent on the most trailing task. For this given dataset, maximum possible speedup over the single node cluster is bound to 170×, which is based on the maximum runtime of a single task in seed-and-filter and extension phases.

Table II shows the weak scaling analysis for SegAlign. The weak scaling efficiency is 97.8% at 32 and lowers to 96% at 64 nodes. The compute in both phases, seed-and-filter and extension, scale near perfectly but the inefficiencies arise mostly due to increased communication from the slave nodes to the master node. We still achieve a high scaling efficiency since the compute to communication ratio is very high in SegAlign. For example, for the given dataset, communication latency and scheduler delays were found to vary between 1% - 1.5% of the total runtime and it increased with an increase in the number of nodes.

VI. Related Work

Alignment algorithms for finding local regions of similarity between the genomes of different species have been extensively studied in the past. The Smith-Waterman algorithm [24] provides an optimal approach for finding local alignments, given a scoring matrix and gap penalties, but is computationally intractable for most practical problems. Many heuristic approaches have been developed as a result. BLAST [33] was the first algorithm to incorporate and popularize the seed-and-extend paradigm for protein-sequence alignment and remains among the most cited computational tools in history. Later versions of BLAST [34] modified the original heuristic to seed-filter-extend, with an ungapped filtering stage between the seeding and gapped extension stages, that further improved its performance, particularly for long DNA sequences. However, BLAST is rarely used for aligning entire genomes of large sizes due to its poor performance and large memory requirement [35]. Whole genome aligners, such as MUMmer [36], AVID [37], BLASTZ [16] and LASTZ [17] addressed these issues, and are also largely based on the

seed-filter-extend heuristic. LASTZ is a standard tool for whole genome alignments, and has also been incorporated in a number of multiple genome alignment tools, including Cactus [38] and MULTIZ [39].

Hardware accelerators, including GPUs and FPGAs, have also been studied in the context of cross-species homology search and sequence alignment. A 2013 survey [40] reviews many of these accelerators. Many CUDA implementations for accelerating the BLAST algorithm exist. GPU-BLAST [41] and cuBLASTP [29] accelerate BLAST protein sequence alignment, while G-BLASTN [42] accelerates the BLAST nucleotide sequence alignment using GPUs. In particular, cuBLASTP implements ungapped extension in a manner similar to SegAlign, however, SegAlign differs from cuBLASTP in two significant ways: (i) SegAlign works with nucleotide sequences and not protein sequences like cuBLASTP, and (ii) SegAlign is a system designed to accelerate LASTZ, which, as mentioned earlier, is a superior tool for cross-species whole genome alignments compared to BLAST. Beyond BLAST, many prior CUDA implementations have looked at accelerating dynamic programming based sequence alignment algorithms. CUDAlign [43] can perform Smith-Waterman score matrix computation on multi-GPU platforms and has been evaluated for cross-species alignments. However, CUDAlign has many limitations including (i) it only computes alignments scores, and not the complete alignment path, and (ii) it can only return highest-scoring alignment cells, which cannot be used for discovering millions of homologous alignments possible in whole genomes. LOGAN [44] is another recent tool that accelerated the X-drop gapped alignment extension on GPUs and has been evaluated for long read alignment. While gapped alignment is not the dominant stage in LASTZ and other whole genome aligners, it starts dominating once the seeding and filtering stages are accelerated on GPUs, as is the case with SegAlign. LOGAN could be used to further accelerate SegAlign but this would require substantial changes to LOGAN's code, such as being able to support arbitrary scoring matrices. This has been left for future work.

FPGA accelerators for BLAST include Mercury-BLASTP [45], TimeLogic [46] and RC-BLAST [47]. An FPGA accelerator for whole genome alignment has been studied in Darwin-WGA [48]. However, Darwin-WGA implemented on FPGA is slower in comparison to LASTZ, since it modifies the LASTZ algorithm for higher alignment sensitivity. Another GPU tool, called MaxSubGenomeAlign [49], provides a slower implementation than LASTZ but with higher sensitivity. SegAlign is not aimed to address the sensitivity issues of LASTZ but to serve as its high-performance, drop-in replacement in current bioinformatics pipelines, such as Cactus and MULTIZ. To our knowledge, SegAlign is the first hardware-accelerated tool that accelerates LASTZ on commodity hardware. FPGAs have also been extensively studied for accelerating dynamic programming based sequence alignments [50]–[52].

More recently, much work on hardware acceleration has focused on read assembly using short [53]–[56] and long reads [27], [57], [58]. Long read assembly is similar to WGA in that it also requires alignment of long sequences that could be dissimilar (due to high error-rates in long read sequencing). However, long read accelerators cannot be directly applied for WGA since cross-species alignments can span millions of bases, much longer than typical long reads, and incorporate insertions and deletions reflecting evolutionary events that span hundreds of bases, which are almost never encountered in long reads.

Genome size (Mbp)	Number of nodes	Time
195	1	44m 25s
390	2	44m 27s
780	4	44m 43s
1560	8	45m
3120	16	45m 20s
6240	32	45m 23s
12480	64	46m 5s

TABLE II: Weak scaling analysis for SegAlign.

VII. Conclusion and Future Work

Whole genome alignments help us to realize the vast potential of the large number of genomes being sequenced and assembled to better understand evolution. The deluge of species assemblies necessitates hardware acceleration. In this paper, we describe SegAlign as a scalable tool to generate WGAs using GPUs. SegAlign provides up to 14× speedup with around 2.3× cost improvement over the state-of-the-art software whole genome aligner, LASTZ, and can serve as a drop-in replacement for many use cases. SegAlign also scales on a cluster of multiple GPU nodes with high efficiency.

SegAlign has been integrated into the Cactus multiple genome alignment tool [15] in its latest release (https://github.com/ComparativeGenomicsToolkit/cactus/releases/tag/v1.2.0). In the near future, we plan to generate thousand-way vertebrate genome alignments using the SegAlign-integrated version of Cactus. The order of magnitude acceleration provided by SegAlign as shown in the paper should reduce the time to generate such large multiple genome alignments from several months to days, and thereby enable the computational capacity to keep up with the sequencing rate in the coming years.

VIII. Code availability

SegAlign code is released under the MIT license and is available at https://github.com/gsneha26/SegAlign.

IX. Acknowledgement

We thank the anonymous reviewers for their helpful feedback. We thank Glenn Hickey for reviewing the SegAlign code and integrating it into the Cactus tool. We thank Hiram Clawson and Robert (Bob) Harris for helpful discussions. This work was supported by the Stanford SystemX Alliance, DARPA DSSoC, NVIDIA funding for YT, National Human Genome Research Institute (NHGRI) award no. R01HG010485 to BP and AWS research credits.

REFERENCES

[1] J. Armstrong, I. T. Fiddes, M. Diekhans, and B. Paten, "Whole-genome alignment and comparative annotation," *Annual review of animal biosciences*, vol. 7, pp. 41–64, 2019.

[2] O. Couronne, A. Poliakov, N. Bray, T. Ishkhanov, D. Ryaboy, E. Rubin, L. Pachter, and I. Dubchak, "Strategies and tools for whole-genome alignments," *Genome research*, vol. 13, no. 1, pp. 73–80, 2003.

[3] K. Lindblad-Toh, M. Garber, O. Zuk, M. F. Lin, B. J. Parker, S. Washietl, P. Kheradpour, J. Ernst, G. Jordan, E. Mauceli, *et al.*, "A high-resolution map of human evolutionary constraint using 29 mammals," *Nature*, vol. 478, no. 7370, pp. 476–482, 2011.

[4] M. Alexandersson, S. Cawley, and L. Pachter, "Slam: cross-species gene finding and alignment with a generalized pair hidden markov model," *Genome Research*, vol. 13, no. 3, pp. 496–502, 2003.

[5] A. Siepel, G. Bejerano, J. S. Pedersen, A. S. Hinrichs, M. Hou, K. Rosenbloom, H. Clawson, J. Spieth, L. W. Hillier, S. Richards, *et al.*, "Evolutionarily conserved elements in vertebrate, insect, worm, and yeast genomes," *Genome research*, vol. 15, no. 8, pp. 1034–1050, 2005.

[6] J. K. Schull, Y. Turakhia, W. J. Dally, and G. Bejerano, "Champagne: Whole-genome phylogenomic character matrix method places myomorpha basal in rodentia," *bioRxiv*, p. 803957, 2019.

[7] J. W. Sahl, M. N. Matalka, and D. A. Rasko, "Phylomark, a tool to identify conserved phylogenetic markers from whole-genome alignments," *Appl. Environ. Microbiol.*, vol. 78, no. 14, pp. 4884–4892, 2012.

[8] J. Ma, L. Zhang, B. B. Suh, B. J. Raney, R. C. Burhans, W. J. Kent, M. Blanchette, D. Haussler, and W. Miller, "Reconstructing contiguous regions of an ancestral genome," *Genome research*, vol. 16, no. 12, pp. 1557–1565, 2006.

[9] R. E. Green, E. L. Braun, J. Armstrong, D. Earl, N. Nguyen, G. Hickey, M. W. Vandewege, J. A. S. John, S. Capella-Gutiérrez, T. A. Castoe, *et al.*, "Three crocodilian genomes reveal ancestral patterns of evolution among archosaurs," *Science*, vol. 346, no. 6215, p. 1254449, 2014.

[10] NCBI, "Ncbi genome database." http://www.ncbi.nlm.nih.gov/genome/, 2018.

[11] G. K. C. of Scientists, "Genome 10k: a proposal to obtain whole-genome sequence for 10 000 vertebrate species," *Journal of Heredity*, vol. 100, no. 6, pp. 659–674, 2009.

[12] E. C. Teeling, S. C. Vernes, L. M. Dávalos, D. A. Ray, M. T. P. Gilbert, E. Myers, B. Consortium, *et al.*, "Bat biology, genomes, and the bat1k project: to generate chromosome-level genomes for all living bat species," 2018.

[13] H. A. Lewin, G. E. Robinson, W. J. Kress, W. J. Baker, J. Coddington, K. A. Crandall, R. Durbin, S. V. Edwards, F. Forest, M. T. P. Gilbert, *et al.*, "Earth biogenome project: Sequencing life for the future of life," *Proceedings of the National Academy of Sciences*, vol. 115, no. 17, pp. 4325–4333, 2018.

[14] D. Karolchik, R. Baertsch, M. Diekhans, T. S. Furey, A. Hinrichs, Y. Lu, K. M. Roskin, M. Schwartz, C. W. Sugnet, D. J. Thomas, *et al.*, "The ucsc genome browser database," *Nucleic acids research*, vol. 31, no. 1, pp. 51–54, 2003.

[15] J. Armstrong, G. Hickey, M. Diekhans, A. Deran, Q. Fang, D. Xie, S. Feng, J. Stiller, D. Genereux, J. Johnson, *et al.*, "Progressive alignment with cactus: a multiple-genome aligner for the thousand-genome era," *bioRxiv*, p. 730531, 2019.

[16] S. Schwartz, W. J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, and W. Miller, "Human–mouse alignments with blastz," *Genome research*, vol. 13, no. 1, pp. 103–107, 2003.

[17] R. S. Harris, *Improved pairwise alignment of genomic DNA*. PhD Thesis, The Pennsylvania State University, 2007.

[18] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE micro*, vol. 28, no. 2, pp. 39–55, 2008.

[19] Intel, "Intel® Threading Building Blocks." https://software.intel.com/en-us/tbb, 2020.

[20] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[21] Amazon, "Amazon Elastic MapReduce." https://aws.amazon.com/emr/, 2020.

[22] S. L. Clarke, J. E. VanderMeer, A. M. Wenger, B. T. Schaar, N. Ahituv, and G. Bejerano, "Human developmental enhancers conserved between deuterostomes and protostomes," *PLoS genetics*, vol. 8, no. 8, p. e1002852, 2012.

[23] M. Hiller, S. Agarwal, J. H. Notwell, R. Parikh, H. Guturu, A. M. Wenger, and G. Bejerano, "Computational methods to detect conserved non-genic elements in phylogenetically isolated genomes: application to zebrafish," *Nucleic acids research*, vol. 41, no. 15, pp. e151–e151, 2013.

[24] T. F. Smith and M. S. Waterman, "Comparison of biosequences," *Advances in applied mathematics*, vol. 2, no. 4, pp. 482–489, 1981.

[25] B. Ma, J. Tromp, and M. Li, "Patternhunter: faster and more sensitive homology search," *Bioinformatics*, vol. 18, no. 3, pp. 440–445, 2002.

[26] NVIDIA, "NVIDIA® CUDA C++ Programming Guide." https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2019.

[27] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics coprocessor provides up to 15,000 x acceleration on long read assembly," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 199–213, ACM, 2018.

[28] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," in *GPU computing gems Jade edition*, pp. 359–371, Elsevier, 2012.

[29] J. Zhang, H. Wang, H. Lin, and W.-c. Feng, "cublastp: Fine-grained parallelization of protein sequence search on a gpu," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 251–260, IEEE, 2014.

[30] C. E. Shannon, "A mathematical theory of communication," *Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[31] A. Siepel and D. Haussler, "Phylogenetic estimation of context-dependent substitution rates by maximum likelihood," *Molecular biology and evolution*, vol. 21, no. 3, pp. 468–488, 2004.

[32] K. Voss, G. V. D. Auwera, and J. Gentry, "Full-stack genomics pipelining with GATK4 + WDL + Cromwell [version 1; not peer reviewed]," 2017.

[33] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[34] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped blast and psi-blast: a new generation of protein database search programs," *Nucleic acids research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[35] M.-S. Kim, C.-H. Sun, J.-K. Kim, and G.-S. Yi, "Whole genome alignment with blast on grid environment," in *The Sixth IEEE International Conference on Computer and Information Technology (CIT'06)*, pp. 47–47, IEEE, 2006.

[36] A. L. Delcher, S. L. Salzberg, and A. M. Phillippy, "Using mummer to identify similar regions in large sequence sets," *Current protocols in bioinformatics*, no. 1, pp. 10–3, 2003.

[37] N. Bray, I. Dubchak, and L. Pachter, "Avid: A global alignment program," *Genome research*, vol. 13, no. 1, pp. 97–102, 2003.

[38] B. Paten, D. Earl, N. Nguyen, M. Diekhans, D. Zerbino, and D. Haussler, "Cactus: Algorithms for genome multiple sequence alignment," *Genome research*, 2011.

[39] M. Blanchette, W. J. Kent, C. Riemer, L. Elnitski, A. F. Smit, K. M. Roskin, R. Baertsch, K. Rosenbloom, H. Clawson, E. D. Green, *et al.*, "Aligning multiple genomic sequences with the threaded blockset aligner," *Genome research*, vol. 14, no. 4, pp. 708–715, 2004.

[40] S. Aluru and N. Jammula, "A review of hardware acceleration for computational genomics," *IEEE Design & Test*, vol. 31, no. 1, pp. 19–30, 2013.

[41] P. D. Vouzis and N. V. Sahinidis, "Gpu-blast: using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, vol. 27, no. 2, pp. 182–188, 2011.

[42] K. Zhao and X. Chu, "G-blastn: accelerating nucleotide alignment by graphics processors," *Bioinformatics*, vol. 30, no. 10, pp. 1384–1391, 2014.

[43] F. d. O. Edans, G. Miranda, A. C. de Melo, X. Martorell, and E. Ayguadé, "Cudalign 3.0: Parallel biological sequence comparison in large gpu clusters," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 160–169, IEEE, 2014.

[44] A. Zeni, G. Guidi, M. Ellis, N. Ding, M. D. Santambrogio, S. Hofmeyr, A. Buluç, L. Oliker, and K. Yelick, "Logan: High-performance gpu-based x-drop long-read alignment," *arXiv preprint arXiv:2002.05200*, 2020.

[45] A. Jacob, J. Lancaster, J. Buhler, B. Harris, and R. D. Chamberlain, "Mercury blastp: Accelerating protein sequence alignment," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 1, no. 2, pp. 1–44, 2008.
[46] T. B. Solisions, "Timelogic decypher blast engine introduction," 2010.
[47] K. Muriki, K. D. Underwood, and R. Sass, "Rc-blast: Towards a portable, cost-effective open source hardware implementation," in *19th IEEE International Parallel and Distributed Processing Symposium*, pp. 8–pp, IEEE, 2005.
[48] Y. Turakhia, S. D. Goenka, G. Bejerano, and W. J. Dally, "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)*, pp. 359–372, IEEE, 2019.
[49] A. Aljouie, L. Zhong, and U. Roshan, "High scoring segment selection for pairwise whole genome sequence alignment with the maximum scoring subsequence and gpus," *International Journal of Computational Biology and Drug Design*, vol. 13, no. 1, pp. 71–81, 2020.
[50] D. T. Hoang and D. P. Lopresti, "Fpga implementation of systolic sequence alignment," in *International Workshop on Field Programmable Logic and Applications*, pp. 183–191, Springer, 1992.
[51] Z. Nawaz, M. Nadeem, H. van Someren, and K. Bertels, "A parallel fpga design of the smith-waterman traceback," in *2010 International Conference on Field-Programmable Technology*, pp. 454–459, IEEE, 2010.
[52] K. Benkrid, Y. Liu, and A. Benkrid, "A highly parameterized and efficient fpga-based skeleton for pairwise biological sequence alignment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 4, pp. 561–570, 2009.
[53] L. Wu, D. Bruns-Smith, F. A. Nothaft, Q. Huang, S. Karandikar, J. Le, A. Lin, H. Mao, B. Sweeney, K. Asanović, *et al.*, "Fpga accelerated indel realignment in the cloud," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 277–290, IEEE, 2019.
[54] K. Koliogeorgi, N. Voss, S. Fytraki, S. Xydis, G. Gaydadjiev, and D. Soudris, "Dataflow acceleration of smith-waterman with traceback for high throughput next generation sequencing," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 74–80, IEEE, 2019.
[55] Edico Genome, "Dragen bio-it platform.."
[56] E. B. Fernandez, J. Villarreal, S. Lonardi, and W. A. Najjar, "Fhast: Fpga-based acceleration of bowtie in hardware," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 12, no. 5, pp. 973–981, 2015.
[57] L. Guo, J. Lau, Z. Ruan, P. Wei, and J. Cong, "Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 127–135, IEEE, 2019.
[58] P. Chen, C. Wang, X. Li, and X. Zhou, "Accelerating the next generation long read mapping with the fpga-based system," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 11, no. 5, pp. 840–852, 2014.