2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

GenAx: A Genome Sequencing Accelerator

Daichi Fujiki* Arun Subramaniyan* Tianjun Zhang* Yu Zeng
Reetuparna Das David Blaauw Satish Narayanasamy
University of Michigan - Ann Arbor
{*dfujiki, arunsub, tianjunz, yuzeng, reetudas, blaauw, nsatish*}*@umich.edu*

***Abstract*—Genomics can transform health-care through precision medicine. Plummeting sequencing costs would soon make genome testing affordable to the masses. Compute efficiency, however, has to improve by orders of magnitude to sequence and analyze the raw genome data. Sequencing software used today can take several hundreds to thousands of CPU hours to align reads to a reference sequence.**

This paper presents GenAx, an accelerator for read alignment, a time-consuming step in genome sequencing. It consists of a seeding and seed-extension accelerator. The latter is based on an innovative automata design that was designed from the ground-up to enable hardware acceleration. Unlike conventional Levenshtein automata, it is string independent and scales quadratically with edit distance, instead of string length. It supports critical features commonly used in sequencing such as affine gap scoring and traceback.

GenAx provides a throughput of 4,058K reads/s for Illumina 101 `bp` reads. GenAx achieves 31.7× speedup over the standard BWA-MEM sequence aligner running on a 56-thread dual-socket 14-core Xeon E5 server processor, while reducing power consumption by 12× and area by 5.6×.

***Keywords*-Automaton, Sequence alignment, Accelerator**

I. Introduction

Whole genome sequencing (WGS) determines the complete DNA sequence of an organism's genome. While it cost nearly $3 billion to sequence the first human genome in 2001 [1], just over the last one decade, the production cost of sequencing has plummeted from ten million dollars to thousand dollars, and is soon expected to go below a hundred dollars per genome [2]. This remarkable growth in genomics has far outpaced Moore's Law and has the potential to transform personalized medicine. By understanding mutations in the cancer cell of a particular patient, it is possible to devise individualized treatment plans [3]. By analyzing large volumes of genome data of diverse populations, we can better understand the causes of various diseases ranging from cancer [4], Alzheimer's [5], to rare genetic disorders [6], assess risk factors and develop better cure. Several governments around the world have now launched projects aiming to bring genome testing to clinical practice [7], and it is likely to become a standard practice of care over the next decade, when genome analysis may become as common as blood-tests.

To realize the full potential of genomics, however, computing system efficiency needs to improve by orders of magnitude. Data generated from sequencing just a million genomes would produce over 300 Petabytes of data [8] (larger than Facebook data [9]). One Illumina's HiSeq Ten high-throughput sequencing machine alone can now sequence 45 genomes per day, producing nearly 2 TB of data in a week [10]. Oxford Nanopore has even started producing hand-held portable devices for sequencing organisms in the wild [11].

As Moore's Law tapers off, we envision hardware acceleration for genomics applications would soon become essential. Cloud service providers such as Amazon are now starting to make FPGA (F1) instances available on their cloud, which would accelerate the adoption of hardware accelerators for applications such as genome sequencing. Reduced form factor due to hardware customization can also allow raw data to be sequenced within portable sequencers to produce a smaller processed output (*variants*).

Sequencing each genome (referred to as secondary analysis) could take hundreds to thousands of CPU hours depending on the *read length* [12]. A genome is essentially a long string (3.08 `Giga bp` for a human genome) of DNA base-pairs (`bp`) `A, G, C`, and `T`. A sequencing machine splits a DNA into billions of small *reads*. In reference-guided assembly, these reads are aligned by matching them to a previously sequenced genome. This task is complicated by the fact that the new individual's genome may not exactly match that of the reference genome. In fact, the end goal is to determine the variants in the new genome. Furthermore, the sequencing machine can introduce error into the reads as well. Illumina's short reads have about 2% error. To partly address this problem, the sequencing machine produces several reads (30× - 50×) to cover every position in the genome.

While there are several computational steps in sequencing raw genome data, we focus on accelerating *read alignment*, a time-consuming step in secondary analysis. Most commercial software today is based on the Broad Institute's BWA-MEM software [12] for read alignment, whose output is treated in practice as a standard. One of our design goals is to not introduce heuristics in the accelerator, so that software built on top of it can meet these standards.

Read alignment determines the position of a read in the genome. Due to variants and sequencing errors, a read (referred as query `Q`) may not perfectly match a substring in the reference genome `R`. Sequence aligners solve this problem in two steps: *seeding* and *seed-extension*. Seeding finds perfect matches in the reference genome for small substrings (*seeds*) in a read. The seed positions are then *extended* to determine the best position by using an approximate string matching algorithm based on computing Levenshtein (edit) distance.

*Mr Fujiki, Mr Subramaniyan, and Mr Zhang contributed equally to the paper; their names are placed alphabetically in the author list.

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00017

In this paper, we present GenAx, an ASIC custom hardware accelerator for both steps in read alignment. Seeding involves many irregular accesses to a reference index structure. We address this problem by segmenting the genome to produce a smaller cache-able index for each segment, and seeding for each segment separately. While this may produce more seeds to extend than the baseline, the seeds are extended at a high-throughput in GenAx using our accelerator for approximate-string matching, which we introduce next.

Approximate string matching has been widely studied. The most widely used algorithm, particularly in genomics, is a dynamic programming algorithm called Smith-Waterman [13]. It computes the edit distance between two strings by filling a grid of size N^2, where N is the string length. While there have been several optimizations [14], [15], including hardware accelerators [16], [17], this quadratic algorithm fundamentally does not scale for long strings. While widely used Illumina machine's reads are short (100 `bp`), new generation machines from PacBio and Oxford Nanopore are starting to support longer reads.

While Levenshtein Automata (LA) based solutions have been known, they are rarely used in sequencing software as they fail to outperform Smith-Waterman [18]. Recent work [19] has investigated using Micron's Automata Processor (AP) or a Cache Automaton [20] for accelerating LA based solutions. However, an LA too has $\mathcal{O}(K*N)$ states, proportional to string length. A more important issue is that an LA is specific to a given string. Therefore, a hardware accelerator needs to context-switch the automata states after every read. Given that there are billions of reads, these context-switches can become prohibitive. Furthermore, none of these automata accelerators support critical features necessary for sequencing: finding the best string match using an affine gap function [21] instead of edit distance as the scoring metric, and traceback.

We design a new automata, String Independent Local Levenshtein Automata (*Silla*), from the ground-up to enable an efficient and scalable hardware accelerator for approximate string matching. Silla uses a state to represent the number and types of edits, instead of tracking the number of matched positions as done in LA, and operates them using a primitive called *retro comparison*. As we have three types of edits to support (insertion, deletion, and substitution), this would require a 3D state machine, where each dimension's size is the same as the edit distance. We present a technique to collapse this to a 2D state machine.

Unlike LA, Silla is string independent, meaning it can compute the edit distance of any two strings. It has only $(K+1)^2$ states, where `K` is the edit distance. Since edit distances are typically much smaller than the string length, Silla scales well. Furthermore, all of its states communicate only with their local neighbors in a 2D plane, which makes its overall structure *regular* and *composable*.

By leveraging the above properties of Silla, we design an efficient hardware accelerator called SillaX. Our 28nm implementation of the SillaX edit machine requires only 13 gates per state and each processing element operates at 6 GHz. We further extend the edit machine to support a number of functionalities required for seed-extension in genome sequencing. It includes support for computing the best solution based on affine gap penalty [21] and clipping [12], instead of using edit distance as the metric for string comparison. Also, it enables traceback [13], which allows us to gather the exact sequence of edits in the reference genome. Prior hardware accelerators based on Smith-Waterman either delegate traceback to software [17], [22], which then becomes a bottleneck, or require significant hardware space proportional to the read length [22], [23].

We synthesized SillaX in 28 *nm* technology and estimate its frequency to be 2 GHz while consuming 1.5 W power and 1.41 mm^2 area. SillaX provides 62.9× speedup over optimized banded Smith-Waterman running on a 56-thread dual-socket 14-core Xeon E5 server processor. SillaX was verified for GRCh38 human genome assembly with 787,265,109 Illumina 101 `bp` reads. GenAx achieves a 31.7× speedup over standard BWA-MEM aligner, while reducing power consumption by 12× and area by 5.6×. GenAx provides a read alignment throughput of 4,058K reads/s.

This paper makes the following contributions:

- We present Silla, a novel automata for computing the edit distance between two strings. Unlike Levenshtein Automata (LA), its state space is proportional to edit distance, and string independent. Its structure is regular and composable, where all states communicate locally.
- We present SillaX, an accelerator for approximate string matching based on Silla. We present a number of solutions to efficiently support affine gap functions and traceback features required in sequencing.
- We present GenAx, an accelerator for read alignment in genome sequencing. It is composed of SillaX, and a seeding accelerator. The seeding accelerator overcomes irregular high-bandwidth memory accesses to the reference index by segmenting the genome to generate cache-able indexes and seeding within each genome segment.

II. Seed Extension: Background and Motivation

Problem statement: Seeding (§V) finds a set of positions in the reference genome (`hits`) where a read could find a match. In the seed-extension step for a read, the reference strings at the hit positions are matched with the read. Matches are scored using an *affine gap function* [21], which is based on edit distance, but weighs different edit types differently. The hit position for a read that yields the highest score is chosen as that read's mapping position. The final output also contains a trace of edits to the reference string needed to align the read at the chosen reference position. This final step is referred to as *traceback*.

Dynamic Programming: The fundamental operation in seed extension is approximate string matching [24], [25]. The most widely used algorithm in sequencing soft-

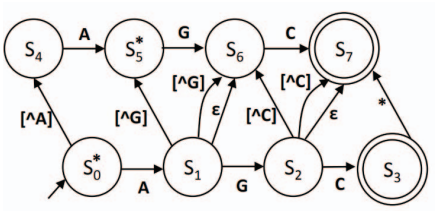

Figure 1: LA for `K = 1` and reference string `AGC`.

ware is a dynamic programming algorithm called Smith-Waterman [13]. It computes optimal local alignments between two sequences by comparing segments of all possible lengths. It operates in two phases. *Score-computation* builds the dynamic programming matrix (N^2) based on a general scoring scheme. Then *traceback* constructs the optimal alignment by tracing back pointers starting from the highest scoring cell. It fundamentally has $\mathcal{O}(N^2)$ time and space complexity. While there have been several optimizations [14], [15], and approximation heuristics [26] developed to reduce their time, it does not scale well as string length increases.

Several FPGA-based hardware accelerators have been proposed for the Smith-Waterman algorithm [16], [17]. These leverage wavefront parallelism in systolic arrays to accelerate the *score-computation* phase of the Smith-Waterman algorithm. However, they require up to $\mathcal{O}(N)$ processing elements and do not scale to long reads. There has also been work on banded implementations of the Smith-Waterman algorithm [27], where only cells within a $2K+1$ band around the principal diagonal of the Smith-Waterman matrix are computed. Most of these accelerators also either offload the *traceback* phase to software or have traceback support only for short string lengths for an additional $\mathcal{O}(N)$ space overhead [17], [22].

Automata-based: The Levenshtein Automata (LA) for approximate string matching accepts all strings that lie within `K` edit distance of its stored pattern. Figure 1 shows an example LA. Each state essentially represents the position in the reference string up to which a match has been found, and the number of edits seen so far. As a result, it has a total of $K*N$ states. Its time complexity is $\mathcal{O}(N^2)$, as in the worst case all of its states may be active. Sequencing software systems rarely use LA based implementations as they struggle to outperform Smith-Waterman.

In-memory [28], [20] and ASIC automata accelerators [29], [30] can be used to implement LA. However, LA is poorly suited for hardware acceleration due to several reasons. One, since it is string dependent, the hardware needs to be reprogrammed every time the string changes, which can be prohibitive especially for seed extension in sequencing. It requires processing billions of different reads, where each read needs to be compared to several seeds in the reference. Two, its space requirement is proportional to string length. When read lengths increase to millions of base-pairs, LA based hardware solutions would be impractical. Third, none of the existing hardware automata accelerators support unique features required in sequence aligners: scoring, clipping, and traceback. It is challenging to include these features. For example, adding logic to compute gap affine scores for state transitions in Micron's Automata Processor (AP) is likely to be expensive.

A recent advancement in automata theory called Universal Levenshtein Automata (ULA) addressed some of the limitations of LA [31]. While ULA is string independent, it does not efficiently map to a hardware accelerator as communication between states are not local. Also, each state has a high-degree of fan-out ($\mathcal{O}(K)$), as every state in ULA is connected to a state in every higher level of edit distance to support deletions. To date, there is no hardware realization of the Universal Levenshtein Automata (ULA), nor has it been used in sequencing software.

III. Silla Algorithm

We present a non-deterministic finite-state automata for approximate string matching called String Independent Local Levenshtein Automata (Silla). Silla is designed from the ground-up to enable efficient hardware acceleration. Unlike Smith-Waterman implementations [13], Silla's space complexity is quadratic in edit distance, not quadratic in string length, and its hardware implementation's (§ IV) time complexity is $\mathcal{O}(N)$. Thus, it is particularly attractive for matching long strings with limited edit distance.

Unlike Levenshtein Automata, Silla is *string independent*, as one automata can process any pair of strings. It is *local*, because communication exists only between adjacent states which are physically placed next to each other in silicon. This eliminates the need for long wires and enables scaling to very large automata without degrading performance. Also, the structure is regular and composable, allowing two smaller Sillas to be composed into a larger one. These properties are crucial to realize an efficient and general hardware accelerator for approximate string matching described in the next section (§IV).

Silla solves the following problem: Given two strings, a reference `R` and a query `Q`, compute the minimum Levenshtein (edit) distance between them if it is less than a small bound `K`.

A. Silla for Indel

We first describe the Silla design assuming only insertions and deletions (indels), and then extend it to support substitutions. A key observation is that we can use the states to represent the number and type of edits made so far, and not explicitly track the matches as it is done in Levenshtein automata. Figure 2(c) illustrates indel Silla for a maximum edit distance of two (`K = 2`), where a state `i,d` means that when that state is reached, the automata has seen `i` insertions and `d` deletions. All states have a match transition back to the state itself as shown for the start state (omitted for other states for clarity).

Computation begins at the start state (`0,0`) which represents no edits (`i=d=0`). At all active states, one character

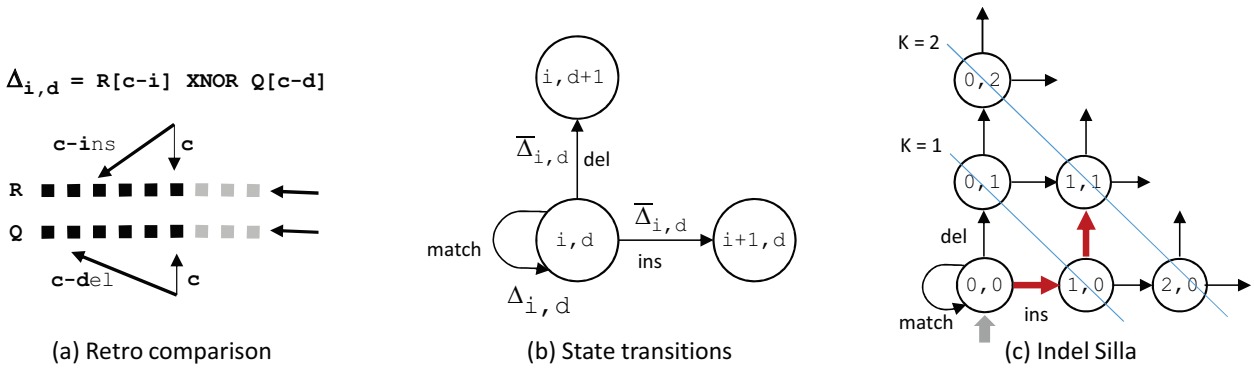

Figure 2: Retro comparisons (a) are used to determine state transitions (b) in Indel Silla (c).

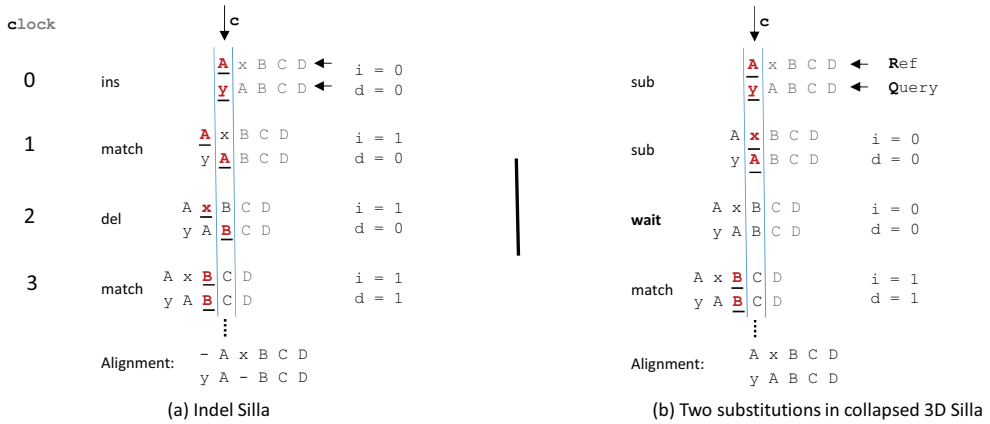

Figure 3: Silla Illustration.

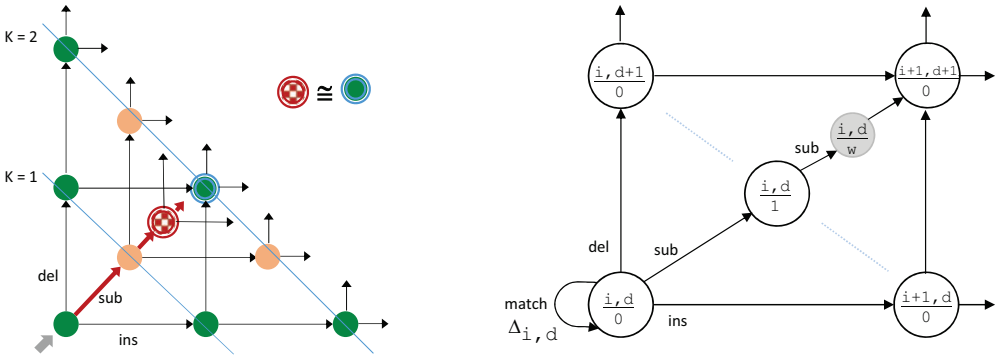

Figure 4: Collapsing 3D Silla. Supports Indels and substitutions.

from each of the two strings is compared in every cycle (step) starting from the first character. As long as there are no edits, the positions of the compared characters in the two strings `R` and `Q` are the cycle number `c` as shown in Figure 2(a) by the two vertical arrows labeled `c`. We refer to this position `c` as the *cycle* position.

On an insertion (or a deletion) the position in the reference (or query) needs to be offset with respect to the cycle position. This can be understood from the example in Figure 3(a). As the comparison fails in the first cycle (`A` $\neq$ `y`), Silla explores as one possibility that a character (`y`) is inserted into the query `Q` by transitioning into state `1,0` . In

the next cycle, state `1,0` should compare the previously unmatched character `A` in the reference `R` to the current character in the query `Q`. This is achieved by offsetting the character position in the reference `R` by as many insertions as the state represents (one in our example).

Similarly, the character position in the query `Q` is offset by the number of deletions. For example, when the comparison fails again in the third cycle (`x` $\neq$ `B`), Silla explores deleting `x` from `R` by transitioning to state `1,1` . The new state increments the character position offset for the query, so that the unmatched character `B` from the previous cycle is again compared, but this time to the following (now current) character `B` in the reference.

Silla then generates two more matches for characters `C` and `D` and thus discovers a solution for aligning the given two strings within an edit distance of two. The final alignment is shown at the bottom of the Figure 3(a).

Thus, the character positions whose comparison controls a state is determined by the indels that the state represents. We refer to these comparisons as *retro comparisons*, and the offsets as *indel offsets*. Figure 2(a) shows the equation and its illustration for computing the retro comparison for a state `i,d` . The state transitions based on a retro comparison is depicted in Figure 2(b). As you can notice, Silla explores both options, insertion and deletion, when a retro comparison fails for a state.

All the states in Silla are accepting states. After Silla completes processing of a pair of strings, the remaining active states represent possible string alignments with edit distance `i+d <= K` . The active state with the smallest indel indicates the minimum edit distance for the given strings. If no states remain active at the end of processing, there is no alignment with `indel < K` . The number of states in the indel Silla is $(K+1)^2/2$, as it is half a square with a side of length $(K+1)$.

B. 3D Silla for Substitutions

We now extend Silla to support substitutions. An easy solution for tracking substitutions is to add states in the third dimension to Silla. Each layer in the third dimension looks like a 2D indel Silla, and there are as many layers as the maximum possible number of substitutions (which is limited by `K`). When the retro comparison fails at a state `i,d|s` , to explore substitution, Silla transitions to a corresponding state in the next substitute layer along the third dimension (`i,d|s+1`).

Figure 4(a) depicts 3D Silla. A state's color represents the 3D layer it belongs to. As there are `K+1` layers, we have $(K+1)^3/2$ states.

C. Collapsed 3D Silla for Indels and Substitutions

3D Silla requires $\mathcal{O}(K^3)$ states. Furthermore, a hardware for 3D Silla would also be inefficient due to challenges in laying out a 3D design on a 2D plane. We avoid these problems by reducing a 3D Silla to an equivalent 2D Silla as follows.

Our key observation is that we need only one additional layer of 2D Silla, not K, to support substitutions, and that we can collapse the states needed in the higher substitution layers into one of those two layers.

Intuitively, the reason for having two dimensions in the 2D indel Silla is that we need to track the indel offsets in the two strings for different states of the automaton. However, a substitute action does not change the indel offsets and the function of the third dimension in the 3D Silla is simply to "count" or record the number of substitutions. Since we are only interested in the total edit distance, we notice that state `i,d|s` in the 3D Silla has the same edit distance as state `i+1,d+1|s-2` . Furthermore, the *relative* indel offsets of these two states is also the same `i-d` , although state `i+1,d+1|s-2` is shifted one character position earlier in the string than `i,d|s` . Hence, we can merge state `i,d|s` with state `i+1,d+1|s-2` by inserting one wait cycle in the path from `i,d|s-1` to `i+1,d+1|s-2` .

The example in Figure 3(b) illustrates this merger operation. It is for the same two strings discussed before, but this time we discuss a solution that uses two substitutions to align them instead of an insert followed by a delete. When retro comparison fails in cycle 0 (`A` $\neq$ `y`), control switches to the `0,0|1` state to explore a substitution. When the comparison fails again (`x` $\neq$ `A`), to explore another substitution, 3D Silla would transition to `0,0|2` . But, as noted above, the number of edits represented by the state `0,0|2` is same as `1,1|0` and the relative difference between their indel offsets is the same (zero). Hence, in cycle 2, a `0,0|2` state would be comparing the two characters `B` , which are the same characters as state `1,1|0` is comparing in cycle 3. In a way, state `0,0|2` is one cycle *ahead* of `1,1|0` .

Therefore, we can merge `0,0|2` with `1,1|0` simply by delaying the path from `0,0|1` to `1,1|0` on substitution by one cycle. Figure 3(b) illustrates this. When the retro comparison fails in cycle 1, Silla transitions to a wait state that takes no action in cycle 2. In the next cycle 3, the execution correctly resumes in state `1,1|0` .

To generalize, our final Silla design supports indels and substitutions using two layers of 2D Silla. Final state transitions are shown in Figure 4(b). Matching transitions are again omitted for clarity. To explore a substitution from a state `i,d|1` in the second layer, Silla transitions to a wait state `i,d|w` , and then in the following cycle, transitions back to merge with a state in the first layer `i+1,d+1|0` . Figure 4(a) can now be re-interpreted as a collapsed 3D Silla, where the checkered states represents the wait state and has a single outgoing transition to merge with states in the first layer.

Collapsed 3D Silla has $(K+1)^2/2$ regular states in each of the two layers, and also has an additional $(K+1)^2/2$ wait states resulting in a total $3(K+1)^2/2$ number of states. Also, by grouping states `i,d|0`, `i,d|1` and `i,d|w` together as one unit in the layout, a completely regular design is obtained with only local communication between neighboring units. Henceforth, we refer to this collapsed 3D Silla simply as Silla.

D. Merging Confluence Paths is Sound

Silla explores multiple solutions concurrently in different states. When a retro comparison fails, a state activates all three of its outgoing edges to pursue all possible edits to handle the mismatch. For example, Figure 3 shows two paths for the same string. However, in fact, Silla would explore many more paths than shown, and often there are more than one solution. In the example, the path with a deletion and an insertion and the path with two substitutions are both optimal solutions.

Silla merges a set of paths that reach a state in the *same* cycle into one active path. We refer to these paths as confluence paths. A state in Silla can have as many as four incoming edges (e.g., state `1,1|0` in Figure 4(a)), including the matching edge.

Fortunately, it turns out that merging confluence paths is safe without additional precautions. The reason is as follows. For a given Silla state, and given cycle, we can partition the reference (`R`) and query (`Q`) strings as `x||y` and `u||v`, where `x` and `u` are prefixes of `R` and `Q`, respectively. A nice property is that this partition is the same for all the confluence paths. The prefixes that have been processed in the previous cycles, will not be examined going forward, and it is guaranteed that all the confluence paths observed the same number of edits for them. All the confluence paths also share the same suffix, and the edit distance computation for the unprocessed suffixes (`y` and `v` in `R` and `Q` respectively), is *independent* of the path taken so far. Thus, it is not necessary to independently explore that same suffix for each of the confluence paths, and therefore they can be safely merged.

IV. Silla Accelerator for Genome Sequencing

This section presents the Silla hardware accelerator (SillaX) for genomics. It uses a form of a systolic array architecture that efficiently computes and locally distributes retro comparisons to all the states. Besides edit distance, it also supports more sophisticated scoring schemes based on the affine gap penalty [21] used in genomics. Finally, it adds capability to traceback the sequence of edits made to reach the final alignment solution. These capabilities are essential to perform seed-extension in genome sequence alignment.

We synthesized and validated the implementation for a whole human genome and confirmed that its output matches that of Broad Institute's BWA-MEM standard pipeline [12] for all 787,265,109 single-ended reads.

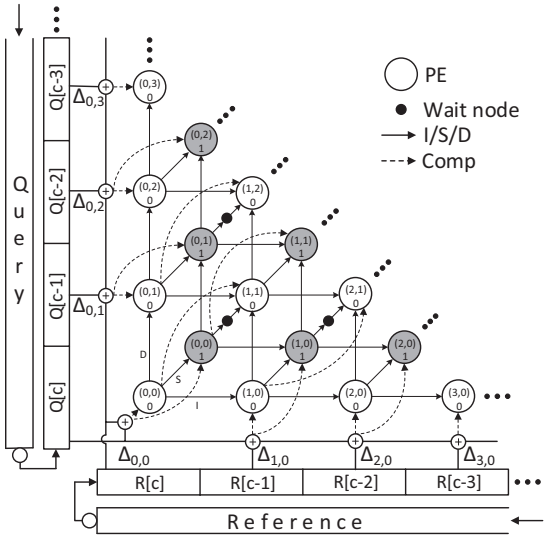

Figure 5: SillaX Accelerator.

A. SillaX Edit Machine

SillaX implements each regular state in Silla as a small processing element (PE) shown in Figure 6. PEs for wait states are not shown, but they simply activate the outgoing edge when they are active. An important property that Silla guarantees is that a state has to communicate only with its neighbors. This allows us to connect all the PEs using a locally communicating regular network (Figure 5). We refer to a PE simply as a state in our discussions.

Every regular state is controlled by the result of its retro comparison in each cycle. A significant challenge that we address is the efficient calculation and distribution of the retro comparisons to all the regular states. In a naive system, we would need as many retro comparisons as the number of regular states ($(K+1)^2/2$), every cycle. However, across two clock cycles, many of the the retro comparisons are reused. This allows us to solve this problem with just $2K+1$ comparisons per cycle as described next.

A retro comparison for a state is computed based on the current cycle (`c`) and that state's indel (`i,d`) as we discussed in Figure 2(a). *We observe that the states along a diagonal can reuse the retro comparisons*. A state `i,d` needs the same retro comparison that `i-1,d-1` needed a cycle earlier. Therefore, in each cycle, SillaX computes the retro comparisons for all the peripheral states, $\forall i$ `i,0` and $\forall d$ `0,d`, and then shifts them diagonally into the interior states every cycle. That is, a state `i,d` latches its incoming retro comparison and forwards it to `i+1, d+1` the next cycle (`Comp` in Figure 6).

To implement the above functionality, SillaX has two sets of shift registers, one for each dimension. Input characters from two strings `R` and `Q` flow through those shift registers as shown in Figure 5. A set of $2K+1$ comparators outside

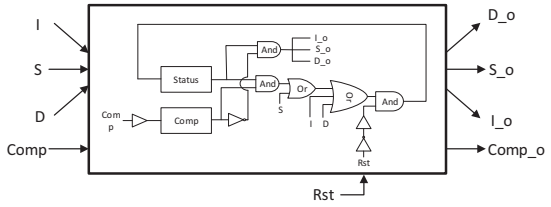

Figure 6: PE for SillaX Edit Machine.

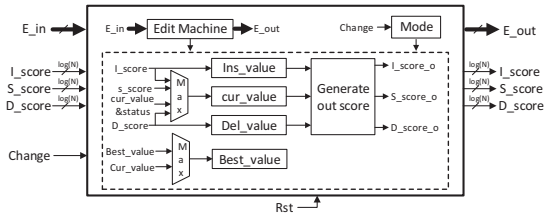

Figure 7: PE for SillaX Scoring Machine.

the grid compute the retro comparisons every cycle ($K+1$ comparisons for each dimension with one common comparison for `0,0`). Note that computing and distributing the retro comparisons again requires only local communication between neighboring states allowing for a highly scalable design. The only exception to this local communication is the distribution of the current cycle's characters in the reference and query strings `R[c] and Q[c]` which are distributed across the entire periphery. However, distribution of these two values can be accomplished using a distribution tree which delivers the values to all comparators at the same time, much like a clock tree distributes a synchronized clock to all the PEs.

Efficiency: SillaX requires only $\mathcal{O}(K^2)$ states (processing elements) and computes in about N cycles, where K is the edit distance and N is the string length. Typically, $K << N$. Our implementation in 28nm can be clocked at 6 GHz, and each PE has only 13 gates. This is significantly more efficient than software implementations, whose time complexity is $\mathcal{O}(N^2)$. Hardware accelerators for these require $\mathcal{O}(N)$ processing elements, which does not scale as well for large N.

B. Scoring Machine

Edit distance is a simple form of scoring alignment between two strings which can have many different uses. However, read alignment in genome sequencing uses a more sophisticated scoring scheme based on empirical evidence gathered from analyzing many genomes [21].

If we use a constant score for each type of edit, then it remains safe to merge *confluence paths* by selecting the one with the highest score, as the properties discussed in § III-D continue to hold.

The scoring scheme used in the standard BWA-MEM pipeline, however, raises a new problem. It rewards every match (`+1`) and penalizes every substitution (`spenalty = -4`) with predefined scores. Each indel, which represents a set of consecutive deletions or insertions, is penalized using

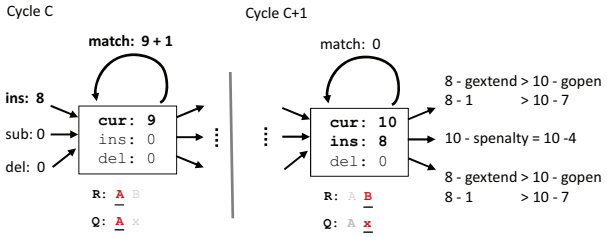

Figure 8: Delayed merging is needed to support affine gap penalty.

affine gap penalty (`G`):

$$G = gopen + gextend * id,$$

where `id` is the number of characters deleted or inserted, `gextend` (`-1`) is the penalty for each consecutive edit, and `gopen` (`-6`) is an additional one-time penalty for each indel.

Given this, paths that have opened an indel gap (`open-path`) have an advantage over a path where the latest retro-compare is a match or substitution (`closed-path`). An open-path need not pay a gap opening penalty for the next insertion or deletion, but a closed-path should. As a result, we cannot merge the confluence paths into one at a given state and cycle based on their current scores alone since the future score depends on whether the confluence path is open or closed. Fortunately, we can address this by delaying the merge to the following cycle as shown in Figure 8.

To enable delayed merge, we latch the scores of the incoming active insertion and deletion paths in a state as shown in Figure 7. If there is a mismatch in the next cycle, we can select the best outgoing indel path by adding the gap penalty to previously closed paths. If there is a match in the next cycle, we can select the active path for the state by choosing between the closed paths and the open-paths from the previous cycle, which are now closed due to the match, based on which path has the highest score. We refer to this technique as *delayed merging*.

Closing an open-path due to a match in a state would prevent that open-path from potentially exploring a better solution in the higher edit states without having to re-open the path. For this reason, in the scoring machine, an active state conservatively activates the outgoing insertion and deletion transitions even on a match.

Figure 8 illustrates delayed merging. In cycle `C`, we cannot discard the incoming insertion open-path in favor of the matching closed-path, although the latter has a higher score. Instead, we latch its score. In the next cycle, when there is a mismatch, the latched open-path produces the best score for outgoing indel transitions.

After processing the strings, we need to compute the best score. BWA-MEM applies a heuristic called *clipping*, where it selects the best score seen during seed-extension, instead of determining the best score only among the final states at the end of string processing. The reason for this is the expectation that the ends of a read are likely to suffer from

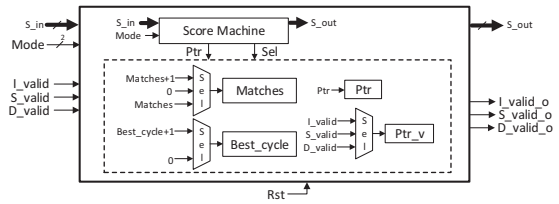

Figure 9: PE for SillaX Traceback Machine.

sequencing machine errors and are less likely to be true variants.

The SillaX scoring machine supports clipping as follows. Each state stores the best score it has seen during the entire computation. Once the strings are processed, the mode of the machine is changed to instruct the states to back-propagate their best scores, through local communication, but now in the reverse direction. Each state receives the best scores from three of its upstream nodes, computes the maximum of those scores along with its own best score, and passes the computed maximum to its downstream paths. In this way the best score seen at any time during the string processing in any state is read out at node `0,0|0`.

Efficiency: A scoring PE includes an edit PE, four scoring registers (`log(N)` bits each), and a programmable scoring logic. Three additional output and input ports are used for communicating scores. It also takes an additional K cycles to back-propagate the scores to the starting node. In spite of these additional overheads, the scoring machine remains efficient in space ($\mathcal{O}(K^2)$) and time ($\mathcal{O}(N)$).

C. Traceback Machine

Read alignment requires the sequence of edits made and their positions in the string for the best solution found. This step is referred to as traceback. Hardware accelerators based on Smith-Waterman [13] typically delegate this step to software, which then becomes the bottleneck, or require hardware space that is proportional to the read length [22].

We support traceback by extending our scoring machine as shown in Figure 9. Our main idea is to use a *pointer trail*, much like how an ant creates a scent trail to get back home. Also, we compress the trace representation in the machine by keeping a count of matches discovered in each state. That is, each state, in addition to tracking the best score it has seen, tracks the number of matches it found at that node for the path corresponding to the best score.

The traceback machine works as follows: In the string matching phase, when a state accepts a score from a downstream state as its best score, it sets its traceback pointer (`2 bits`) to that state. After processing the strings, in phase two, just like in the scoring machine, the best score is propagated back, along with the winning final state's identifier. In phase three, a signal is propagated forward to inform the winner. In phase four, by chasing the pointer trail from the winner, the states that are part of the winning path are flagged. In phase five, the trace is collected at the starting node `(0,0|0)` by shifting the matches and pointer values along the flagged winning path downstream, one state at a time per cycle.

One problem is that a pointer trail may get broken in phase one while processing the strings. When a *greedy* state discovers a higher score, it will discard the previous best score seen in an earlier cycle and its corresponding pointer. However, the previously discovered path that is now being explored in some upstream state may eventually emerge as the winner in the end. But the pointer trail for that winner is now overridden at the greedy state. To address this problem, we ask the greedy state to inform its upstream neighbors when it changes its pointer, which allows its neighbors to invalidate their pointer to that greedy state, thereby indicating that the pointer trail terminates in the neighbors state.

At the end of phase five, our controller examines the trace to check if it is complete. If it is broken, it re-runs the machine till the cycle when the winning path left the greedy state. We determine this cycle by keeping track of the cycle at which the best path seen in a state left that state. This is in addition to the best score that we had in the scoring machine. After the re-run, the machine collects the trace from the greedy state, which is now guaranteed to have the correct pointer. If the machine were to discover a new greedy state, it is resolved by re-running the machine again. In practice, however, we find that it is rare for pointer trails to be broken, and hence re-runs are rare.

Efficiency: Traceback machine has five phases, as opposed to two in the scoring machine. The first phase takes N cycles and the remaining phases take about K cycles each. It also adds a counter for tracking matches and a register for best cycle (each of size `log(N)`). These additional overheads are significantly lower than the $\mathcal{O}(N)$ space complexity in previous hardware accelerators that supported traceback [22].

D. Composable SillaX

A key issue in hardware acceleration is maintaining flexibility so that a wide range of applications can be addressed. In our application this means that an accelerator should address both different string and edit lengths. Our proposed SillaX accelerator already allows arbitrary string length. However, the maximum edit distance is constrained by the the size of the PE grid and is fixed in hardware. To address this issue, we propose the use of composable sub-grids where multiple smaller SillaX engines can be combined into fewer larger engines or one maximum size engine. This creates flexibility where a few high edit distance machines can be reconfigured with a simple mode-switch into multiple smaller edit distance machines, thereby optimally addressing the targeted application space.

The concept is illustrated in Figure 10. Six small SillaX accelerator tiles are labeled by their position in the larger grid ((1,1), (1,2) and (2,1)) and whether they are oriented *forward* (0) or *flipped* (1). Each of these six accelerators can operate independently providing six engines each with the

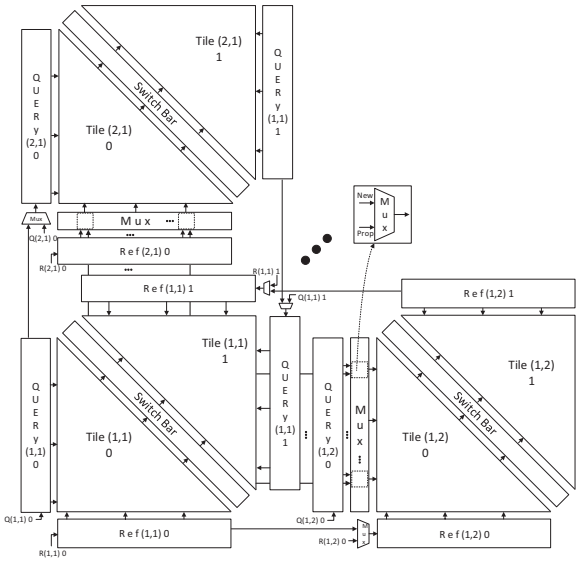

Figure 10: Illustration of composable SillaX

same edit distance K. Note that in the forward oriented tiles, state activation propagates from bottom left to top right as in the previous figures. In the flipped tile, state transitions propagate in the opposite direction. To make a SillaX accelerator with edit distance $2K$, we can combine the following four tiles: (1,1)—0, (1,1)—1, (1,2)—0, (2,1)—0 by changing the configuration of MUXes. In this case, the reference string will stream from Ref(1,1)—0 to Ref(1,2)—0, concatenating the two shift registers. Similarly, the two query registers are concatenated. Also, the connections inside Tile (1,1)—1 are reversed so that state transitions propagate from bottom left to top right. This is accomplished by adding MUXes/tri-state gates at the input/outputs of each PE. They configure which wires are treated as inputs and outputs in an array Tile. Finally, the outputs of Tile (1,1)—1 are fed into the inputs of Tile (1,2)—0 and (2,1)—0 forming a single larger array of PEs instead of 4 smaller ones. Note that in this example, Tile (1,2)—1 and Tile (2,1)—1 are still operating as independent SillaX engines with edit distance K.

This reconfiguration approach incurs only a small overhead of MUXes between tiles and for each PE. It allows many different configurations with edit distances ranging from K to pK where $p = sqrt(T)$ for an implementation with `T` Tiles. This broadens the application space of SillaX.

V. Seeding Accelerator

Problem statement: Seeding determines the positions (*hits*) in the reference genome where there could be potential matches for a read. It does this by finding perfect matches for a given read's substrings (seeds) of length `k` (k-mer) in the reference. To further reduce the number of seeds, BWA-MEM uses a heuristic by defining a seed to be a read's substring that has super-maximal exact matches (SMEMs) with the reference genome. A maximal exact match (MEM) is an exact match that cannot be extended in either direction. An SMEM is a maximal length match (MEM) that is not *fully contained* in any other MEM in the read.

Prior hardware accelerators for seeding directly implement BWA-MEM's FMD-index (Ferragina-Manzini) [32], [33] based seeding, which suffers from poor locality due to irregular memory accesses. We use an implementation that is guaranteed to find all hits as BWA-MEM, but has better locality.

Algorithm: We use an index table that has one entry for each k-mer, which points to a list in a position table [34]. The list contains the *hits* where the k-mer occurs in the reference genome. For each position (*pivot*) in the read, we find a right maximal exact match (RMEM) that is of size at least `k`. To compute RMEM, we determine the hits for the first k-mer starting from the pivot (`H1`). Then we stride by `k`, and find the hits for a k-mer starting at `pivot+k` position in the read (`H2`). We *normalize* these hits to the pivot position by subtracting `k` from their hit values. The set of hits (`H1` and `H2`) are intersected to produce the set of *candidate hits* where we can find the larger string of size `2k`. We can continue this process until the intersection returns an empty set of candidate hits. Then, we reduce the stride progressively from `k/2, k/4, k/8 ..., 1` to compute the RMEM with non-zero candidate hits. We repeat the whole process for each position in the read. The RMEM for the first position in the read is an SMEM. If an RMEM for a later position is a substring of a previously discovered SMEM, it is not reported as a seed, as it is not an SMEM. Seeding returns the hits of all SMEM seeds to the seed-extension step.

Accelerator: We observe that fetching data from the index table and the position table can become a performance bottleneck. To enable greater reuse of the index and position tables across reads during SMEM computation, we segment the genome, and construct index and position tables for each segment. Segmenting also enables the index and position tables to be stored in on-chip SRAM, providing low-latency access, and alleviating the memory bandwidth bottleneck. All reads are processed for one segment, and then repeated for the next segment.

Intersecting hit sets is a performance-critical operation in determining SMEM seeds and their hits. Our seeding accelerator implements several optimizations to optimize this operation. One, we use 512-entry on-chip CAM per seeding lane to compute intersections. We defined its size based on our empirical analysis of k-mer indices for human genomes that showed that most k-mers have less than 512 hits when `k = 12`. Two, if the set of hits of the current k-mer is larger than 512, we do a binary search. A binary search is possible, because position tables are constructed offline for a reference genome, and therefore we can store the hits for a `k-mer` as a sorted list. Three, we find that a common performance issue is when intersecting the hits of the first two k-mers starting from a pivot. We mitigate this problem as follows. Instead of striding by `k` for the second k-mer, we lookup several k-mers with lower strides.

We select the k-mer with the smallest hit set, intersect it with the first k-mer, and continue the RMEM process after that k-mer. Since the size of intersected candidate hits can only decrease, starting RMEM with a small number of hits can reduce the overall number of CAM lookups during the rest of the RMEM computation. Four, we use a variant of the above optimization for quickly seeding reads that have exact matches in the reference genome. We observed that for real world human genome datasets consisting of nearly 1.5 billion short reads, ∼75% of the reads have exact matches in the reference. SMEMs for these reads do not need to be verified by seed-extension. To optimize for this common case, for each read, we lookup the index for a set (of size $\lceil readlength/k \rceil$) of k-mers that span the entire read starting from its beginning, where each k-mer is offset by k. We select the smallest hit set, and then start intersecting with the next smallest, and complete the intersection with all the sets. If the intersection of hit sets of all these k-mers result in a non-empty set, then we have found an exact match for the read in the reference, and therefore we can skip the rest of the above steps for it.

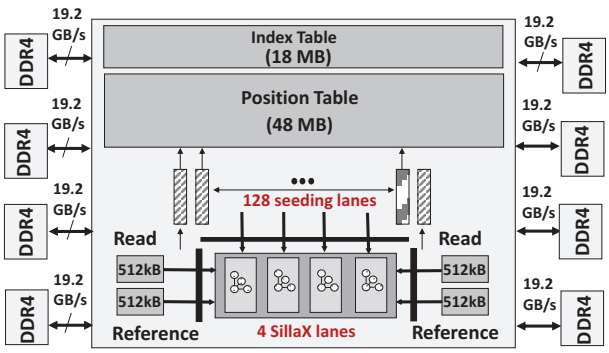

Figure 11: GenAx architecture overview

VI. GenAx Genomics Accelerator

GenAx brings together the seeding and SillaX seed-extension accelerators to enable high-throughput sequence alignment of human genomes. Figure 11 depicts the overall GenAx architecture. It consists of 128 seeding lanes which fetch k-mer positions from a 48 MB index table and k-mer reference hits from a 18 MB position table. Each indexing lane processes one read at a time. It has a CAM and a small control FSM for orchestrating SMEM intersections and k-mer lookups. The resulting hits after SMEM calculations are buffered for seed-extension by the SillaX lanes. GenAx features four SillaX lanes, which have sufficient throughput to process hits from all 128 seeding lanes. A SillaX lane fetches the reference string from the reference cache (4×512 KB) to extend a seed at a specific hit position. A 16 KB buffer (not shown) is used to buffer the reads processed.

The reference genome with 3 billion base-pairs is segmented into 512 segments. Therefore, each segment has 6 million base-pairs and a footprint of 1.5 MB which fits in the reference cache. Segments are processed sequentially for all reads. Before a segment starts, the position table, index table and reference for that segment are streamed in from memory via the 8 DDR4 channels shown. Since these are all spatially co-located memory accesses, streaming them in is efficient.

CPU	Intel Xeon E5-2697 v3 2.6GHz; 2 sockets; 28 cores; 56 threads
L1 I&D cache	14 x 32KB Instruction; 14 x 32KB Data
L2 cache	14 x 256KB
L3 cache	1 x 35MB
Memory	120GB DRAM
GPU	Nvidia TITAN Xp 1.6GHz; 3840 CUDA cores
Shared L2 cache	3MB
Memory	64GB DRAM, GDDR5X

Table I: Baseline system configurations.

VII. Evaluation Methodology

Reference genome and input reads: To build the index and position tables for the reference human genome, we used the latest major release of human genome assembly (GRCh38) from the UCSC genome browser [35] and filtered out unmapped contigs and mitochondrial DNA. Only chromosomes 1-22, X and Y were used. For our evaluation, we use real human genome reads with 50× coverage from the Illumina platinum genomes [36] dataset. The dataset consists of the NA12878 human reference (single-end ERR194147_1.fastq) consisting of 787,265,109 reads of 101 bp length.

System Configuration: We compare GenAx with the de-facto standard: software aligner BWA-MEM [12] running on Intel Xeon E5-2697 CPU operating at 2.6GHz with 56 active threads (best configuration in our environment) and 128GB DDR4 memory. The detailed system configuration is shown in Table I. BWA-MEM uses SMEM-based seeding and a banded Smith-Waterman to compute optimal local alignments. The CPU power is measured using Intel's RAPL Interface. We also compare GenAx with a state-of-the-art GPU aligner (CUSHAW2) [37] on Nvidia's TITAN Xp. CUSHAW2-GPU also identifies maximal-length matches and extends these to form larger gapped alignments. We used the default scoring scheme in BWA-MEM for all the aligners. To study SillaX's alignment throughput independent of the seeding accelerator, we compare against major software implementations of the Smith-Waterman algorithm. We use the SeqAn library [38] as the CPU baseline, and SW# [39] as the GPU baseline.

Synthesis: We synthesized the SillaX accelerator using the Synopsys Design Compiler (DC) in a commercial 28nm process. We synthesized all three Silla machines: edit, scoring, and traceback, to obtain their area, power, and latency with respect to different clock frequency targets.

GenAx performance modeling: We segmented the reference genome into 512 segments and constructed index and position tables for each segment. The reads are processed

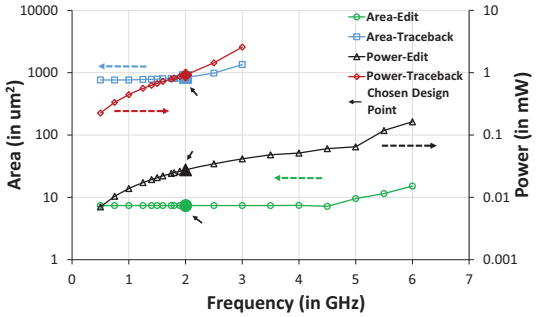

Figure 12: SillaX area and power for a single PE.

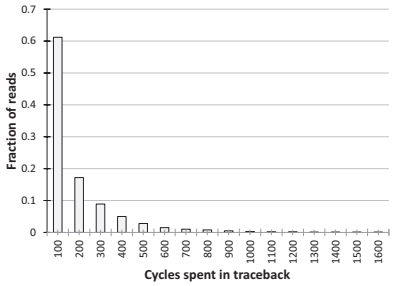

Figure 13: Silla traceback cycle distribution.

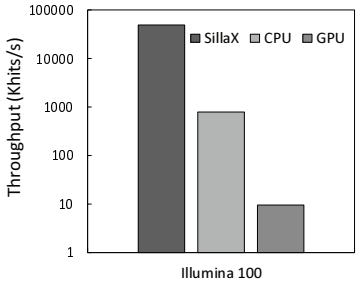

Figure 14: SillaX throughput (in Khits/s).

sequentially in each segment. For a k-mer size of 12, the index and position table require 48 MB and 18 MB of on-chip SRAM respectively. By choosing a small k-mer size such as 12, our index table does not require additional tag meta-data to handle collisions. Also accesses to the both index and position table to compute SMEMs benefit from the low latency and 100% hit rate provided by on-chip SRAM. We used Ramulator [40] to compute the memory cycles required to load these tables for each segment, load the reference and reads for SillaX compute.

VIII. Results

A. SillaX Evaluation

Area, Frequency and Power Figure 12 shows the power and area for each processing element (PE) in the SillaX edit machine and traceback machine. The optimal design points are highlighted. Scoring machine is comparable to the traceback machine, so we omit it. 2 GHz is the inflection point. At 2 GHz, the SillaX edit machine has an area of 0.012 mm^2, power of 0.047 W and latency of 0.17 ns. At the same clock frequency, the traceback machine has an area of 1.41 mm^2, power of 1.54 W, and latency of 0.33 ns.

BWA-MEM reports alignments with score higher than 30. Using this estimate, we can derive that the edit distance (`K`) should be less than 32. Given this, we conservatively use `K = 40` in our analysis. To support `K = 40`, SillaX uses 1,681 processing elements (PEs).

GRCh38 Human Genome Assembly Validation: To evaluate the accuracy of the SillaX traceback machine, we ran all the the non-exact matching reads in the `ERR194147_1.fastq` file and compared the alignments produced with that from BWA-MEM. Exact matching reads are trivially identified using a single state SillaX machine that transitions to itself every cycle on a match.

For all the 351,023,283 non-exact matching reads, the SillaX traceback machine machine alignment results concur with the BWA-MEM's alignments with negligible (0.0023%) variance. On investigating the different alignments further, we noticed that the alignment scores produced by the SillaX traceback machine are exactly the same as that of BWA-MEM, implying that both alignments have the same mapping quality and should be treated the same. These differences are due to the fact that BWA-MEM and SillaX use different traceback techniques and policies for breaking ties when merging multiple paths with the same score.

Broken pointer trail events: An important parameter to be considered while estimating the performance of the SillaX traceback machine is the number of times the machine must be re-executed because of a broken pointer trail. Across all the reads that we tested, we observe that only 7.59% of the reads require re-execution . This is consistent with our expectation. Figure 13 shows the distribution of cycles spent in re-execution. We can see that over 60% of the re-execution events are resolved within the first N (101) cycles. Thus, re-execution events have only a small impact on the performance of the SillaX traceback machine.

Throughput: Figure 14 shows the raw alignment throughput of the SillaX accelerator (4 lanes) when compared to banded Smith-Waterman based approximate string matching using SeqAn (CPU - 28 cores) and SW# (GPU - 3840 CUDA cores) when aligning 100`bp` Illumina short reads. It can be seen that SillaX achieves ∼62.9× throughput improvement over SeqAn and ∼5287× speedup over SW#. SillaX provides these speedups while consuming only 6.6 W of power and 5.64 mm^2 area. These benefits are both due to linear time processing of input symbols as well as efficient support for traceback. GPU-based solutions face high synchronization overheads for short reads leading to low performance.

B. GenAx Evaluation

Throughput, Power and Area: Figure 15 (a) compares the overall throughput (reads/s) of GenAx with BWA-MEM (CPU) and CUSHAW2-GPU. It can be seen that

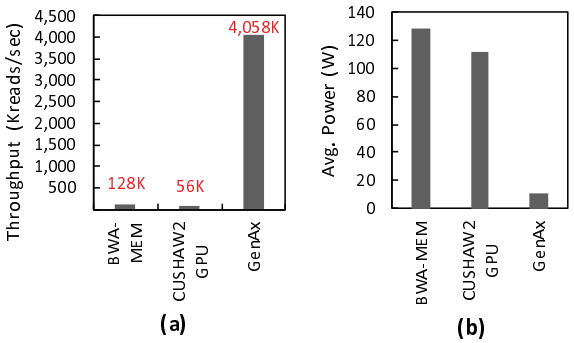

Figure 15: (a) Throughput comparison (in KReads/s) and (b) Power comparison.

GenAx achieves 31.7× speedup over BWA-MEM and 72.4× speedup over CUSHAW2-GPU. The large performance gains can be attributed to the following factors. (1) Efficient and composable SillaX accelerators accelerates seed-extension with in-place traceback. (2) Segmenting of index and position tables and storing them on-chip enables low-latency access and high-reuse across reads. (3) Read loading time takes a small fraction of the overall execution time (∼10%), increasing the benefits from segmenting. (4) Optimizing for the common case of perfect matches helps increase throughput.

Figure 15 (b) compares the average power consumption of GenAx vs. BWA-MEM and CUSHAW2-GPU. By sharing several indexing lanes with SillaX accelerators, GenAx reduces power consumption by 12× when compared to BWA-MEM execution on CPU.

Component	Area (in mm^2)
Seeding lanes (x128)	4.224
SillaX lanes (x4)	5.36
On-chip SRAM (68 MB)	163.2
Total	172.78

Table II: Area breakdown: GenAx.

Table II shows the area breakdown of GenAx. A large fraction of the die area is devoted for accelerating the seeding step using on-chip index and position tables. Each seeding lane consists of a 512-entry CAM. The SillaX lanes consist of a few counters and logic gates as described in § IV-C. Overall, the GenAx architecture takes up 172.78 mm^2 in a 28nm node.

Seeding performance breakdown: While we could have used Burrows-Wheeler Transform (BWT), one of the mainstream solutions for genetic string indexing, it suffers from irregular memory accesses. Naive implementations of a hashing solution, on the other hand, require handling a large number of hit positions in return for better locality. Figure 16 (a) shows the average number of hits generated by the hash table (lower is better for processing). We observe our proposed optimizations, i.e. SMEM and binary extension, can filter out the insignificant hits, resulting in

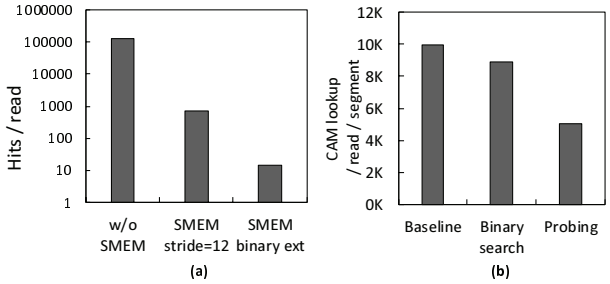

Figure 16: Seeding accelerator optimizations.

reduced workload for the SillaX machine downstream by orders of magnitude.

Figure 16 (b) presents the reduction of CAM lookups from position table lookup optimizations. Since binary lookup of the position table results in logarithmic search time, the number of CAM lookups also decreases in proportion to the search time. Moreover, since certain k-mers are known to have large number of hit positions (e.g. AA...A and ATAT...A), probing effectively helps to find a better starting point with a k-mer having fewer hit positions, reducing the overall CAM lookups.

C. Comparison with Banded Smith-Waterman

Banded Smith-Waterman focuses on identifying near-exact matches (less than K edits) between genomic strings [27], similar to SillaX. Software-based banded Smith-Waterman implementations, however, have $\mathcal{O}(KN)$ time and space complexity. Hardware-based systolic implementations require $\mathcal{O}(N)$ time with $2K+1$ processing elements and additional $\mathcal{O}(KN)$ space for traceback. In contrast, SillaX differs from prior banded Smith-Waterman implementations in the following ways.

Each PE in SillaX has 30× lower area than a banded Smith-Waterman PE when edit distance is used as the scoring scheme (300 um^2 vs 9.7 um^2 for SillaX at 5 GHz). Assuming a conservatively high K (=32) for aligning Illumina short reads, both SillaX edit machine and scoring machine achieve better area efficiency compared to banded Smith-Waterman because of fewer gates used in each PE.

Furthermore, SillaX enables efficient in-place traceback within PEs. Hardware-based banded Smith-Waterman requires additional $\mathcal{O}(KN)$ space for traceback. Hirschberg's algorithm [41] reduces space to $\mathcal{O}(K)$, but increases time to $\mathcal{O}(NlogN)$. There exists no prior accelerator that supports traceback in $\mathcal{O}(K^2)$ space or lesser without sacrificing time complexity.

Since Silla is based on automata theory, it can be easily mapped to versatile automata processors supporting variable-width input symbols such as UDP [30] providing greater flexibility in implementation. From the algorithmic viewpoint, besides the fact that Silla is as an important successor to Levenshtein automata, it can also be easily extended to solve other important problems such as Longest Common Sequence problem and automatic spell correction, as well

as ones in the bioinformatics domain [42].

IX. Related Work

Hardware accelerators for sequence alignment: This paper advances automata based hardware accelerators for seed extension. We related this contribution to prior work in depth in Section § II. Several other hardware accelerators have been proposed to accelerate popular software tools like BWA and Bowtie, and have demonstrated upto 10× performance improvement [43], [44], [45]. These works use Burrows-Wheeler Transform (BWT) index to compute *exact matches* in $\mathcal{O}(N)$ time. To find approximate matches, they require expensive recursive steps, whose time complexity is exponential with edits. SillaX can find approximate string matches in $\mathcal{O}(N)$ time for a given edit distance K. Industry efforts to accelerate the BWA-GATK pipeline include those of Edico Genome's DRAGEN Bio-IT platform [17] and Time Logic's Tera-BLAST [46]. Based on their white paper and patents, DRAGEN implements banded Smith-Waterman for seed extension, and uses a variant of an algorithm used in the LAST aligner [47] for seeding.

Seeding techniques and optimizations: Our seeding accelerator mimics the seeding step in BWA-MEM which computes super-maximal exact matches (SMEMs). Prior hardware accelerators either use hash-tables or Burrows-Wheeler transform (BWT) for seeding [48], [49]. For example, DRAGEN makes hash table index queries to iteratively grow the seed and ensure that fewer than 16 hits are fetched on-chip (64B data). However, restricting seed hits to 16 can lead to loss in accuracy [25]. On the other hand, SMEM computation using BWT has poor cache locality due to highly irregular memory accesses, and is hard to accelerate. In this work, we improve the locality of SMEM computation by segmenting the index and position tables of the whole genome into several chunks, and storing them in on-chip SRAM to improve reuse across k-mers in reads.

Complementary to our work, there is also a rich body of work that optimizes cache locality by indexing the reads [50], parallelizes read alignment across multiple nodes by identifying I/O bottlenecks [51], [52] and addressing them using unified file formats [53].

X. Conclusion

Genomics is at an inflection point. Over the next decade, it is conceivable that every individual's genome would be sequenced and analyzed. Given that a single human genome generates over 300 GB of data, we need orders of magnitude improvement in computing efficiency to realize the full potential of genomics. This paper takes an important step towards this goal by presenting an accelerator that improves the efficiency of sequence aligners. GenAx provides throughput of 4,058K reads/s for Illumina 101 `bp` reads. GenAx achieves 31.7× speedup over the standard BWA-MEM sequence aligner running on a dual-socket 14-core Xeon E5 server processor, while reducing power consumption by 12× and area by 5.6×.

Acknowledgment

We thank our shepherd Mark Oskin and the anonymous reviewers for their suggestions which helped improved this paper. This work was supported in part by NSF CAREER-1149773, CAREER-1652294 and SHF-1527301 awards.

References

[1] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh *et al.*, "Initial sequencing and analysis of the human genome," *Nature*, vol. 409, no. 6822, pp. 860–921, 2001.

[2] "Illumina," https://www.forbes.com/sites/matthewherper/2017/01/09/illumina-promises-to-sequence-human-genome-for-100-but-not-quite-yet.

[3] M. A. Hamburg and F. S. Collins, "The path to personalized medicine," *N Engl J Med*, vol. 2010, no. 363, pp. 301–304, 2010.

[4] E. D. Pleasance, R. K. Cheetham, P. J. Stephens, D. J. McBride, S. J. Humphray, C. D. Greenman, I. Varela, M.-L. Lin, G. R. Ordóñez, G. R. Bignell *et al.*, "A comprehensive catalogue of somatic mutations from a human cancer genome," *Nature*, vol. 463, no. 7278, pp. 191–196, 2010.

[5] A. Lacour, A. Espinosa, E. Louwersheimer, S. Heilmann, I. Hernández, S. Wolfsgruber, V. Fernández, H. Wagner, M. Rosende-Roca, A. Mauleón *et al.*, "Genome-wide significant risk factors for alzheimers disease: role in progression to dementia due to alzheimer's disease among subjects with mild cognitive impairment," *Molecular psychiatry*, vol. 22, no. 1, pp. 153–160, 2017.

[6] Y. Cho, C.-H. Lee, E.-G. Jeong, M.-H. Kim, J. H. Hong, Y. Ko, B. Lee, G. Yun, B. J. Kim, J. Jung *et al.*, "Prevalence of rare genetic variations and their implications in ngs-data interpretation," *Scientific Reports*, vol. 7, no. 1, p. 9810, 2017.

[7] E. Hanna, C. Rémuzat, P. Auquier, and M. Toumi, "Gene therapies development: slow progress and promising prospect," *Journal of Market Access & Health Policy*, vol. 5, no. 1, p. 1265293, 2017.

[8] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: astronomical or genomical?" *PLoS biology*, vol. 13, no. 7, p. e1002195, 2015.

[9] "Facebook data," https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb.

[10] S. Kumar, K. K. Krishnani, B. Bhushan, and M. P. Brahmane, "Metagenomics: retrospect and prospects in high throughput age," *Biotechnology research international*, vol. 2015, 2015.

[11] "Oxford nanopore minion." https://nanoporetech.com/products/minion.

[12] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with bwa-mem," *arXiv preprint arXiv:1303.3997*, 2013.

[13] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[14] M. Farrar, "Striped smith–waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2006.

[15] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *Journal of the ACM (JACM)*, vol. 46, no. 3, pp. 395–415, 1999.

[16] Y.-T. Chen, J. Cong, J. Lei, and P. Wei, "A novel high-throughput acceleration engine for read alignment," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 2015, pp. 199–202.

[17] R. McMillen and M. Ruehle, "Bioinformatics systems, apparatuses, and methods executed on an integrated circuit processing platform," https://www.google.com/patents/

US9014989, Apr. 21 2015, uS Patent 9,014,989.

[18] I. Roy and S. Aluru, "Discovering motifs in biological sequences using the micron automata processor," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 13, no. 1, pp. 99–111, 2016.

[19] I. Tommy Tracy, M. Stan, N. Brunelle, J. Wadden, K. Wang, K. Skadron, and G. Robins, "Nondeterministic finite automata in hardware-the case of the levenshtein automaton."

[20] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache automaton," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 259–272.

[21] O. Gotoh, "Optimal sequence alignment allowing for long gaps," *Bulletin of mathematical biology*, vol. 52, no. 3, pp. 359–373, 1990.

[22] 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.

[23] J. J. Tithi, N. C. Crago, and J. S. Emer, "Exploiting spatial architectures for edit distance algorithms," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 23–34.

[24] M. Šošić and M. Šikić, "Edlib: a c/c++ library for fast, exact sequence alignment using edit distance," *Bioinformatics*, vol. 33, no. 9, pp. 1394–1395, 2017.

[25] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler, "Faster and more accurate sequence alignment with snap," *arXiv preprint arXiv:1111.5572*, 2011.

[26] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller, "A greedy algorithm for aligning dna sequences," *Journal of Computational biology*, vol. 7, no. 1-2, pp. 203–214, 2000.

[27] B. Harris, A. C. Jacob, J. M. Lancaster, J. Buhler, and R. D. Chamberlain, "A banded smith-waterman fpga accelerator for mercury blastp," in *2007 International Conference on Field Programmable Logic and Applications*, Aug 2007, pp. 765–769.

[28] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.

[29] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "Hare: Hardware accelerator for regular expressions," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[30] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien, "Udp: A programmable accelerator for extract-transform-load workloads and more," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 55–68. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123983

[31] P. Mitankin, "Universal levenshtein automata. building and properties," *Sofia University St. Kliment Ohridski*, 2005.

[32] H. Li, "Exploring single-sample snp and indel calling with whole-genome de novo assembly," *Bioinformatics*, vol. 28, no. 14, pp. 1838–1844, 2012.

[33] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, 2000, pp. 390–398.

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

[35] "Uscs genome browser." https://genome.ucsc.edu/.

[36] M. A. Eberle, E. Fritzilas, P. Krusche, M. Källberg, B. L. Moore, M. A. Bekritsky, Z. Iqbal, H.-Y. Chuang, S. J. Humphray, A. L. Halpern *et al.*, "A reference data set of 5.4 million phased human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree," *Genome research*, vol. 27, no. 1, pp. 157–164, 2017.

[37] Y. Liu and B. Schmidt, "Cushaw2-gpu: empowering faster gapped short-read alignment using gpu computing," *IEEE Design & Test*, vol. 31, no. 1, pp. 31–39, 2014.

[38] A. Döring, D. Weese, T. Rausch, and K. Reinert, "Seqan an efficient, generic c++ library for sequence analysis," *BMC bioinformatics*, vol. 9, no. 1, p. 11, 2008.

[39] M. Korpar and M. Šikić, "Sw#–gpu-enabled exact alignments on genome scale," *Bioinformatics*, vol. 29, no. 19, pp. 2494–2495, 2013.

[40] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[41] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Commun. ACM*, vol. 18, no. 6, pp. 341–343, Jun. 1975.

[42] E. Kopylova, L. Noé, and H. Touzet, "Sortmerna: fast and accurate filtering of ribosomal rnas in metatranscriptomic data," *Bioinformatics*, vol. 28, no. 24, pp. 3211–3217, 2012.

[43] E. B. Fernandez, W. A. Najjar, S. Lonardi, and J. Villarreal, "Multithreaded fpga acceleration of dna sequence mapping," in *2012 IEEE Conference on High Performance Extreme Computing*, Sept 2012, pp. 1–6.

[44] H. M. Waidyasooriya, M. Hariyama, and M. Kameyama, "Implementation of a custom hardware-accelerator for short-read mapping using burrows-wheeler alignment," in *2013 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, July 2013, pp. 651–654.

[45] H. M. Waidyasooriya and M. Hariyama, "Hardware-acceleration of short-read alignment based on the burrows-wheeler transform," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1358–1372, May 2016.

[46] R. Luethy and C. Hoover, "Hardware and software systems for accelerating common bioinformatics sequence analysis algorithms," *Drug Discovery Today: BIOSILICO*, vol. 2, no. 1, pp. 12–17, 2004.

[47] S. M. Kiełbasa, R. Wan, K. Sato, P. Horton, and M. C. Frith, "Adaptive seeds tame genomic sequence comparison," *Genome research*, vol. 21, no. 3, pp. 487–493, 2011.

[48] M.-C. F. Chang, Y.-T. Chen, J. Cong, P.-T. Huang, C.-L. Kuo, and C. H. Yu, "The smem seeding acceleration for dna sequence alignment," in *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*. IEEE, 2016, pp. 32–39.

[49] Y.-C. Wu, C.-H. Chang, J.-H. Hung, and C.-H. Yang, "A 135-mw fully integrated data processor for next-generation sequencing," *IEEE Transactions on Biomedical Circuits and Systems*, 2017.

[50] F. Hach, F. Hormozdiari, C. Alkan, F. Hormozdiari, I. Birol, E. E. Eichler, and S. C. Sahinalp, "mrsfast: a cache-oblivious algorithm for short-read mapping," *Nature methods*, vol. 7, no. 8, p. 576, 2010.

[51] J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo, "Sparkbwa: speeding up the alignment of high-throughput dna sequencing data," *PloS one*, vol. 11, no. 5, p. e0155461, 2016.

[52] A. Roy, Y. Diao, U. Evani, A. Abhyankar, C. Howarth, R. Le Priol, and T. Bloom, "Massively parallel processing of whole genome sequence data: An in-depth performance study," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 187–202.

[53] S. Byma, S. Whitlock, L. Flueratoru, E. Tseng, C. Kozyrakis, E. Bugnion, and J. Larus, "Persona: A high-performance bioinformatics framework," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 153–165. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/byma