ASAP: Accelerated Short-Read Alignment on Programmable Hardware

Subho Sankar Banerjee, Mohamed El-Hadedy, Jong Bin Lim, Zbigniew T. Kalbarczyk, Deming Chen, Steven S. Lumetta, and Ravishankar K. Iyer

Abstract—The proliferation of high-throughput sequencing machines ensures rapid generation of up to billions of short nucleotide fragments in a short period of time. This massive amount of sequence data can quickly overwhelm today's storage and compute infrastructure. This paper explores the use of hardware acceleration to significantly improve the runtime of short-read alignment, a crucial step in preprocessing sequenced genomes. We focus on the Levenshtein distance (edit-distance) computation kernel and propose the ASAP accelerator, which utilizes the intrinsic delay of circuits for edit-distance computation elements as a proxy for computation. Our design is implemented on an Xilinx Virtex 7 FPGA in an IBM POWER8 system that uses the CAPI interface for cache coherence across the CPU and FPGA. Our design is $200\times$ faster than an equivalent Smith-Waterman-C implementation of the kernel running on the host processor, $40-60\times$ faster than an equivalent Landau-Vishkin-C++ implementation of the kernel running on the IBM Power8 host processor, and $2\times$ faster for an end-to-end alignment tool for 120–150 base-pair short-read sequences. Further the design represents a $3760\times$ improvement over the CPU in performance/Watt terms.

Index Terms—Bioinformatics, genomics, levenshtein distance, application-specific processor, hardware accelerator

1 Introduction

The advent of high-throughput next-generation sequencing technology (NGS) has created a deluge of genomic data for computational analysis [1]. Efficiently processing this data requires the development of a new generation of high-performance computing systems that can efficiently handle such data. This new generation of application-specific and accelerator-rich computing systems are expected to gain performance, power, and energy improvements over traditional systems [2].

A crucial step in a significant number of NGS data analytics applications (e.g., variant discovery, genome-wide association studies, and phylogeny creation) is the mapping of short fragments of sequenced genetic material (called *reads*) to their most likely points of origin in the genome, popularly called the *short-read alignment* problem. This paper presents the design and implementation of ASAP, an accelerator for computing Levenshtein distance [3], [4] (LD; used interchangeably with edit-distance) in the context of the short-read alignment problem. LD is a measure of the similarity between strings, which is computed by counting the number of single-character edits required to change one string into the other. LD computation is a prominent underlying mathematical kernel that is common to a large number of short-read alignment algorithms and tools (e.g., BLAST [5], Bowtie [6], [7], BWA [8], and SNAP [9]), and is responsible for 50–70 percent of their runtime [10].

ASAP represents a novel approach to accelerate the LD computation, in that it uses algorithmic approximations, and maps these approximations into hardware to significantly improve overall performance ($\sim 200\times$ compared to the CPU baseline). The core algorithm in ASAP leverages two key observations about the computation and datasets involved in the short-read alignment problem:

1) Although all the tools mentioned above calculate the exact value of LD between pairs of nucleotide strings, they use them only *to build a total ordering* (i.e., an ordered list) of the most likely points of origin in the genome. The best alignment is the pair of strings corresponding to the minimum LD in the ordered list. Hence, it is sufficient to only calculate the total ordering (in this instance, returning the pair that corresponds to the minimum LD), and not essential to compute the exact value of the LD. This distinction enables approximation in the computation of LD to gain performance, while preserving the overall accuracy of the alignment algorithm (which comes from the total ordering).
2) Modern sequencing platforms (like the Illumina HiSeq 2,500) represent a very low sequencing error regime ($\leq 1\%$) [11], [12], and modern alignment tools (mentioned above) have accurate candidate region-matching algorithms (described in Section 2). Hence, LD computations process significantly more "matches" than "mismatches," in the majority of

• *The authors are with the Coordinated Science Laboratory, and the Departments of Computer Science and Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801. E-mail: {ssbaner2, hadedy, lim43, kalbarcz, dchen, lumetta, rkiyer}@illinois.edu.*

Manuscript received 13 Dec. 2017; revised 31 July 2018; accepted 10 Sept. 2018. Date of publication 11 Oct. 2018; date of current version 19 Feb. 2019. (Corresponding author: Subho Sankar Banerjee.)
Recommended for acceptance by L. Eeckhout.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2018.2875733

sequencing experiments.[1] The ASAP architecture uses this heuristic to accelerate LD computation (described in Sections 3.1 and 3.2).

To take advantage of these observations, ASAP augments RaceLogic [13][2] using application heuristics, as well as hardware architectural optimizations to realize the design on FPGAs. In particular, this paper proposes (a) a mechanism to encode LD computation parameters (e.g., *gap-penalties*; described further in Section 2) into the ASAP architecture, making it possible to map the time taken to process a "match" exactly as a circuit delay. This mapping gives us the ability to tune the performance of ASAP to match data characteristics; and (b) the use of "zero delay" circuit elements to explore large portions of the search space (LDs of substrings of the strings being compared) in parallel within one clock cycle, and to ignore parts of the search space that do not contribute to an answer, thereby saving energy. Overall, ASAP can compute alignments quickly ($\sim 200\times$ faster than the CPU baseline and $\sim 50\times$ faster than an equivalent RaceLogic design), and with the same accuracy as traditional software- or hardware-based alignment tools. We leverage reconfigurable FPGA devices to prototype ASAP, thereby allowing us to reconfigure the accelerator based on user decisions on input parameters (described in Section 2), as well as to adapt the accelerator to input NGS datasets of varying read lengths.

Contributions. To summarize, the primary contributions of this paper are as follows:

1) Presents a measurement-driven study that demonstrates that computation of LD represents a significant portion of the runtime of several short-read alignment programs.
2) Builds on top of the delay-based computation paradigm presented in [13] to encode gap-penalties as "zero delay" circuit elements. This allows us to calculate approximate the LD between strings by using combinational circuit elements. We prove the correctness of this encoding and demonstrate that the result of the approximation can be used as a proxy for computing LD in short-read aligners. That is, a tool using the approximation and the accelerator produces alignments identical to those of tools based on traditional methods (e.g., BWA-MEM [8]).
3) Presents an FPGA-based implementation of the accelerated LD computation in the ASAP accelerator that leverages the coherent accelerator-processor interface (CAPI) [14], [15] for communication between the host and accelerator.
4) Demonstrates that ASAP on an FPGA is able to accelerate the runtime of the LD computation by $200\times$ compared to a Smith-Waterman-based and $40-60\times$ compared to a Landau-Vishkin based IBM Power8 CPU execution. As well as $5\times$ better that competing FPGA implementations.
5) Demonstrates that integration of the ASAP accelerator into a short-read alignment frameworks SNAP and BWA-MEM. In both cases this results in a $2\times$, $1.9\times$ performance improvement respectively, which is close to the Amdahl's law limits for these applications.

Other Applications. Our approach can be adapted to a variety of other problems in which a total ordering of LDs is computed. For example, in signal processing, where similarity between signals is computed [3]; in text retrieval, where misspelled words have to be accounted for in a dictionary [16]; and in computer-security where virus- and intrusion-detection requires comparison of signatures [17].

Organization. The remainder of this paper is organized as follows. Section 2 describes LD computation and its use in popular short-read alignment tools. Section 3 briefly describes 1) a mathematical formalism for encoding computation in circuit delays; 2) the approximation at the core of ASAP and prove its correctness; and 3) presents the hardware architecture of ASAP leverages this approximation algorithm. Section 4 presents the evaluation of the accelerator. Section 5 compares the ASAP approach to other hardware accelerated approaches for computing LD, and, finally, we conclude in Section 6.

2 Levenshtein Distance Computation and Short-Read Alignment

Traditional methods for aligning reads to a reference genome find the position (*locus*) of a single read in the reference by minimizing the maximum edit distance between the short read being aligned (called the *query*, and denoted by Q) and the reference genome sequence. The Smith-Waterman algorithm (SW) [18] and Needleman-Wunsch algorithm (NW) [19] utilize a dynamic programming-based algorithm to calculate the alignment score (Levenshtein distance) between the read and a particular section R of the reference genome, accounting for base pair substitutions, insertions, and deletions. Both of these algorithms work by constructing a matrix S (which is used interchangeably with lattice) of size $l_Q \times l_R$, where l_Q and l_R are the lengths of the two strings, between which the edit distance must be calculated. Consider a matrix S in which the (i,j)th entry, $S(i,j)$, is the minimum edit distance between the sub strings $Q[1:j]$ and $R[1:i]$. $S(i,j)$ is recursively defined as

$$S(i,j) = min\left\{\begin{array}{l} S(i-1,j) + \Delta(-,R_j), \\ S(i,j-1) + \Delta(Q_i,-), \\ S(i-1,j-1) + \Delta(Q_i,R_j) \end{array}\right\}, \quad (1)$$

where Δ corresponds to input parameters called *gap penalties*. These Δ-parameters assign scores for insertion, deletion, match,[3] or mismatch between the sequences such that a more desirable outcome has a smaller score associated with it. The parameters $\Delta(Q_i,R_j)$, $\Delta(-,R_j)$ and $\Delta(Q_i,-)$ correspond to the match/mismatch, deletion, and insertion penalties respectively. These parameters are chosen to optimize the accuracy of alignments based on prior information about the sequences being compared (e.g., evolutionary

1. This is a facet of the accurate sequencing process and the thoroughly validated reference genome for human subjects. This observation will also apply to most model organisms whose genome has been extensively studied.

2. RaceLogic uses propagation delay of circuit elements to perform computations.

3. Gap penalties traditionally do not have match scores. We group them together for simplicity in our notation.

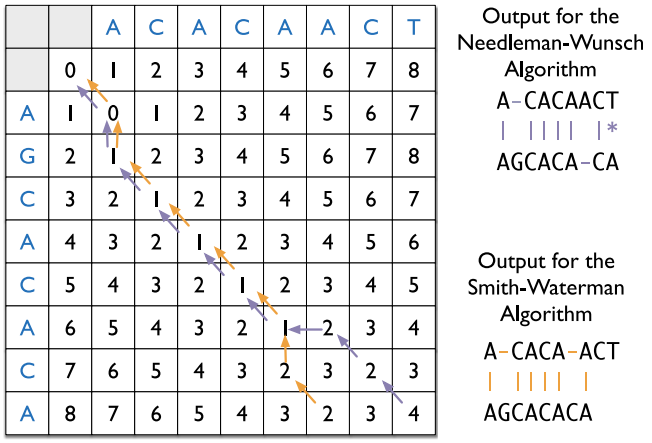


Fig. 1. The matrix S for the strings AGCACACA and ACACAACT, assuming $\Delta(\text{Match}) = 0$, $\Delta(\text{Mismatch}) = 2$, and $\Delta(\text{Insert}) = \Delta(\text{Delete}) = 1$. The colored paths from $S(8,8)$ and $S(8,6)$ to $S(0,0)$ show the optimal alignments produced by the NW and SW algorithms, respectively.

information about mutations in a population [20], [21], [22]). This paper describes the use of constant gap penalties (i.e., a fixed score is assigned to every gap between nucleotides). That is

$$\left.\begin{array}{rcl} \Delta(Q_i, R_j) & = & \Delta(\text{Match}) \text{ if } Q_i = R_j \\ \Delta(Q_i, R_j) & = & \Delta(\text{Mismatch}) \text{ if } Q_i \neq R_j \\ \Delta(-, R_j) & = & \Delta(\text{Delete}) \\ \Delta(Q_i, -) & = & \Delta(\text{Insert}) \end{array}\right\} \forall\, R_i, Q_j. \quad (2)$$

Such gap penalties are are commonly used in DNA alignment (e.g., in NCBI-BLASTN, or WU-BLASTN [20]).

The NW algorithm computes a global alignment in which the entirety of the query is matched to the reference, as shown in Fig. 1. It does so by computing the value of $S(m, n)$. The SW algorithm computes a local alignment and matches the largest (substring) of the query to the reference, and, hence, needs to calculate the minimum value in the row $S(m, -)$. For example, when the strings AGCACACA and ACACAACT are compared with constant penalties $\Delta(\text{Match}) = 0$, $\Delta(\text{Mismatch}) = 2$, and $\Delta(\text{Insert}) = \Delta(\text{Delete}) = 1$, we get the matrix described in Fig. 1. The optimal alignment is then calculated from this matrix by finding the minimum weighted path (in S) from (m, n) to $(0, 0)$ in the NW algorithm and $(m, \mathcal{N})$ to $(0, 0)$ in the SW algorithm. $\mathcal{N}$ corresponds to the largest substring of the reference to which the query string maps with the lowest LD.

Although these methods are guaranteed to produce the optimal alignment, they are prohibitively expensive for whole-genome alignments because of $O(l_Q \times l_R)$ space and time complexity. Therefore, a large number of alignment tools are designed to heuristically reduce the search space required to find the optimal match of a query in the reference. An extensive amount of research, e.g., [5], [6], [7], [8], [9], has been conducted, focusing on indexing strategies for the reference genome to rapidly reduce the number of candidate locations that have to be searched. Most of these tools use some variant of a backwards search algorithm utilizing an FM-index [26] or a hash-table-like data structure. As a result of this reduction in the search space, linear-time heuristic algorithms like the Landau-Vishkin algorithm (LV) [25] (in addition to traditional algorithms like SW and NW) can be applied to the sequence alignment problem in SNAP [9], to compute edit distance accurately up to a particular number of mismatches (assuming that correct alignments have lower numbers of mismatches). Algorithm 1 describes the skeleton of these heuristic accelerated algorithms for single-ended read alignment [27]. The definitions of the Build_Index, Candidate_Locations, Edit_Distance, and Find_Config functions define different variants of these algorithms. For example, Table 1 defines the BWA-MEM and SNAP alignment tools by substituting these placeholder functions with specific algorithms.

Algorithm 1. Algorithmic Skeleton for Single-Ended Short-Read-Alignment Algorithms

Data: NGS Read Dataset, Reference Genome
Result: Aligned positions and mapping of reads in Reference Genome

1 $ngsdata \leftarrow$ Set of reads;
2 $reference \leftarrow$ String(s) corresponding to a reference;
3 $index \leftarrow$ Build_Index($reference$);
4 $alignment \leftarrow \emptyset$;
5 **for** $read \in ngsdata$ **do**
6 $locs \leftarrow$ Candidate_Locations($read, index$);
7 $opt \leftarrow arg\,min_{loc \in locs}$(Edit_Distance($read, loc$));
8 $config \leftarrow$ Find_Config($read, opt$);
9 $alignment \leftarrow alignment \cup config$;
10 **end**
11 **return** $alignment$;

We performed a profiling study of the SNAP aligner on an in-sillico (from an Illumina HiSeq 2,500) whole human genome [28] with $50\times$ coverage (i.e., each nucleotide of the reference is backed by an average of 50 reads that align to that base) on the Blue Waters [29] supercomputer. We chose the SNAP aligner in particular because it is significantly faster than other alignment tools like BWA and Bowtie. Also, as the LV algorithm used in SNAP has a linear time complexity, its comparison to ASAP as the CPU baseline is much more challenging. Table 2 describes the distribution of runtime across for the SNAP aligner for corresponding steps of Algorithm 1.[4] These measurements, along with static analysis of Algorithm 1, show the following:

1) The LD computation corresponds to nearly 60 percent of the running time of the SNAP aligner.
2) The LD computation is one of the most frequently called algorithmic kernels in the alignment process (on average called 54.1 times per read).
3) The LD kernel is used to build a total ordering of all candidate locations for a read in the reference; refer to Line 7 of Algorithm 1.
4) The backtrack-based alignment [18], [19] is computed only for the best-matched location in the reference.
5) The remaining portion of SNAP's runtime (after the LD computation) is spent in either memory or IO bound computation (e.g., hash table look-ups and

4. Note that some steps of the SNAP aligner implementation includes a variety of other miscellaneous tasks, e.g., memory allocation, IO. These are collectively described in the "Misc" category. Also note, the SNAP aligner is optimized to perform asynchronous pre-fetch based disk IO. Hence wait time for IO is minimized.

TABLE 1
Mathematical Formulation of Different Aligners to Fit them into the Structure of Algorithm 1

Function	BWA-MEM [8]	SNAP [9]
`Build_Index`	Burrows-Wheeler transform [23] of prefix trie	Ukkonen's algorithm [24]
`Candidate_Locations`	Prefix trie traversal	Hash table lookup
`Edit_Distance`	Smith-Waterman algorithm [18]	Landau-Vishkin algorithm [25]
`Find_Config`	Smith-Waterman algorithm [18]	Landau-Vishkin algorithm [25]

reading/writing files). This part is unsuitable for acceleration on PCIe-based devices because of the time-cost associated with performing data transfer over the bus.

3 Design of the ASAP Accelerator

This section describes the approximation algorithm that drives the design of ASAP, provides a proof for its correctness, and describes its implementation in programmable hardware. Section 3.1 briefly summarizes the RaceLogic paper [13], describing an formalizing the encoding the computation of LD scores into circuit propagation delay. Section 3.2 describes the approximation at the heart of ASAP: using the ability to directly tune the performance of the algorithm to input-data characteristics (i.e., using circuit propagation delays encode both the algorithm and its computation time), we show a method to chose appropriate propagation delays to compute approximate answers for LD while maintaining their total ordering (i.e., satisfy the application invariant for correctness). Finally, Sections 3.3 and 3.4 describes the ASAP FPGA implementation.

3.1 Encoding LD Computation as Propagation Delays

The core idea is to map addition and minimization, the two mathematical operators necessary for the recursive computation defined in (1), to particular topologies of circuit elements. Fig. 2 illustrates the mapping explained here:

1) If circuit elements are combined in series, the net propagation delay of a signal is the sum of the propagation delays for all of the individual elements. This construction is a proxy for addition.
2) If two circuit elements are connected to an OR gate, the signal that emerges out of the OR gate corresponds to the signal that arrived first at the gate. This construction is a proxy for the minimization operator (in particular, the rising edge of the OR gate's output computes a minimization in time).

TABLE 2
Distribution of Runtime Across the Steps of Algorithm 1 for the SNAP Tool Aligning an In-Sillico Human Genome with $50\times$ Coverage

Lines in Algorithm 1	% of runtime	# of calls
Line 5	6.79	1.5×10^{10}
Line 6	18.59	6×10^{10}
Line 7	59.22	8.3×10^{11}
Line 8	9.25	1×10^{10}
Misc	6.15	–

For example, Fig. 3 demonstrates the computation of "$\min(X+2, X+3)$" using the aforementioned delay based computing. In the example, X corresponds to an arbitrary input signal that is represented in the delay encoding, the 2- and 3-length shift register serving as the delay element implementing the $\cdot + 2$ and $\cdot + 3$ operator respectively, the OR gate serves as the minimization operator and the counter serving as the decoder.

We formalize this delay based computation succinctly in the following lemma.

Lemma 1. *Propagation-delay-based computation can occur on a tropical semiring structure $\mathcal{T}$ over $\{0\} \cup \mathbb{Z}^+$ (i.e., time measured in clock ticks) that defines a binary addition operation, a minimization operator (using an OR gate), and a maximization operator (using an AND gate).*

The delay-based proxies for the addition and minimization operators can be used by replacing the LD values $S(i, j)$ in (1) with the equivalent propagation delays. The resulting circuit represents the application of the addition and minimization operators in the computation of $S(i, j)$. Fig. 4 shows the structure of the circuit that produces this computation. It is composed of a lattice of $l_Q \times l_R$ delay elements (DEs). The connections in the lattice build on the recursive definition of S: each DE $\mathcal{D}(i, j)$'s inputs are connected to the

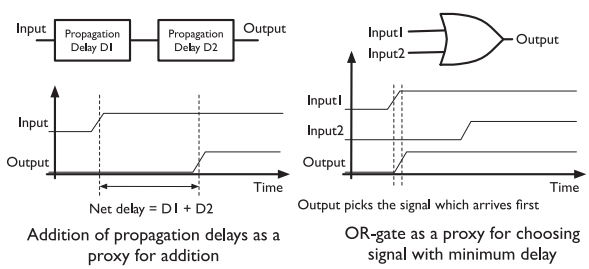


Fig. 2. Computing with propagation delays: Delay-based proxy for the addition operator is a series connection, and the proxy for the $\min$ operator is the OR gate.

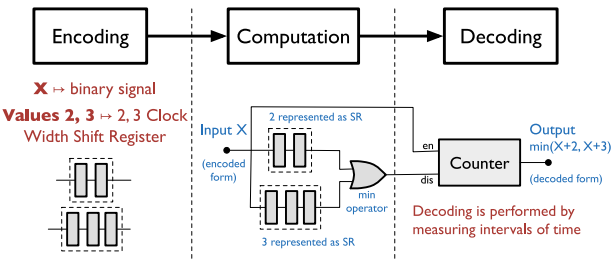


Fig. 3. Example of the encoding, computation, and decoding phase for computing "$\min(X+2, X+3)$" using the circuit-delay proposed in RaceLogic [13]. Note that we present this example using shift-registers for delay elements as opposed to comparators proposed in [13].

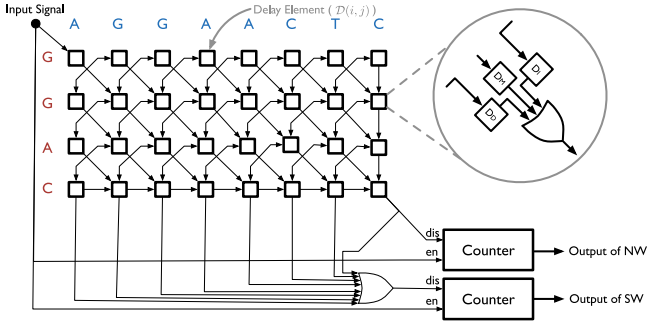


Fig. 4. High-level design of the ASAP accelerator to compute the minimum edit distance between two strings. The accelerator lattice is of size $l_Q \times l_R$, where l_Q and l_R are the sizes of the query and reference, respectively.

outputs of the preceding elements $\mathcal{D}(i-1, j-1)$, $\mathcal{D}(i-1, j)$, and $\mathcal{D}(i, j-1)$, and its outputs are connected to the input of $\mathcal{D}(i+1, j+1)$, $\mathcal{D}(i+1, j)$, and $\mathcal{D}(i, j+1)$. At a high level, each DE is composed of three delay blocks: 1) D_M (delay due to match or mismatch at (i, j)), 2) D_I (delay due to insertion at (i, j)), and 3) D_D (delay due to deletion at (i, j)). This design is specialized for FPGAs in Section 3.3)

The computation can be started by injecting a high signal (logic value 1) at the inputs of index $\mathcal{D}(0, 0)$ in the array. The time-encoded value of the LD is then found by measuring the propagation delay of the signal exiting the array of delay elements. Note that the delay-based computation can be applied to all variants (SW, NW, and LV) of the LD computation as follows.

1) The delay-based version of the *SW* variant can be computed by measuring the delay between the introduction of the input signal in the lattice, and its emergence at any of the delay elements on the last row, i.e., $(l_R, -)$th DE. Fig. 4 illustrates this configuration.
2) The delay-based version of the *NW* variant can be computed by measuring the delay between the introduction of the input signal in the lattice, and its emergence at the (l_R, l_Q)th DE. This configuration is also shown in Fig. 4.
3) The delay-based version of the *LV* variant can be computed by assigning the maximum permissible LD as the result of the computation. This represents the "timeout" with which the signal wavefront will emerge from the DE lattice. If the timeout is triggered, the maximum value of LD, as set by the user, is used as the result of the computation. One delay element and one AND gate (not shown in the Fig. 4) suffice to implement the timeout.

3.2 Approximating LD Computations in ASAP

A key aspect of the aforementioned method is the mapping of gap-penalty parameters (Δ-parameters) to their corresponding circuit delays. The ASAP accelerator uses this mapping both to encode the approximation (mentioned in Section 1), and to reduce the time taken to do the "match"-based computation. Both actions are formally stated below.

Definition 1. *A* Delay Encoding Function $\mathcal{E} : \mathbb{R} \rightarrow \mathcal{T}$ *is a mapping between the set of real numbers and its propagation-delay-based representation. $\mathcal{E}$ is constrained to obey the Cauchy functional equation ($\mathcal{E}(x + y) = \mathcal{E}(x) + \mathcal{E}(y)$).*

More general delay encoding functions can be considered, for example in analog circuits, where circuit elements do not exhibit linear behavior for all inputs. We constrain ourselves to those that satisfy the Cauchy functional equation (CFE) because of simplicity in proving of correctness of the transformation. Although the domain of $\mathcal{E}$ can be the set of real numbers $\mathbb{R}$, the ASAP implementation presented in this paper uses integer or rational gap penalties which can be easily mapped to integer delay values (which can further be represented as a multiples of the clock width).

Definition 2. *A δ-parameter is the time-encoded representation of a user-inputted Δ-parameter. That is*

$$\begin{aligned} \delta(\text{Insert}) &= \mathcal{E}(\Delta(\text{Insert})) \\ \delta(\text{Delete}) &= \mathcal{E}(\Delta(\text{Delete})) \\ \delta(\text{Match}) &= \mathcal{E}(\Delta(\text{Match})) \\ \delta(\text{Mismatch}) &= \mathcal{E}(\Delta(\text{Mismatch})). \end{aligned} \tag{3}$$

These parameters are used to define the delays in the D_M, D_I, and D_D blocks. Note that we have assumed that $\Delta(\text{Match}) = 0$, and thus $\delta(Match) = \mathcal{E}(0)$ is also 0 based on Definition 1.

Based on Definitions 1 and 2, we now show that any encoding of δ-parameters based on $\mathcal{E}$ produces the same ordering of LDs as the original algorithm.

Lemma 2. *When a query string Q and a reference string R are compared under the traditional (see (1)) and delay-based algorithm for computing LD at loci $l_1, \ldots, l_n$ of the reference, to produce LDs $e_1, \ldots, e_n$ and propagation delays $d_1, \ldots, d_n$, respectively, then $d_i = \mathcal{E}(e_i)$, and consequently*

$$e_i \leq e_j \Leftrightarrow \mathcal{E}(e_i) \leq \mathcal{E}(e_j) \Leftrightarrow d_i \leq d_j \; \forall i, j.$$

Lemma 2 is sufficient to show that using the ASAP accelerator to compute LD in the context of Algorithm 1 (in line 7; i.e., using an "$argmin$" operator over the results of multiple executions of the ASAP accelerator) produces the same result as the traditional algorithm (without requiring the computation of the inverse for $\mathcal{E}$). A key observation in the formalism of $\mathcal{E}$ is that the choice of the numerical values of δ can be tuned to directly change the performance of the accelerator, as they corresponds to circuit propagation delays. That is, the parameters and inputs to the accelerator jointly define the net propagation delay of the circuit. Below we demonstrate one such transformation, which forms the core of the approximation used in ASAP.

Lemma 3. *When a query string Q and a reference string R are compared at loci $l_1, \ldots, l_n$ of the reference, they produce LDs $e_1, \ldots, e_n$ for gap penalties Δ, and LDs $e'_1, \ldots, e'_n$ for gap penalties $\Delta + k$, for some number k. The e'_i obey the relationship: $e'_i = e_i + n_i k$, for some $n_i \in \mathbb{Z}$ such that $(n_i \geq 0) \wedge (e_i \leq e_j \Leftrightarrow n_i \leq n_j)$, and consequently*

$$e_i \leq e_j \Leftrightarrow e'_i \leq e'_j \; \forall i, j.$$

Our algorithm for the approximation at the core of ASAP uses Lemmas 2 and 3 to select values of the delay-encoded parameters that correspond to minimizing the time taken to process a dataset. For example, to optimize performance for

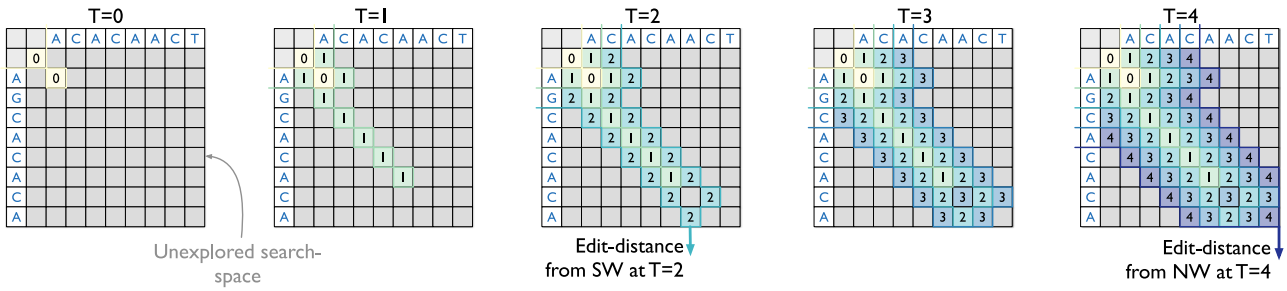


Fig. 5. An example of the ASAP accelerator processing the same inputs used in Fig. 1. The signal wavefront is shown progressing through the ASAP lattice until the outputs of the SW and NW algorithms are produced in 2 and 4 clock cycles, respectively. The values in the matrix represent the clock cycles in which the corresponding DEs were enabled.

our observed case of most nucleotides corresponding to "matches," we modify the gap-penalties to set the match penalty (i.e., $\delta(\text{Match})$) to 0 cycles.[5] This transformation uses a two-step process to convert (encode) user-inputted Δ-parameters into δ-parameters:

1) $\Delta \mapsto \Delta + k$, choosing k so that $\Delta(\text{Match}) = 0$ after the transformation;
2) $\Delta + k \mapsto \mathcal{E}(\Delta + k)$, with $\mathcal{E}(x) = mx$ to produce the required delay value.[6]

As a result, the parameters in the LD algorithm are tweaked to better suit the delay-based computation hardware. The answer (i.e., the exact values of LD) produced by this approximate version of the algorithm is not identical to that produced by the original algorithm. However, based on the aforementioned lemmas, we can see that the total ordering created by the approximated LDs is identical to that of the original algorithm. Furthermore, assuming that most nucleotide comparisons are matches (which is true for the indexed reference-based techniques described in Section 2), this encoding ensures that (almost) zero time is taken to explore large portions of the search space that correspond to matches. We explore the relation of this optimization to timing closure on the FPGA design in Section 3.3. In other re-sequencing experiments, where "matches" do not represent the common computation, a user can set $\delta(k) = 0$ for $k \in \{\text{Insert}, \text{Delete}, \text{Mismatch}\}$. Note that in our formulation of the problem (as described in Section 2), $\Delta(\text{Match})$ is required to be the minimum positive value amongst all the Δ-parameters.

Consider the example of computing the LD between the strings AGCACACA and ACAACAACT, presented in Section 2. Based on our encoding mechanism ($k = 0, m = 1$), we compute the δ-parameters of the ASAP accelerator as $\delta(\text{Match}) = 0$, $\delta(\text{Mismatch}) = 2$, and $\delta(\text{Insert}) = \delta(\text{Delete}) = 1$. Fig. 5 illustrates the propagation of the signal wavefront through the ASAP accelerator for that example. The accelerator produces an output for the SW notion of LD (local alignment) in two clock cycles and the NW notion of LD (global alignment) in four clock cycles. The figure shows the portion of the array explored and the value of the propagation delay at each element $\mathcal{D}(i, j)$ of the lattice. Note that some portions of the array are not explored at all (e.g., for SW and NW, only 25 and 53 DEs out of a total of 81 are triggered, respectively). This design thus provides a large savings in both time (using "zero delay" circuit components for the most commonly used computation) and power (clock-gating unused DEs with their input signals ensures minimal power usage) compared to traditional methods.

To summarize, using the encoding of δ-parameters described in this section, the ASAP accelerator has two clear advantages over traditional techniques:

1) *Faster Processing*: One can explore large portions of the search space in a small amount of time by setting delay parameters appropriately.
2) *Energy Savings*: DEs in the ASAP lattice are used only when their output can contribute to the answer; otherwise, they are switched off to save energy. This can be accomplished by clock-gating the DEs with their input signal.

3.3 ASAP: The FPGA Implementation

3.3.1 Why FPGA?

The techniques discussed so far in the paper represent an approximation technique and architecture, one which can be implemented ASICs, FPGAs, or any other platform. The original RaceLogic design was demonstrated in simulation as an ASIC [13]. However, some key characteristics of the short-read alignment problem and the ASAP architecture make ASAP particularly suitable for FPGAs, as they offer programmability and reconfiguration. The ASAP accelerator is runtime-programmable only for changing the values of gap penalties. The input data size, which defines the size of the accelerator lattice, is fixed at compile time. To allow users to sweep experiment such "meta-parameters" (i.e., input data size, gap-penalty bit-width, and input encoding), ASAP is designed to be re-synthesized and re-programmed on an FPGA. Potentially, the use of partial reconfiguration can allow users to change these parameters on the fly. We leave this possibility for future work. We discuss the advantages of the ASAP design compared to the commonly used systolic array based design (e.g., [30], [31], [32], [33], [34]) in Section 5.

3.3.2 Design of a Delay Element

The overall architecture of the ASAP accelerator is shown in Fig. 4. Fig. 6 shows the design of a single DE. A DE utilizes sequential logic in the form of a shift-register to add a

5. True "0 cycle" propagation delay is not possible because of finite combinational and wire delays in the circuit. Here we imply that the computation is done in combinational logic, whose propagation delay is much much lower than the clock width of the circuit (i.e., 0 time). This is explained further in Section 3.3.

6. The choice of k and m has to ensure that none of the encoded gap penalties are negative. As the encoded values represent circuit propagation delays, negative numbers are meaningless.

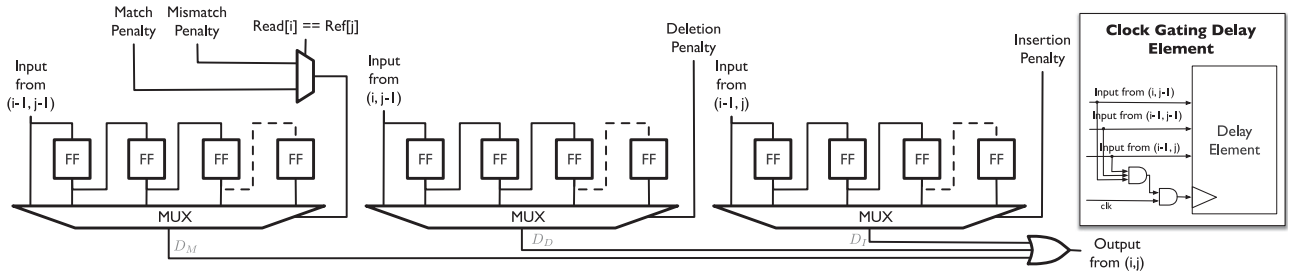


Fig. 6. Design of a single delay element $\mathcal{D}$ in ASAP. The DE is composed of three separate delay units corresponding to D_M, D_I, and D_D in Fig. 4.

user-specified amount of delay. Each DE has 1) three input signals (representing input wavefront) that connect it to its preceding DEs in the grid, 2) two input signals representing the nucleotides being compared by the element, and 3) three input signals representing the δ-parameters. Each DE has one output signal representing the propagated wavefront after the delay has been added. The match, mismatch, insertion, and deletion delay penalties are defined in terms of multiples of the clock period. When the input signal wavefront first reaches an element, it is propagated through a shift register to create delay. Based on the gap penalty specified for match/mismatch, insertion and deletion, the DE propagates the input signals to the output. The output of each flip-flop in the shift register is muxed to allow for the selection of the bit corresponding to the gap-penalty of the block (illustrated in Fig. 6). The ASAP array allows the user to program (i.e., dynamically set at runtime) the values of the select lines of these `MUX`s. This provides the ASAP array with a degree of programmability, allowing it to be reused across computations that merely require re-parameterization of the gap-penalties. Changes in input-sizes, or the dynamic range of the gap penalties (i.e., number of bits required to represent the gap-penalties) requires a re-synthesis and reconfiguration of the accelerator on the FPGA.

As described in the motivating example for the ASAP accelerator given in Fig. 5, the power of the ASAP accelerator is that it can explore a large portion of the search space of possible mappings between the query string and the reference within a clock cycle by setting $\delta(\text{Match}) = 0$. This improvement in computational speed can be coupled with a decrease in energy consumed by the accelerator by clock-gating the DE (illustrated in Fig. 6) with the input signal.

The approach mentioned above has problems with long chains of combinational logic and may lead to timing violations on large lattices of DEs. To get around this problem, larger lattices of delay elements are composed by using the smaller tiles of ASAP accelerators (for which the timing violations do not occur) and by adding a sets of clock-triggered flip-flops between the tiles to break the chains of combinational logic (see Fig. 7). Further, the diagonal tile crossing (i.e., the flip-flops at the lower right corner of the tile) corresponds to a 2 cycle delay (i.e., two flip-flops in serial). Although the additions of the tile flip-flops changes the results of ASAP from what was described in the last section, the overall total-ordering is preserved, as this constitutes a constant addition of delay to all outputs of the ASAP accelerator. Each tile is synthesized, optimized, and placed-and-routed separately by defining separate design partitions. This approach prevents the compiler from performing optimizations across partition boundaries [35]. This approach also ensures that unintended wiring delays do not creep into the netlist of the ASAP accelerator.

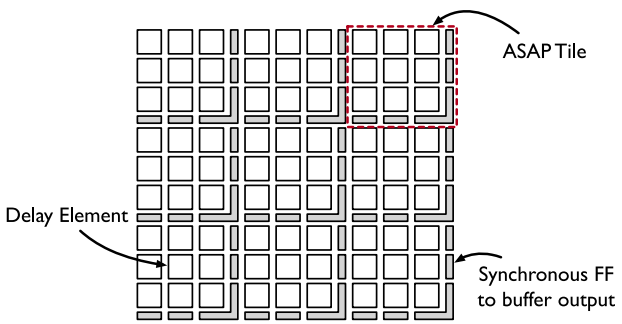


Fig. 7. The architecture of the ASAP accelerator in terms of tiles whose output is buffered by clock synchronous flip-flops (FFs).

The counter that decodes the delayed signal output from the ASAP lattice (shown in Fig. 4) is designed based on a computation of the number of clock cycles for the signal wavefront to emerge from the lattice. The bit-width of this counter, N_o, is calculated from the sizes of the input strings and the user-input gap-penalty parameters, and is given by

$$N_o = \left\lceil \log_2 \min \left\{ \begin{array}{l} \delta_I l_Q + \delta_D l_R, \\ \delta_M l_Q + \delta_D (l_R - l_Q) \end{array} \right\} \right\rceil.$$

This expression is an upper bound (albeit a loose one) on the maximum delay caused by a DE.

3.3.3 Scalability Issues in the ASAP Accelerator

There are challenges involved in scaling the ASAP accelerator to large input sizes and large gap penalties. Those challenges can be addressed as follows:

1) *Large Input Sizes.* The size of the reference and read strings being compared in the ASAP accelerator plays a role in the size of the lattice defined by the ASAP accelerator. The size of the accelerator grows as $O(l_Q \times s)$ with the input size.[7] The tile size parameter defines a tunable knob to control the critical combinational path in the circuit. It can be used to trade off performance against meeting timing closure as the size of the accelerator grows to a significant portion of the resources available on the FPGA. Section 4 demonstrates our scaling experiments with the accelerator.

7. This corresponds to quadratic growth in size of the ASAP lattice (i.e., $O(n^2)$) when $l_Q = s = n$.

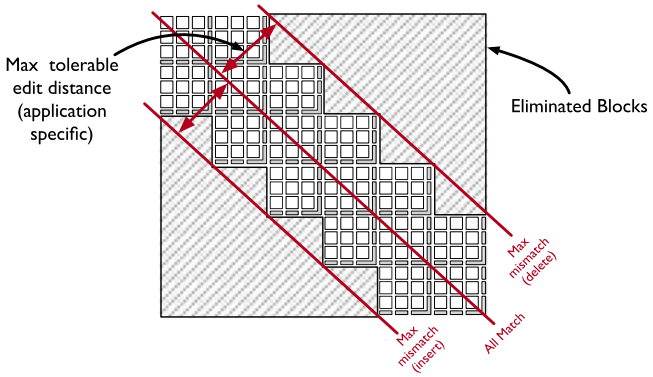


Fig. 8. Elimination of unused tiles from the ASAP lattice in the case of LV variant of the LD algorithm.

2) *Large Gap Penalties.* A large dynamic range of the gap-penalty values negatively affects the ASAP accelerator, as it increases the size of the shift-registers and multiplexers in the DE (see Fig. 6). We work around this problem by using BRAM-based shift registers, which can be $\sim 10^3$ bits long (without intermediate routing). In general, we do not expect large gap penalties to be a problem for genomic sequences (as opposed to protein sequences), for which the dynamic range in gap-penalties is low.
3) *Potentially Unused Tiles.* Fig. 5 shows that a large part of the ASAP array is not involved in computation when the input strings have low LD (which is indeed the case in the short read alignment problem). There are several ways to tackle the problem of unused tiles across the three variants of the LD computation (i.e., SW, NW, and LD). As mentioned earlier, in the case of SW or NW, clock-gating individual delay elements ensures minimal power consumption. Further, in the LV case, as a the worst case LD is specified, we can use this information at compile (in this case synthesis) time to eliminate part of the ASAP lattice that will not contribute to an answer. Fig. 8 illustrates such an elimination on an 18×18 lattice with a maximum of 6 insertions or deletions permitted, resulting in a $56\%(= 20/36 \times 100)$ reduction in area.

3.3.4 Issues with Timing Closure

Computing with propagation delays is disadvantaged by the fact that thermal dissipation and temperature variations at different parts of the FPGA chip to change the physical time associated with unit delay. However, the ASAP accelerator is resilient to these thermal changes up to the maximum operating temperature of the FPGA (i.e., timing violations do not occur). Further, only delays that are multiples of the clock period can affect the computed LD. The tile length serves as a tunable knob between runtime performance and worst case negative slack for the circuit. This slack is enforced by the compiler (e.g., Xilinx Vivado, Altera Quartus) as only values of tile length for which timing closure can be met can be used in the FPGA. Furthermore, the counters in Fig. 4 that measure edit distance are synchronously triggered by the clock, thereby ensuring that all delay-based LDs are computed as multiples of the clock cycle.

3.3.5 Encoding Input Sequences

The implementation of the ASAP accelerator assumed use for genomic data, implying that the entire alphabet can be represented in two bits (i.e., `A`, `C`, `G` and `T`). The bases `N`, `-`, `R`, `Y`, `K`, `M`, `S`, and `W` (which represent an unknown or ambiguous nucleotide) are removed from the alphabet. Our design could potentially be extended to larger alphabets, e.g., for protein sequence alignment.

3.4 Host-to-Accelerator Communication via CAPI

Communication between the host and accelerator is implemented using the CAPI interface [14], [15] provided on an IBM Power8 CPU. The CAPI interface gives an accelerator (a PCIe-attached FPGA) coherent access to the virtual address space of a process running on the host CPU, with all address translations from virtual to physical memory done in the CPU. Fig. 9 shows the interface and mechanism by which the host CPU communicates with the ASAP accelerator. The Power8 is a superscalar symmetric multiprocessor, that has 12 cores per chip, with up to 8 hardware threads per core. All cores have access to shared memory through a PowerBus (shared memory bus). The Coherent Attached Processor Proxy (CAPP) enables the interface (CAPI) by maintaining a directory of cache lines held by the processor and providing coherency by snooping the PowerBus on behalf of the accelerator (or any other PCIe device). The PCIe host bridge provides connectivity between the CAPP and the Power Service Layer (PSL) on the FPGA over the PCIe bus. The PSL on the accelerator acts as a proxy for the CAPI protocol on the FPGA, communicating between the CAPP and the Accelerator Functional Unit (AFU). The AFU contains the custom acceleration logic and reads/writes coherent data across the PCIe. The PSL unit runs at the same speed as the PCIe bus (250 MHz). It contains a memory management unit (MMU) to handle address translation on the accelerator side on its copy of the processor's cache directory.

The AFU interacts with the PSL to provide word-level read and write commands. If these requests are made to cache lines (which are 1,024 bits long) in a shared or exclusive state on the device, they are served locally. Otherwise the PSL interacts with the CAPP over the PCIe bus to attempt virtual to physical address translation, loading of the cache line from main memory (if it is already not present in the processor's cache), moving (or copying of) the cache line to the PSL, and changing the coherence of the cache line in the processor's directory [14], [36]. We use the AFU in dedicated mode, meaning only one MMU context is supported by the accelerator. That is, only one user-space process can use the accelerator at one time.

Fig. 9 shows the configuration of the interface to the PSL for an ASAP accelerator that computes on two 64-bp strings, with each nucleotide encoded by two bits. Hence the accelerator takes 256-bit inputs ($64\text{bp} \times 2\ \text{bits/bp} \times 2$) and produces a propagation delay measurement encoded in 32 bits (to keep with the signed integer implementation in short-read aligner), which is the number of clock cycles for the signal to emerge from the ASAP accelerator (depending on whether the SW or NW algorithm is used). There is an internal 32 kB cache, which has a 1,024-bit input port connected to the PSL, and a 1,024-bit output port that is connected to the input of the ASAP accelerator. This cache is configured in a modified

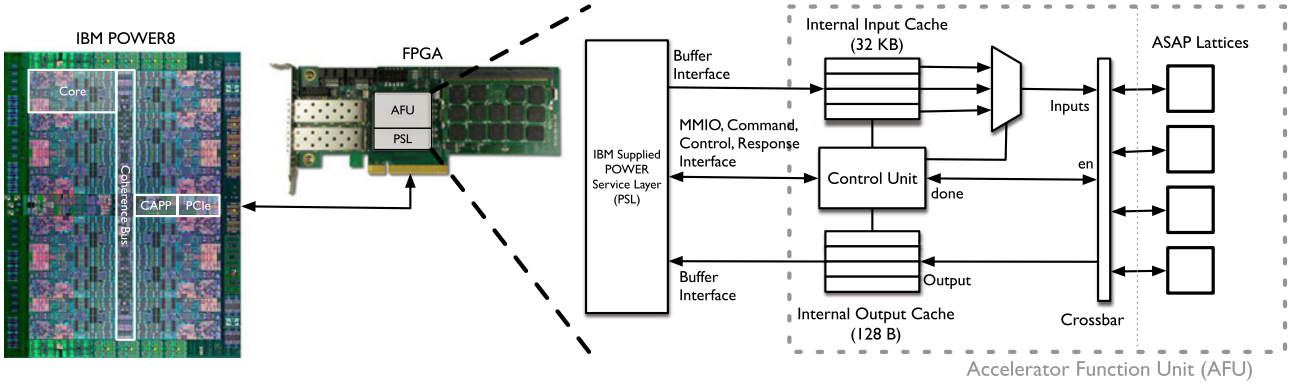


Fig. 9. The design of the interface between the host Power8 processor and the FPGA running the ASAP accelerator using the CAPI interface. The diagram assumes an ASAP accelerator that computes on input strings that are 64 nucleotides long and encoded as 2 bits per nucleotide.

FIFO configuration; each entry in the FIFO contains multiple input cases (in this case, four). A 4×1 MUX controlled by the AFU control unit is responsible for producing 256 bits at a time from the 1,024-bit input. The AFU packs the 32 bit outputs from the ASAP array into 1,024 bit cache-lines before writing them back to the address space of the host over DMA. The AFU uses the work element descriptor (WED; [14]) to communicate the pointer to the input and output, as well as the progress of the accelerator.

4 Evaluation and Discussion

Experimental Setup. The ASAP accelerator is implemented in Chisel [37] and can potentially be compiled across FPGAs by Xilinx and Altera. The host-accelerator interface (which utilizes IBM CAPI) is implemented in VHDL and is specific to an IBM Power8 S824L system with an Alpha-Data ADM-PCIE-7V3 board (that uses a Xilinx Virtex 7 XC7VX690T FPGA) clocked at 250 MHz. All measurements (baseline CPU as well as FPGA-based) were done on this machine. Fig. 10 illustrates the layout of four ASAP lattices and the CAPI based interface on the Virtex 7 FPGA mentioned above.

Input Data & Validation. All inputs for the experiments presented in this section are derived from the human reference genome `hg38` by simulating [28] 100 million reads of appropriate length. The read simulation introduced random mutations and simulated sequencing-error models from an Illumina HiSeq 2,500 with a 0.1 percent sequencing error rate. We verified the correctness of our implementation through comparison with 1) answers generated from the software tools (i.e., BWA [8] or SNAP [9]); 2) the ground truth values generated by the simulator.

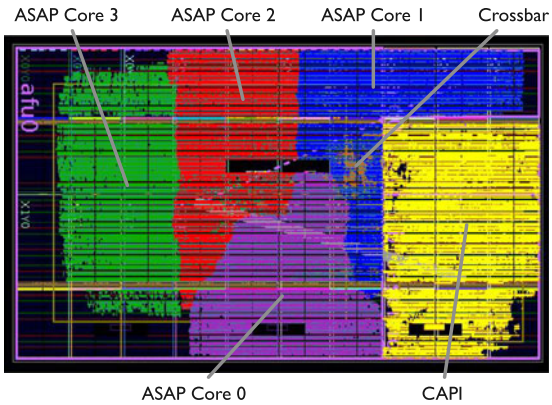


Fig. 10. Layout of the accelerator on the Xilinx Virtex 7 XC7VX690T FPGA. The design implemented above has 4 instances of the ASAP accelerator and the IBM CAPI interface for host-accelerator communications.

The remainder of this section is organized as follows. In Section 4.1 we discuss microbenchmark performance (in terms of runtime, communication bottlenecks, FPGA resource utilization, and energy consumption) of various configurations of the ASAP accelerator. Then in Section 4.2 we discuss the end-to-end performance of integrating the ASAP accelerator into the SNAP [9] and BWA-MEM [8] aligners.

4.1 Microbenchmark Performance

4.1.1 Performance of the Accelerator

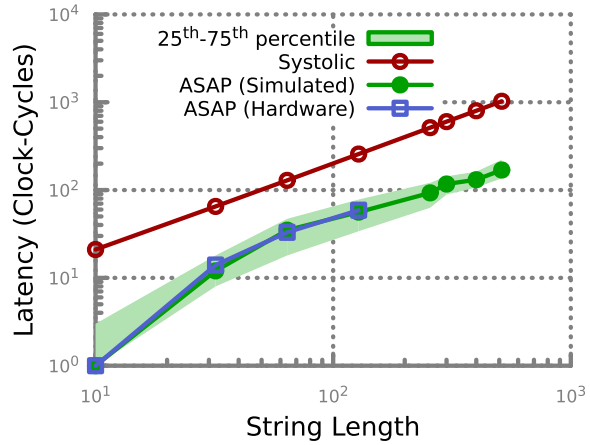
In this section we compare the performance of ASAP-accelerated LD computation against their respective CPU baselines. Here we do not account for time taken to perform disk IO, serialization/de-serialization (i.e., parsing inputs, writing in-memory data structures to disk), and reference lookups (see Section 2) that are required as a part of the end-to-end computation. These other factors are described in Section 4.2.

SW Configuration. The ASAP accelerator is approximately $200\times$ faster than the baseline C implementation of the SW algorithm for computing LD that is optimized to use single instruction multiple data (SIMD; e.g., Intel AVX instructions) and simultaneous multi-threading (SMT; e.g., Intel Hyperthreads) based multi-threading [38]. The baseline implementation exploits inter-task parallelism (i.e., data parallelism) by processing multiple reads across threads. Table 3 describes the comparison of the performance of a single lattice ASAP accelerator. Having multiple cores on

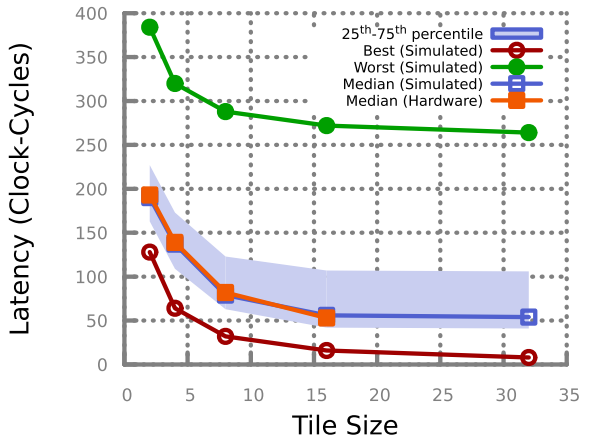
TABLE 3
Comparison of Median Run-Time for LD Computation on CPU and ASAP (SW & LV Configurations)

	SW Configuration			LV Configuration		
Read Size	CPU	ASAP	Speedup	CPU	ASAP	Speedup
64	1,890 μs	10.3 μs	183×	238 μs	6.8 μs	34.8×
128	2,083 μs	10.7 μs	194×	497 μs	10 μs	49.7×
192*	3,326 μs	16.4 μs	203×	729 μs	16.3 μs	44.7×
256*	3,906 μs	17.2 μs	219×	944 μs	17.2 μs	54.9×
320*	4,484 μs	18.9 μs	237×	1,190 μs	18.8 μs	63.3×

Rows marked with "" are simulated results. The LV configuration uses $k = 1/4\times$Read Size.*



(a) Input string length (Tile length = 16).



(b) Tile length (Input length = 128).

Fig. 11. Latency of the ASAP-SW accelerator as a function of the input string length. The shaded area in both the graphs show 25th and 75th percentile measurement from simulation.

the CPU or multiple ASAP lattices on the FPGA does not change this comparison, as each core/lattice is expected to be computing a separate unrelated instance of the LD computation. The performance of ASAP depends not only on the size of the inputs, but on the inputs themselves (i.e., more mismatched inputs mean a higher computation time). Hence we present all ASAP measurements as the median across all the randomly generated reads. We observe that a single ASAP lattice shows $\sim 200\times$ speedup relative to a single CPU core (containing 8 SMT threads and SIMD units), with potential improvements in performance with growing input size (see Table 3). Overall, a Power8 CPU chip contains six such cores, whereas our implementation of ASAP can scale to four lattices (see Fig. 10). Hence a chip-to-chip comparison yields a $133\times$ improvement in performance.

Fig. 11a illustrates the latency of the accelerator (without the overhead of communication between the host and device) in computing LD (in the SW sense) for a single read-reference pair. In contrast to traditional systolic-array-based accelerators, ASAP needs to update only the cells (DEs) that can contribute to the LD computation (i.e., corresponding to the colored cells in Fig. 5). Hence, throughput of the ASAP accelerator can be computed in two ways: we can compute it either by considering the total number of cells in the LD lattice, or by considering only the cells updated by ASAP. The first method which we refer to as *effective-GCUP/s* is directly comparable to traditional techniques as they too consider updating all elements in the LD lattice. In terms of the first method, ASAP achieves an average of 609.6 GCUP/s (10^9 cell updates per second) for 128-bp reads; the second method, it achieves an average of 204.8 GCUP/s. This implies that in the median case, ASAP is approximately $5\times$ better than an equivalent systolic-array-based FPGA implementations (e.g., 122 GCUP/s were physically achieved on an FPGA in [39][8]). Fig. 11b shows the effect of changing tile-length on the latency of the accelerator. It is evident that there are diminishing returns for increasing the tile length, with almost no improvement beyond tile size 16.

LV Configuration. Table 3 shows a comparison of the ASAP accelerator running in the LV configuration to the C++-based LV implementation in the SNAP alignment software [9] (which is multi-threaded and uses SIMD instructions). Overall, LV has a lower computational complexity than SW (i.e., $O(nk)$ compared to $O(n^2)$ for SW). This difference in performance is apparent in the baseline CPU implementations shown in Table 3. Further, we observe that ASAP-accelerated LV is $40-60\times$ faster than the baseline for representative input sizes. This corresponds to a $\sim 4\times$ better performance of the LV configuration compared to the SW configuration of the ASAP accelerator. Note that this difference is input-dependent, with the LV variant performing significantly better (as the maximum delay in the ASAP LV lattice is upper-bounded by k) for string pairs that have larger degrees of mismatches.

Input-Dependence. Another point to note about Fig. 11 is that ASAP represents a method to trade-off worst-case performance and average-case performance. The approximations that we present may be slower than the baseline performance for the worst-case (i.e., when read mismatches reference completely). However, we see that for representative data sets, the median performance as well as the 75th percentile performance are significantly better than the baseline. For the short read alignment problem, we observe that matches occur more frequently than insertions, deletions or mismatches. The ASAP accelerator can also be applied to other cases where insertions or deletions are more frequent by dealing with those cases in combinational logic.

4.1.2 Performance of the CAPI Interface

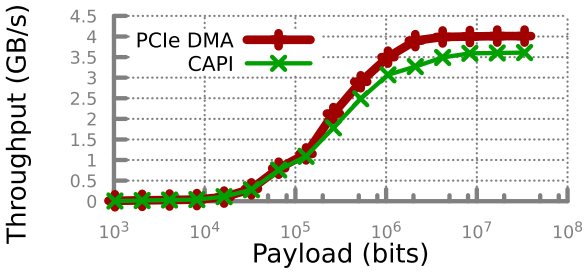
The ASAP accelerator benefits from the use of the CAPI interface, because CAPI 1) significantly simplifies, and 2) significantly streamlines the process of initializing and communicating with the accelerator. We benefit from using a unified virtual memory space across the PCIe bus with hardware-supported address translation, compared with the traditional model, which requires significant hand-holding by an OS. For example, a typical device driver would execute approximately $20k$ instructions, PCIe bounce-buffering, and page-pinning to perform communication between host and accelerator. We performed measurements on the CAPI interface using a loopback accelerator [36] (i.e., an accelerator

8. The comparison to [39] is made based on numbers presented in their paper, and has not been re-implemented by us.

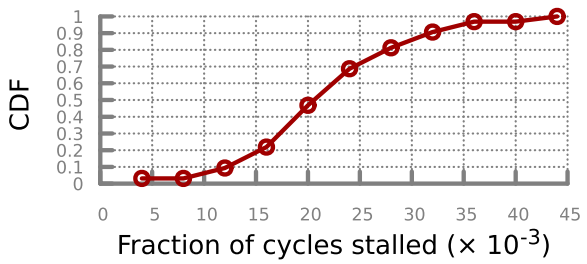
TABLE 4
Measured CAPI-Based Memory Access Performance

Interface	Payload (B)	Type	Measurement
PCIe	128	Mean read/write latency	0.87 μs
CAPI	128	Mean read/write latency	126 ns
CAPI	128	Mean read/write bandwidth	3.88 GB/s

Latency measurements includes round-trip latency to shared memory as seen from the accelerator.



(a) Mean observed bandwidth.

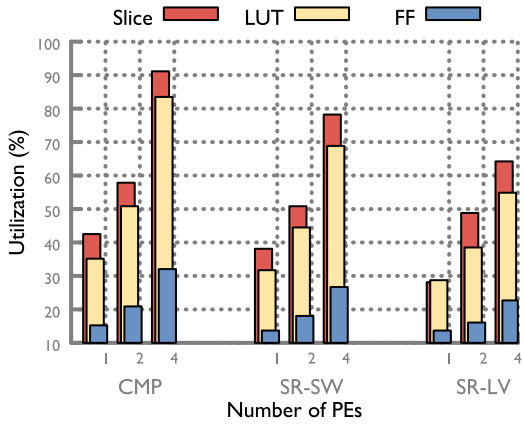


(b) Fraction of cycles stalled due to unavailability of data.

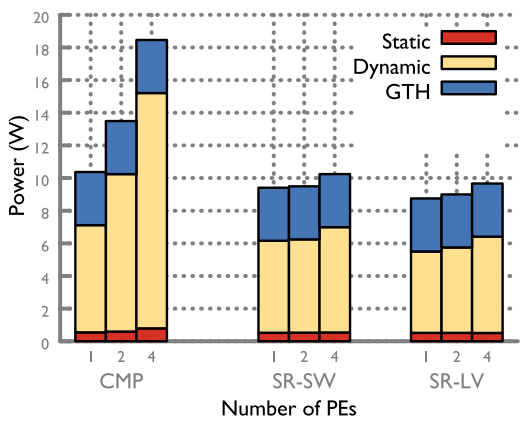
Fig. 12. Mean host-accelerator bandwidth over the CAPI interface and its effect on the performance of the ASAP accelerator.

reads a cache-line and writes it back to a different location). We observed that (see Table 4 and Fig. 12a) the CAPI interface can perform random reads and writes with 1) sub-μs latency, and 2) 4 GB/s bandwidth which are both close to the measured native PCIe latency/bandwidth for the FPGA board used in the evaluation. The one disadvantage that we observe with the CAPI interface is that it allows an AFU to use only 50 percent of the available peak-theoretical PCIe bandwidth. Our measurements of PCIe goodput (i.e., bandwidth for user data to and from the accelerator) are similar to those from CAPI (see Fig. 12a).[9] Bandwidth is currently not a limitation for the accelerator. Fig. 12b shows the fraction of the runtime of the accelerator spent in stall over the execution of a large number of reads. However, moving to a larger FPGA that supports larger ASAP lattices or multiple smaller ASAP lattices (executing in parallel), or clocking the ASAP accelerator higher than 250 MHz will require larger bandwidth for the host-accelerator interface.

9. We speculate that this limitation occurs because of non-optimal interactions between the OS-modules (e.g., CAPI cache misses trigger TLB/ERAT or page misses) and the PCIe-endpoint ASIC (e.g., dealing with out-of-order packet delivery) on the FPGA board. We leave the optimization of such direct memory access (DMA) issues to future work.



(a) Scaling of FPGA resource utilization (accelerator size) with increase in number of ASAP lattices.



(b) Power dissipation from the ASAP accelerator with increase in number of ASAP lattices per chip.

Fig. 13. Comparison of on-chip resource utilization of the `CMP` and `SR` implementations of the ASAP design. Each ASAP lattice is of size 128×128.

4.1.3 FPGA Resource Utilization

This section describes the overall on-chip resource utilization to implement the CAPI interface and multiple ASAP lattices on the FPGA. Fig. 13 illustrates this utilization with the increasing number of lattices for two implementation styles for the ASAP delay element. First, the comparator based design that was presented in the original RaceLogic paper [13] (referred to as `CMP` in the figure), and second, the shift-register based design (presented in Section 3) that has been optimized for FPGAs (referred to as `SR-SW` and `SR-LV` in the figure). Fig. 13a demonstrates the significant reduction (nearly 15 percent) in number of logic elements (i.e., slice resources) required to implement `SR` compared to `CMP`. This further translates to a $\sim 1.9\times$ reduction in power consumed by the `SR` design (shown in Fig. 13b). In the SW configuration, the proposed design is nearly $18.8\times$ more power efficient than the IBM Power8 CPU (~ 10.1 W compared to 190 W). This implies an overall $3,760\times$ ($= 200 \times 18.8$; based on Section 4.1.1) improvement over the CPU in performance/Watt terms. The LV configuration of the accelerator utilizes 19 percent less FPGA resources and consumes 10 percent less power than the SW configuration. The diminished returns from the LV optimizations are a result of the IBM PSL module's occupying $\sim 30\%$ of the

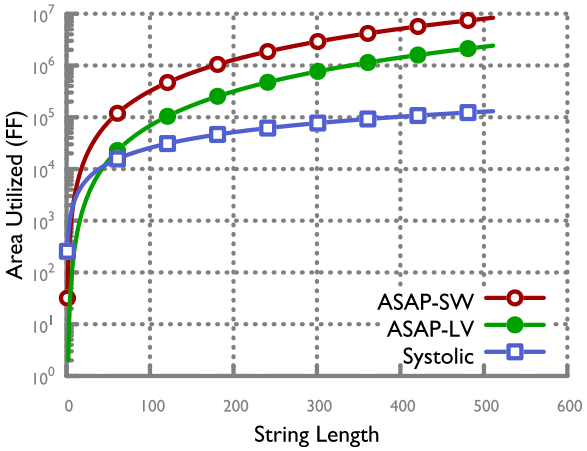


Fig. 14. Scaling of FPGA resource utilization (accelerator size) with increase in input string size for the ASAP lattice in SW configuration.

FPGA area, thereby dominating the relative decrease in resource utilization and power.

Note that the power consumption for the chip is calculated from the synthesis tool (i.e., Xilinx Vivado) and represents worst-case power consumed by the accelerator. However, the real power consumption is input-dependent and lower than that mentioned above, as clock-gating on off-diagonal delay elements will be enabled differently based on inputs (recall Fig. 5). We computed this difference in power consumption using the S824L's on-board power meters on the Flexible Service Processor (FSP).[10] The FSP measurements report power consumption of the entire computer system averaged over 30 s intervals. To calculate the power consumption of the ASAP accelerator, we measured the difference in power consumed by the system when executing the 4-lattice instance of the ASAP accelerator shown in Fig. 10, and when idling. We observed an average difference (i.e., the ASAP accelerator's average power consumption) over 100 executions (of the entire benchmark dataset) of 6.9 W $\pm$ 2.8 W (error is expressed as standard deviation) for the SW configuration and 6.8 W $\pm$ 1.6 W for the LV configuration. These measurements support our claim that the actual power consumption of ASAP is lower than that reported by the synthesis tool.

Area-Based Scaling. The resource utilization of the ASAP accelerator scales quadratically with the lengths of the sequences being compared. For example, Fig. 14 shows the number of flip-flops (including those used in shift registers) used by the ASAP accelerator with increasing string length, based on a 16×16 square tile size.[11] In comparison, an FPGA-based systolic array implementation of the LD computation [30] (described in Section 5) scales linearly (i.e., $2N + 1$, where N is the length of the strings being compared). It is apparent that for larger sequences, ASAP quickly exhausts the FPGA resources.

However, ASAP is able to compute LD for short-read sequences (e.g., the 100-150 bp sequences that are typically obtained from an Illumina HiSeq 2500) which are popularly used in resequencing experiments. In addition, we leave approximately 20 percent of the area of the FPGA free, to allow the CAD tools to place-and-route the circuit without timing violations due to wiring delays.[12] As a result, we are able to fit a maximum 128 bp read accelerator on our FPGA. Fitting larger blocks leads to timing violations because of delays introduced by the on-chip interconnect. Given the industry trend towards FPGAs with larger programmable area, in the future it should be possible to extend ASAP to read sizes that are potentially thousands of nucleotides long.

Handling Inputs Larger Than Lattice Size. Currently, the ASAP accelerator can be used to compute LD for larger strings by adding a special software-based control algorithm in software to compute LD between sub-strings of the original queries, and combine them to compute the result. The algorithm works by measuring (and storing) the time at which the signal wavefront leaves the extremal DEs of the ASAP lattice, and reintroducing this signal wavefront in the same lattice after updating the nucleotides to be another disjoint substring of the queries. We leave the hardware implementation of this approach for future work.

NW Configuration. Note that the NW and SW configurations of the ASAP accelerator are identical in terms of performance and FPGA-resource utilization. Section 3.1 describes how the NW and SW configurations differ in how delay between the input and output are measured.

4.2 Integration Into End-to-End Alignment Software

In this section, we compare ASAP-accelerated versions of end-to-end alignment software tools with their baseline CPU versions. This comparison includes time taken for LD computation as well as other auxiliary functions like, disk IO, and marshaling and un-marshaling of data (from disk and from accelerator). As a result, improvements in LD computation provide diminishing returns (i.e., asymptotic behavior similar to Amdahl's law); we show that the current ASAP represents a speedup that is very close to the asymptotic limits for this computation. We use two alignment tools, SNAP [9] (which uses ASAP in the LV configuration) and BWA-MEM [8] (which uses ASAP in the SW configuration).[13] These results are described below.

Host-Accelerator Communication. The baseline SNAP & BWA aligners exploit parallelism in the alignment problem by dividing the work of aligning a set of reads among all of the 192 threads available on the system. We use the same communication algorithm to dispatch LD computations to the accelerator in both cases. Since our current implementation of ASAP allows for only one calling context on the host-side, accelerator executions are dispatched by maintaining a pool of memory shared among all threads to communicate with the accelerator. The procedure for each thread communicating with the accelerator is as follows: 1) picks a read from the set it was assigned; 2) queries the

10. The FSP is an auxiliary processor on the S824L that is an always-on management processor enabling out-of-band management of the server.

11. This example does not include flip-flops required for the CAPI interface.

12. There is no simple analytical method to derive the optimal tile size, sequence size and free area on the FPGA, as the synthesis tools are a black box.

13. We used version 1.0 of the SNAP tool and version 0.7.17 for BWA.

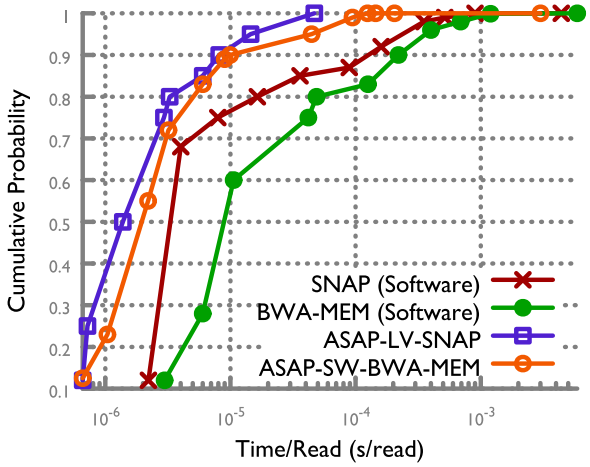


Fig. 15. Performance comparison of the ASAP-accelerated SNAP and BWA-MEM alignment tools (called `ASAP-LV-SNAP` and `ASAP-SW-BWA-MEM`, respectively) with their baseline CPU versions (called `SNAP (Software)` and `BWA-MEM (Software)`, respectively).

reference index for candidate locations for the read; 3) contends for a lock, then writes nucleotides for the read and the candidate locations into shared memory; 4) at this point, the accelerator reads from the shared memory and writes out the results to another shared segment of memory; and 5) polls for results from the accelerator using a test and test-and-set based locking protocol [40], then consumes the output. This algorithm exemplifies CAPI's benefit, as we can make use of cache coherence between the CPUs and FPGA to easily implement mutual exclusion.

The SNAP & BWA-MEM Aligners. Fig. 15 shows the distribution of time taken per read by the baseline and the ASAP accelerator for all LD computations. We see that there is a large spread for total time spent in computing LD because some reads map to more regions of the reference than others. This variation is an artifact of both the nature of the human genome and the read simulator's practice of picking reads at random from the genome. We observe that the SNAP aligner is accelerated by $2\times$ (i.e., $^{1.85\text{hr}}/_{0.92\text{hr}}$) and that the BWA-MEM aligner is accelerated by $1.86\times$ (i.e., $^{2.64\text{hr}}/_{1.42\text{hr}}$), respectively. These results are representative of the mean time spent in processing a single read. Fig. 15 shows the long-tailed behavior of some of the input pairs in the datasets that are significantly mismatched (for the accelerated implementations). The long-tailed behavior of the CPU baselines stem from non-determinism in the IO and thread scheduling subsystems of the host. The LV upper-bound for maximum time spent in computing LD ensures that the ASAP-accelerated SNAP version (called `ASAP-LV-SNAP` in Fig. 15) has a significantly shorter tail than the SW configuration in BWA (called `ASAP-SW-BWA-MEM` in Fig. 15).

The performance measurements presented above are close to the Amdahl's law limit of the SNAP algorithm based on our measurements presented in Table 2. In the case of the BWA-MEM aligner, we observe that we achieve a lower absolute improvement, that is expected, as the asymptotic limit for improvement is lower. The BWA-MEM algorithm performs larger amounts of non-LD computation compared to SNAP, i.e., Burrows-Wheeler transform based index lookup while SNAP computes hashes for a hash table lookup. Both baselines are measured with huge-page support turned on in the host's Linux-kernel to negate effects of ERAT-misses (TLB-misses in Intel parlance).

Gap Penalty Models. A subtlety to be noted in the comparison presented above is that BWA-MEM's default behavior uses *affine gap-penalties* in addition to the SW local-alignment algorithm (instead of the *constant gap-penalties* used by ASAP). Hence we have to use the tool's command line arguments to set the gap-penalty parameters such that they replicate a constant gap-penalty model (i.e., set the requisite parameters to 0). We discuss handling of affine gap-penalties in ASAP as part of our future work in Section 6.

5 Related Work

The sequence alignment problem has been addressed by an extensive body of work that looks at algorithms and their high-performant implementations on CPUs and on accelerators like GPUs and FPGAs. This section focuses on comparing ASAP to other implementations of the LD computations. Refer to Section 2 for a discussion of algorithms.

On CPUs and GPUs. The LD computation and sequence-alignment problem has been studied on SIMD and MIMD processors that exploit parallelism in the problem at two levels. Inter-task parallelism [41] (using multiple cores to independently compute alignments of different short reads), and intra-task parallelism [42], [43] (using SIMD instructions and efficient use of the memory hierarchy to effectively compute (1)). Most of the popular SW or NW implementations exploit the use of both of these techniques. These techniques have also been applied to GPUs [44], [45], [46]. One such example is NVIDIA's NVBIO [47] library and the accompanying set of tools nvBWT, nvFM-server. These look at accelerating the construction and look-up of data structures that index the reference genome. The major disadvantages of this approach is the large power consumption of these processors, and their restrictive lock-step parallelism based programming models.

On FPGAs and ASICs. Custom hardware acceleration of the problem on FPGAs and ASICs has also been widely studied. Most of the popular hardware architectures are based on systolic arrays [30], [31], [32], [33], [34]. These architectures like the SIMD and MIMD approaches, are limited by the amount of parallelism they can exploit. It has been shown in [48], that exploiting deeper pipelines with much larger inter-task parallelism can potentially enable more efficient use of FPGAs. We may be able to use this optimization to futher increase the throughput of the accelerator, particularly on larger FPGAs that can sustain larger off-chip bandwidth. Kaplan et al. [49] present an ASIC design for a Processing-in-Memory accelerator for the SW algorithm that leverages resistive content-addressable memory to compute matches/mismatches of nucleotides. ASAP represents a significant improvement over [49] in throughput/Watt terms, i.e., ASAP achieves 61 GCUP/s/W ($=^{609.6}/_{10.1}$) compared to their 53 GCUP/s/W. Turakhia et al. [50] present an accelerator to perform long-read assembly, one step of which includes a SW-based alignment (through a seed-and-extend approach). Alser et al. [51] present an FPGA based accelerator to efficiently filter candidate locations to calculate LD. This accelerator is targeted at Line 6 of Algorithm 1, as opposed to ASAP which targets

Line 7, hence the accelerator can be used in addition to ASAP to accelerate the end-to-end alignment process. More recent work [52] has also shown the benefit of distributing the compute intensive LD computation across multiple accelerators (including CPUs, GPUs, FPGAs, Xeon Phis). We observe that ASAP significantly outperforms such multi-accelerator systems both in terms of performance and performance per-Watt. The Host + $2\times$ FPGA design presented in [52] only achieves a 441.6 GCUP/s performance at 1.51 GCUP/s/W. In comparison ASAP achieves 609.6 effective GCUP/s at 61 GCUP/s/W on a single FPGA.[14] Other work, e.g., [53], [54], [55], has demonstrated the use of systolic-array-based designs to accelerate computations on Pair-HMM models, where gap-penalties are replaced by probability distributions. That may be a future direction for the extension of the ASAP design.

ASAP's design philosophy is most closely related to Madhavan et. al.'s RaceLogic [13] ASIC design, which also encodes LD computations as circuit delay. However, ASAP builds on this basic model to further optimize the design by using 1) approximation algorithms for the LD computation which maintains the total ordering of LDs, and 2) accelerating the most common computation (in this case the processing of "matches") in combinational circuitry thereby spending minimal runtime in its computation. This is demonstrated by the fact that ASAP is $\sim 50\times$ faster than a RaceLogic implementation. Further, the nature of the alignment problem and the rapidly evolving sequencing technology (i.e., read lengths), implies that fixed function ASICs are not favorable because of the large monetary investment required and the inability of the accelerator to adapt to new input sizes. ASAP circumvents these problems by using reconfigurable FPGAs. Of course, an ASIC will almost always outperform an FPGA in energy efficiency because of its customized layout. Hence going forward, a design with a fixed function (i.e., ASIC-based) IO interface (i.e., CAPI) with a configurable substrate for ASAP accelerators might present an ideal trade-off.

Comparison to Systolic Arrays. Relative to the related work described above, ASAP has some decided advantages:

1) The systolic array based approaches require each element of the array to compute on as many bits as the maximum LD computed. Our approach requires only as many bits per delay element as the maximum delay between inputs at that point in the lattice.
2) The earlier accelerators have to explore the entirety of the lattice before computing the LD. We show that the ASAP accelerator explores only the portions of the lattice that is reachable before the final result is produced. This represents a significant savings in run time and energy expended for computation.
3) The ASAP accelerator can explore multiple elements in the lattice in under one clock cycle by setting $\delta(\text{Match}) = 0$. Systolic array based architectures cannot perform this optimization, as this creates large combinational chains which make timing closure difficult to obtain.

14. The comparison is made across an equivalent generation of Altera and Xilinx FPGAs, using *effective-GCUP/s* (described in Section 4.1.1).

On Neuromorphic Computers. Neuromorphic computing is modeled on biological neurons that communicate and compute using temporal-encoding of information as voltage pulses, or spikes. This is similar in principle to the delay based computation outlined in this paper, however it is still an open research question [56], [57].

On Sequencing Technologies. Recall that the alignment computation (shown in Section 2) is composed of the LD computation as well heuristics to identify candidate reference regions. The use of novel sequencing technologies (e.g., PacBio, Nanopore which are based on single-molecule sequencing), introduces new sequencing error regimes which will change the heuristic components of the alignment computation, but not the LD. As ASAP targets the LD computation, we believe it can be applied to data generated from these long-read sequencing machines. Turakhia et al. [50] present one such accelerator for long reads. Their accelerator targets the acceleration of the entirety of Algorithm 1 and uses LD computation as a submodule. ASAP can replace that module and provide significant performance and energy benefits as shown in this paper.

6 Conclusion and Future Work

This paper proposed ASAP, an accelerator for rapid computation of LD, in the context of the short-read alignment problem. ASAP builds upon the idea that the LD between strings can be approximated for the short-read alignment problem by encoding gap penalties in propagation delays of circuit elements. We show that by effectively setting these delays, it is possible to accelerate performance significantly, and at the same time ensure that the accuracy of alignment is maintained. ASAP significantly outperforms (both in performance and performance-per-Watt terms) purely CPU/GPU-based as well as sytolic array-based accelerator implementations of LD computation in the all the SW, LV and NW configurations.

The ASAP accelerator, and the approach (based on heuristic approximations) presented in this paper, can also be adapted to a variety of other problems in which a total ordering of LDs is computed. For example, in signal processing, where different instances of a signal have to be aligned to compute similarity [3]; in text retrieval, where misspelled words have to be accounted for in a dictionary [16]; and in virus- and intrusion-detection, where signatures have to be aligned to a baseline [17].

Future Work. Our future work will primarily look to extend ASAP to handle more complex gap-penalty models. This paper describes the use of constant gap penalties (i.e., a fixed score is assigned to every gap), which are commonly used in DNA alignment (e.g., in NCBI-BLASTN, or WU-BLASTN [20]). We can extend ASAP to handle linear, affine, and convex gap penalties by letting each DE track the propagation of the signal wavefront in the portion of the lattice before it. We can do so by dynamically resizing the length of the shift registers on off-diagonal DEs depending on their positions (i.e., i, j coordinates). Further, re-using the lattice for input strings larger than the lattice dimensions would involve dynamic reconfiguration of the FPGA to allow for different *taps* in the shift registers. Further, ASAP can also be extended for use in the alignment of proteins by using

substitution matrices, like BLOSUM [5], which assign unique scores to each pair of residues.

Acknowledgments

This research was supported by several grants: in part by the US National Science Foundation (NSF) under Grant Nos. CNS 13-37732 and CNS 16-24790; in part by the Blue Waters sustained-petascale computing project supported by the US National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois; and in part by IBM Faculty Awards. We thank Zachary Stephens, Jenny Applequist and Kathleen Atchley for their help in preparing the manuscript.

References

[1] 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 Biol.*, vol. 13, no. 7, Jul. 2015, Art. no. e1002195.

[2] Y. S. Shao and D. Brooks, "Research infrastructures for hardware accelerators," *Synthesis Lectures Comput. Archit.*, vol. 10, no. 4, pp. 1–99, 2015.

[3] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, p. 707, Feb. 1966, http://adsabs.harvard.edu/abs/1966SPhD...10..707L

[4] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surveys*, vol. 33, no. 1, pp. 31–88, Mar. 2001.

[5] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *J. Mol. Biol.*, vol. 215, no. 3, pp. 403–410, Oct. 1990.

[6] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biol.*, vol. 10, no. 3, 2009, Art. no. R25.

[7] B. Langmead and S. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, pp. 357–359, Mar. 2012.

[8] H. Li and R. Durbin, "Fast and accurate long-read alignment with burrows-wheeler transform," *Bioinf.*, vol. 26, no. 5, pp. 589–595, Jan. 2010.

[9] 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:1111.5572*, 2011, https://arxiv.org/abs/1111.5572

[10] S. S. Banerjee, A. P. Athreya, L. S. Mainzer, C. V. Jongeneel, W.-M. Hwu, Z. T. Kalbarczyk, and R. K. Iyer, "Efficient and scalable workflows for genomic analyses," in *Proc. ACM Int. Workshop Data-Intensive Distrib. Comput.*, 2016, pp. 27–36.

[11] T. C. Glenn, "Field guide to next-generation DNA sequencers," *Mol. Ecology Resources*, vol. 11, no. 5, pp. 759–769, May 2011.

[12] M. G. Ross, C. Russ, M. Costello, A. Hollinger, N. J. Lennon, R. Hegarty, C. Nusbaum, and D. B. Jaffe, "Characterizing and measuring bias in sequence data," *Genome Biol.*, vol. 14, no. 5, 2013, Art. no. R51.

[13] A. Madhavan, T. Sherwood, and D. Strukov, "Race logic: A hardware acceleration for dynamic programming algorithms," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 517–528, Jun. 2014.

[14] I. C. Systems and T. Group. Coherent accelerator processor interface: User's manual. 2015. [Online]. Available: http://www.nallatech.com/wp-content/uploads/IBM_CAPI_Users_Guide_1–2.pdf

[15] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, "CAPI: A coherent accelerator processor interface," *IBM J. Res. Develop.*, vol. 59, no. 1, pp. 7:1–7:7, Jan. 2015.

[16] R. Baeza-Yates and B. Ribeiro-Neto, et al., "Modern information retrieval," vol. 463, 1999, http://people.ischool.berkeley.edu/~hearst/irbook/

[17] S. Kumar and E. H. Spafford, "A pattern matching model for misuse intrusion detection," in *Proc. 17th Nat. Comput. Secur. Conf.*, 1994, pp. 11–21.

[18] T. Smith and M. Waterman, "Identification of common molecular subsequences," *J. Mol. Biol.*, vol. 147, no. 1, pp. 195–197, 1981.

[19] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–453, Mar. 1970.

[20] W.-K. Sung, *Algorithms in Bioinformatics: A Practical Introduction (Chapman & Hall/CRC Mathematical and Computational Biology)*. London, U.K.: Chapman and Hall/CRC, 2009.

[21] S. Henikoff and J. G. Henikoff, "Amino acid substitution matrices from protein blocks," *Proc. Nat. Academy Sci. United States America*, vol. 89, no. 22, pp. 10 915–10 919, Nov. 1992.

[22] C. Wang, R.-X. Yan, X.-F. Wang, J.-N. Si, and Z. Zhang, "Comparison of linear gap penalties and profile-based variable gap penalties in profile–profile alignments," *Comput. Biol. Chemistry*, vol. 35, no. 5, pp. 308–318, Oct. 2011.

[23] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," 1994, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.121.6177

[24] E. Ukkonen, "Algorithms for approximate string matching," *Inf. Control*, vol. 64, no. 1, pp. 100–118, 1985.

[25] G. M. Landau and U. Vishkin, "Efficient string matching with k mismatches," *Theoretical Comput. Sci.*, vol. 43, pp. 239–249, 1986.

[26] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proc. 41st Annu. Symp. Found. Comput. Sci.*, 2000, Art. no. 390.

[27] Illumina. Pair-end sequencing. 2010. [Online]. Available: http://www.illumina.com/technology/next-generation-sequencing/paired-end-sequencing_assay.html

[28] Z. D. Stephens, M. E. Hudson, L. S. Mainzer, M. Taschuk, M. R. Weber, and R. K. Iyer, "Simulating next-generation sequencing datasets from empirical mutation and sequencing models," *PLoS One*, vol. 11, no. 11, Nov. 2016, Art. no. e0167047.

[29] N. C. for Supercomputing Applications (NCSA). Blue waters supercomputer. 2012. [Online]. Available: https://bluewaters.ncsa.illinois.edu/

[30] R. J. Lipton and D. Lopresti, "A systolic array for rapid string comparison," in *Proc. Chapel Hill Conf. VLSI*, 1985, pp. 363–376.

[31] D. T. Hoang and D. P. Lopresti, "FPGA implementation of systolic sequence alignment," in *Proc. Int. Workshop Field Programmable Logic Appl.*, 1993, pp. 183–191.

[32] S. A. Guccione and K. Eric, "Gene matching using JBits," in *Proc. Reconfigurable Comput. Going Mainstream 12th Int. Conf. Field-Programmable Logic Appl.*, 2002, pp. 1168–1171.

[33] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform," in *Proc. 1st Int. Workshop High-Perform. Reconfigurable Comput. Technol. Appl. Held Conjunction SC07*, 2007, pp. 39–48.

[34] N. Ahmed, V. M. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars, "Heterogeneous hardware/software acceleration of the BWA-MEM DNA alignment algorithm," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, Nov. 2015, pp. 240–246.

[35] Xilinx. Hierarchical design methodology guide. 2010. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/Hierarchical_Design_Methodology_Guide.pdf

[36] M. J. Jaspers, "Acceleration of read alignment with coherent attached FPGA coprocessors," Master's thesis, Microelectron. Comput. Eng., Delft Univ. Technol., The Netherlands, 2015.

[37] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *Proc. Des. Autom. Conf.*, Jun. 2012, pp. 1212–1221.

[38] J. Daily, "Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments," *BMC Bioinf.*, vol. 17, no. 1, Feb. 2016, Art. no. 81.

[39] A. Sirasao, E. Delaye, R. Sunkavalli, and S. Neuendorffer, "FPGA based OpenCL acceleration of Genome sequencing software," in *Poster Presented at Supercomputing*, Austin, TX, USA, Nov. 2015. [Online]. Available: http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/tech_poster_pages/post269.html

[40] G. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*. Reading, MA, USA: Addison-Wesley, 2000.

[41] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "merAligner: A fully parallel sequence aligner," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2015, pp. 561–570.

[42] M. Farrar, "Striped smith-waterman speeds database searches six times over other SIMD implementations," *Bioinf.*, vol. 23, no. 2, pp. 156–161, Nov. 2006.

[43] R. Hughey, "Parallel hardware for sequence comparison and alignment," *Comput. Appl. Biosci.*, vol. 12, no. 6, pp. 473–479, Dec. 1996.

[44] Y. Liu, W. Huang, J. Johnson, and S. Vaidya, "GPU accelerated Smith-Waterman," in *Proc. 6th Int. Conf. Comput. Sci.*, 2006, pp. 188–195.
[45] Y. Liu, B. Schmidt, and D. L. Maskell, "CUSHAW: A CUDA compatible short read aligner to large genomes based on the burrows-wheeler transform," *Bioinf.*, vol. 28, no. 14, pp. 1830–1837, May 2012.
[46] K. Zhao and X. Chu, "G-BLASTN: Accelerating nucleotide alignment by graphics processors," *Bioinf.*, vol. 30, no. 10, pp. 1384–1391, Jan. 2014.
[47] N. Corporation. Nvbio. 2015. [Online]. Available: https://developer.nvidia.com/nvbio
[48] Y. T. Chen, J. Cong, J. Lei, and P. Wei, "A novel high-throughput acceleration engine for read alignment," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Programmable Custom Comput. Machines*, May 2015, pp. 199–202.
[49] R. Kaplan, L. Yavits, R. Ginosar, and U. Weiser, "A resistive CAM processing-in-storage architecture for DNA sequence alignment," *IEEE Micro*, vol. 37, no. 4, pp. 20–28, 2017.
[50] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A Genomics co-processor provides up to 15,000x acceleration on long read assembly," in *Proc. 23rd Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2018, pp. 199–213.
[51] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "GateKeeper: A new hardware architecture for accelerating pre-alignment in DNA short read mapping," *Bioinf.*, vol. 33, no. 21, pp. 3355–3363, May 2017.
[52] E. Rucci, C. Garcia, G. Botella, A. E. D. Giusti, M. Naiouf, and M. Prieto-Matias, "OSWALD: OpenCL Smith–Waterman on Altera's FPGA for large protein databases," *Int. J. High Perform. Comput. Appl.*, vol. 32, no. 3, pp. 337–350, 2018.
[53] J. Peltenburg, S. Ren, and Z. Al-Ars, "Maximizing systolic array efficiency to accelerate the PairHMM forward algorithm," in *Proc. IEEE Int. Conf. Bioinf. Biomed.*, 2016, pp. 758–762.
[54] S. S. Banerjee, M. El Hadedy, C. Y. Tan, Z. T. Kalbarczyk, S. Lumetta, and R. K. Iyer, "On accelerating pair-HMM computations in programmable hardware," in *Proc. 27th Int. Conf. Field Programmable Logic Appl.*, Sep. 2017, pp. 1–8.
[55] S. Huang, G. J. Manikandan, A. Ramachandran, K. Rupnow, W. Mei, W. Hwu, and D. Chen, "Hardware acceleration of the pair-HMM algorithm for DNA variant calling," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 275–284.
[56] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Sci.*, vol. 345, no. 6197, pp. 668–673, Aug. 2014.
[57] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J.-M. Bussat, R. Alvarez-Icaza , J. V. Arthur, P. A. Merolla, and K. Boahen, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proc. IEEE*, vol. 102, no. 5, pp. 699–716, May 2014.

Subho Sankar Banerjee received the BTech degree in computer science and engineering from LNMIIT, India. He is working toward the PhD degree in computer science at the University of Illinois at Urbana-Champaign. His research focuses on the design and implementation of workload optimized computing systems (using hardware accelerator and parallel runtime environments) for data analytics workloads.

Mohamed El-Hadedy received the BSc and MSc degrees from the Mansoura University, Egypt, in 2002 and 2006, respectively, and the PhD degree in electrical and computer engineering from the Telematics Department, Norwegian University of Science and Technology, Trondheim, Norway, in 2012. He is a research scientist with the University of Illinois at Urbana-Champaign. His main research interests include FPGA-based accelerator design for cryptography, signal/image processing, robotics, and genomics.

Jong Bin Lim received the BS degree in electrical engineering from the University of Illinois at the Urbana-Champaign, in 2014. He is currently working toward the PhD degree in the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign. His current research interests include optimal system-on-chip and accelerator design by using high-level synthesis, and hardware-software co-design.

Zbigniew T. Kalbarczyk is a research professor with the Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign. His research interests include the area of design and validation of reliable and secure computing systems.

Deming Chen received the BS degree in computer science from the University of Pittsburgh, Pennsylvania, in 1995, and the MS and PhD degrees in computer science from the University of California at Los Angeles, in 2001 and 2005, respectively. He is a professor with the ECE Department, University of Illinois at Urbana–Champaign, where he is the Donald Biggar Willett Faculty Scholar. His current research interests include system-level and high-level synthesis, nano-systems design and nano-centric CAD techniques, GPU and reconfigurable computing, hardware security, and computational genomics.

Steven S. Lumetta received the AB degree in physics from the University of California, Berkeley, in 1991, and the MS and PhD degrees in computer science from the University of California, Berkeley, in 1994 and 1998, respectively. He is an associate professor of Electrical and Computer Engineering and a research associate professor with the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign. His research interests include optical networking, high-performance networking and computing, hierarchical systems, and parallel run-time software.

Ravishankar K. Iyer is the George and Ann Fisher distinguished professor of engineering with the University of Illinois at Urbana-Champaign. He holds appointments with the Department of Electrical and Computer Engineering, the Coordinated Science Laboratory (CSL), and the Department of Computer Science, serves as chief scientist of the Information Trust Institute, and is affiliate faculty of the National Center for Supercomputing Applications (NCSA).