RAiSD-X: A Fast and Accurate FPGA System for the Detection of Positive Selection in Thousands of Genomes

NIKOLAOS ALACHIOTIS, Technical University of Crete, Greece
CHARALAMPOS VATSOLAKIS, Technical University of Crete, Greece
GRIGORIOS CHRYSOS, Technical University of Crete, Greece
DIONISIOS PNEVMATIKATOS, Technical University of Crete, Greece

Detecting traces of positive selection in genomes carries theoretical significance and has practical applications, from shedding light on the forces that drive adaptive evolution to the design of more effective drug treatments. The size of genomic datasets currently grows at an unprecedented pace, fueled by continuous advances in DNA sequencing technologies, leading to ever-increasing compute and memory requirements for meaningful genomic analyses. The majority of existing methods for positive selection detection either are not designed to handle whole genomes or scale poorly with the sample size; they inevitably resort to a run-time versus accuracy trade-off, raising an alarming concern for the feasibility of future large-scale scans. To this end, we present RAiSD-X, a high-performance system that relies on a decoupled access-execute processing paradigm for efficient FPGA acceleration, and couples a novel, to our knowledge, sliding-window algorithm for the recently introduced μ statistic with a mutation-driven hashing technique to rapidly detect patterns in the data. RAiSD-X achieves up to three orders of magnitude faster processing than widely used software implementations, and more importantly, it can exhaustively scan thousands of human chromosomes in minutes, yielding a scalable full-system solution for future studies of positive selection in species of flora and fauna.

CCS Concepts: • **Computer systems organization → Reconfigurable computing**; **Data flow architectures**; **Special purpose systems**; • **Applied computing → Bioinformatics**; **Population genetics**; *Computational genomics.*

Additional Key Words and Phrases: Decoupled Access-Execute Architecture, Hardware Accelerator, Positive Selection, Selective Sweep

ACM Reference Format:
Nikolaos Alachiotis, Charalampos Vatsolakis, Grigorios Chrysos, and Dionisios Pnevmatikatos. 2018. RAiSD-X: A Fast and Accurate FPGA System for the Detection of Positive Selection in Thousands of Genomes. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (January 2018), 30 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Positive selection is a natural process of evolution that is initiated by a beneficial mutation, i.e., a change in the structure of a gene that increases the biological fitness of the bearer. In evolutionary terms, fitness (also called Darwinian fitness) defines the reproductive success of an individual, and is typically measured in terms of number of offsprings which produce offsprings that survive. As generations pass, the allele (a variant of a gene) that arose by the beneficial mutation, and

Authors' addresses: Nikolaos Alachiotis, Technical University of Crete, Chania, 73100, Greece, nalachiotis@mhl.tuc.gr; Charalampos Vatsolakis, Technical University of Crete, Chania, 73100, Greece, cvatsolakis@isc.tuc.gr; Grigorios Chrysos, Technical University of Crete, Chania, 73100, Greece, chrysos@mhl.tuc.gr; Dionisios Pnevmatikatos, Technical University of Crete, Chania, 73100, Greece, pnevmati@ece.tuc.gr.

1936-7406/2018/1-ART1 $15.00
https://doi.org/0000001.0000001

consequently improves the chances of survival and reproduction for the carrier (beneficial allele), increases in frequency until all individuals possess it, eventually becoming the only variant of the gene that remains in the population. When this happens, the beneficial allele is said to have become fixed. In the process, neutral alleles (gene variants with no effect on survival and reproduction) that are linked to the beneficial one rise to high occurrence counts, partially eliminating genetic variation/mutations in the region, creating a so-called selective sweep [30]. In other words, a selective sweep is the emergence of a subgenomic region that includes the positively selected locus and exhibits a reduced number of mutations in comparison with the rest of the chromosome.

Detecting positive selection in genomes carries theoretical significance and has practical applications, from shedding light on the forces that drive adaptive evolution [35] to identifying drug-resistant mutations in pathogens [13] and designing more effective drug treatments [6]. Recent breakthroughs in DNA sequencing technologies [43] have reduced sequencing costs and improved accuracy and throughput, leading to the rapid accumulation of genomic data. Inferring positive selection from large-scale genomic data, however, entails several challenges. With a growing number of genomes being sequenced worldwide, supported by multi-party academic and/or industrial collaborations [18, 45], new mutations are constantly discovered, allowing for more accurate scans [36]. The inference of positive selection from genomic data is feasible due to a series of so-called signatures that a selective sweep leaves in the genomes as a result of the fixation of the beneficial mutation and the consequent reduction of linked neutral variation. However, the majority of existing methods and software implementations are not designed to scale with the number of genomes or the genome size, exhibiting prohibitively elevated compute and/or memory requirements, or numerical stability issues. Furthermore, despite the fact that at least three sweep signatures are known, each of the existing software implementations relies on a single signature of a selective sweep.

Due to the aforementioned, method-level insufficiency, a commonly used practice to increase credibility of an analysis is to examine the outcome of multiple runs using different software, with the aim to detect loci for which more than one methods/tests yield a high score. While sweep-detection workflows that combine multiple signatures exist, such as CMS [20] and S/HIC [42], it has been reported that the increased complexity of the underlying pipelines can yield their application cumbersome [2]. To this end, the μ statistic was recently introduced [2], which quantifies three sweep signatures with reduced computational complexity, mostly relying on the enumeration of patterns in the data (polymorphic sites), rather than on compute-intensive kernels that heavily use floating-point arithmetic [14, 36]. To the best of the author's knowledge, the only readily available software implementation of the μ statistic is the sequential tool RAiSD [2] (*https://github.com/alachins/raisd*). Extensive comparisons with existing methods revealed that the μ statistic, as implemented in RAiSD, achieves higher accuracy and sensitivity for various models of the evolutionary history of a population [2]. Furthermore, RAiSD considerably outperforms existing implementations in terms of execution time [2]. However, it relies on a series of brute-force algorithms that deteriorate performance as the sample size and the number of loci increase, due to accessing memory in strides and iteratively conducting redundant computations. Consequently, the employed algorithms do not scale well with the sample size and the number of loci.

In this work, we build upon our previous efforts to accelerate the computation of the μ statistic [4], and propose an optimized FPGA-based accelerator system to improve scalability and performance of RAiSD. Our initial efforts to accelerate the computation of the μ statistic introduced i) an Out-of-Core (OoC) algorithm for parsing data from storage in chunks to maintain reduced memory requirements, ii) a streaming algorithm for a sliding-window implementation of the μ statistic, and iii) a Decoupled Access-Execute Reconfigurable (DAER) accelerator architecture [10] mapped on an FPGA-based system with Hybrid Memory Cube (HMC). While the OoC algorithm maintains low memory requirements irrespectively of the dataset size, it detects the required polymorphic patterns employing a brute-force algorithm that runs exclusively on the host processor. Expectedly, OoC performance dramatically drops when the amount of memory used for storing patterns exceeds the size of the processor's cache, which restricts the size of the chunk that can be fetched from storage before the OoC becomes the bottleneck. However, the effect of acceleration diminishes with a smaller chunk size since the less the data per chunk, the more the required OoC iterations/chunks to be processed. This requires more frequent synchronization between the host processor and the accelerator hardware (more OoC iterations), and reduces the amount of computation on the FPGA (smaller data chunks), hence deteriorating overall accelerator performance. Furthermore, the DAER accelerator comprises a network of interconnected FETCH and PROCESS units that cooperate to process the same chunk. Thus, the memory bandwidth quickly becomes the bottleneck when inter-chunk parallelism is exploited, leading to underutilization of FPGA resources.

Here, we present RAiSD-X, an optimized accelerator system that improves upon our initial acceleration efforts, making the following algorithmic and architectural contributions:

- We present a mutation-driven hashing algorithm to detect polymorphic data patterns, which we dub HAM (Hashing Algorithm driven by Mutations). The brute-force scan of known patterns for each new locus fetched from storage by the OoC algorithm yields a highly inefficient solution that does not scale with the dataset size, and is highly sensitive to the order by which mutations appear in the genome. HAM alleviates this disadvantage by partitioning the memory space reserved for storing patterns based on the number of mutated alleles per pattern, requiring brute-force scanning of at most two partitions instead of the entire set of patterns.
- We propose a new memory layout that places chunk data in memory at contiguous addresses, and introduce a basic set of update rules, henceforth referred to as SUR (Set of Update Rules). These rules allow to fetch data from memory sequentially and update a set of registers to reflect each new step of the sliding-window algorithm using solely the previously stored values in the registers and the data fetched last. This not only improves the effective memory bandwidth of the system, since DRAM page misses are considerably reduced due to sequential memory accessing, but additionally simplifies the design of the FETCH units that provide input data to PROCESS units with enhanced functionality, as described below.
- We present a new DAER architecture that exhibits completely redesigned FETCH and PROCESS units. A PROCESS unit now performs two distinct operations: i) the HAM-dictated, brute-force scan of memory partitions, and ii) the application of SUR for the sliding-window-based processing of the OoC-produced chunk of data. A single FETCH unit serves both PROCESS operations through generic address-generation circuitry, while only occupying one memory port, thereby allowing to increase inter-chunk parallelism by deploying additional FETCH-PROCESS pairs.

RAiSD-X integrates the aforementioned improvements into a complete system-level solution implemented on a small form factor, Linux-based desktop PC, the SC6-mini by Micron Technology, which couples a high-bandwidth Hybrid Memory Cube (HMC) with a mid-range Kintex UltraScale FPGA. Computing the μ statistic on the DAER architecture using SUR, and coupling the resulting hardware accelerator with the HAM-enabled OoC algorithm delivers unprecedented performance for large-scale detection of positive selection. RAiSD-X is up to 124 and 40 times faster than the parallel tools OmegaPlus [1] and SweeD [36], respectively, when 40 threads are launched on 20 CPU cores, and up to 1,755 and 75 times faster than the sequential tools SweepFinder2 [14] and RAiSD [2], respectively. Furthermore, we observed up to 47% overall performance improvement over our first SC6-mini-based accelerator system [4]. Importantly, RAiSD-X can analyze the 2,504 human genomes of the 1000 Genomes project [45] (5,008 sequences due to ploidy, and approximately 78×10^6 loci excluding the sex chromosomes) in less than three hours, thereby showcasing its potential in real-world analyses with thousands of whole genomes.

2 DETECTION OF POSITIVE SELECTION

In the following section, we outline the data preparation process (Section 2.1), and describe a selective sweep (Section 2.2) and its three signatures (Section 2.3).

2.1 Data preparation

Genetic variation (mutations) is observed in the form of single-nucleotide polymorphisms (SNPs), i.e., single base-pair changes in a DNA sequence. A SNP results from one or more mutations at the same genetic location in a genome. Prior to an analysis, a multiple sequence alignment (MSA) is constructed, i.e., a two-dimensional matrix of S rows and l columns that comprises S DNA sequences (one per individual under investigation) of length l nucleotide bases, as shown in Figure 1. Thereafter, a SNP calling tool, such as SAMtools [27] or GATK [31], is used to find SNPs in the MSA, facilitating that way the execution of selection inference tools, since monomorphic sites are not informative for sweep detection.

```
ref_g     ATCATACCCCTCACAAGTAGGTTTTC

seq_1     ATCATACCCCT-CCAACTAGGATTCC
seq_2     ATCCTACCACTCCCAACTAGGTTTCC
seq_3     ATCATAC-C-TCCCAAGTAGGTTTTC
seq_4     ATCATAC-C-TCACAAGTAGGTTTTC
seq_5     ATCATAC-C-TCCCAAGTAGGTTTTC
```

Figure 1. An example of a Multiple Sequence Alignment (MSA) with $S = 5$ DNA sequences (seq_1-5) and $l = 26$ nucleotide bases per sequence. SNPs are highlighted in color, with blue indicating the ancestral allele and red indicating the derived one, using sequence ref_g as reference (not considered part of the MSA).

The example MSA in Figure 1 contains 6 SNPs (polymorphic columns), at positions 3, 8, 12, 16, 21, and 24 (indexing starts at 0). Based on a reference genome (ref_g), which is not part of the MSA, the alleles in each SNP are characterized as either derived (red) or ancestral (blue). The reference genome is constructed based on sequenced DNA data from multiple donors, and therefore serves as a representative example of a species' set of genes. The derived alleles are the result of mutations, whereas the ancestral ones are those that are not derived, thus appearing in the reference genome at the respective positions as well. Note that, without seq_2 present in the data, for instance, there would only be 4 SNPs in the example MSA, since positions 3 and 8 would not be polymorphic (only the respective ancestral allele would be present). Columns that contain alignment gaps ("-") require special handling, which typically takes place during data parsing and varies based on the implemented detection method. The default action of selection detection tools like OmegaPlus and RAiSD (and consequently RAiSD-X) is to discard such columns, as potentially monomorphic. Since this can considerably deteriorate detection accuracy, both RAiSD and RAiSD-X provide optional gap-handling strategies that prevent such columns from being discarded by either imputing gaps or representing them using an additional state ("N"). Deploying any of these strategies only leads to additional polymorphic patterns being identified during the data parsing stage, and thus it does not affect the subsequent calculation of μ statistic scores.

2.2 Selective sweep

A selective sweep is the reduction or elimination of polymorphisms/mutations in a subgenomic region as a result of positive selection. When a beneficial mutation occurs, its frequency increases in the population and eventually reaches fixation (all individuals carry the beneficial mutation). This is due to the fact that a beneficial mutation improves the chances of survival and reproduction for the individuals carrying it over the rest of the population. Due to the existence of additional evolutionary forces that form the genetic composition of a population, neighboring neutral variation (mutations that do not affect survival/reproduction) that is linked to the beneficial mutation diminishes in the proximity of the selected locus, thereby creating a selective sweep. Figure 2 illustrates two instances of the same population at different junctures: Figure 2A depicts the population when a beneficial mutation first occurs (prior to a sweep), while Figure 2B shows the expected reduction of polymorphisms when the beneficial mutation is fixed (after the sweep).

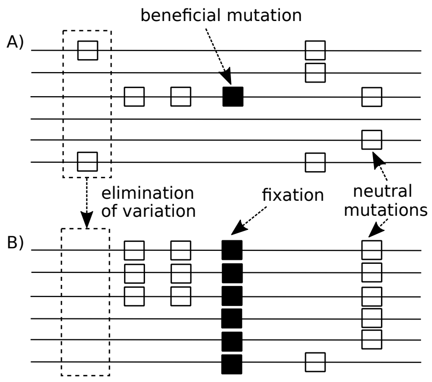

Figure 2. A population at two different junctures: A) when the beneficial mutation (solid square) first occurs, and B) after the beneficial mutation is fixed. Note the elimination of neutral variation due to the selective sweep.

2.3 Sweep signatures

A selective sweep is expected to leave one or more of three distinct signatures in a genome as a result of the aforementioned increase of the frequency of the beneficial mutation and additional evolutionary forces, facilitating the detection of loci that have undergone positive selection. Studies typically rely on a combination of sweep signatures (tests), under the rationale that the more tests agreeing on an outcome, the more likely the outcome. According to the selective sweep theory [30], the first signature is the localized reduction of the polymorphism level (genetic variation), which is observed as a subgenomic region with a reduced number of SNPs in comparison with the rest of the chromosome. To the best of our knowledge, there is no readily available software implementation for this signature.

The second sweep signature is a particular shift in the Site Frequency Spectrum (SFS) toward low- and high-frequency derived variants [8]. As already mentioned, a SNP consists of two allele types, the derived and the ancestral. While the former is the result of a mutation, the latter is any allele that is not derived. A selective sweep therefore creates subgenomic regions with the majority of the SNPs comprising either a low number of derived alleles, e.g., one or two, or a high number of derived ones, e.g., $S-1$ or $S-2$, where S is the total number of sequences under investigation (sample size). Widely used tools that rely on the SFS-based signature of selective sweeps, such as SweepFinder [34], SweepFinder2 [14], and SweeD [36], exhibit highly compute-intensive implementations of composite likelihood ratio (CLR) tests, and thus suffer from prohibitively long execution times.

The third sweep signature is a localized pattern of Linkage Disequilibrium (LD) levels. LD is the non-random association between alleles at different loci, and is used to quantify the existence of associated alleles when the observed allele associations differ from what one would expect if the alleles were inherited independently. Various measures of LD have been proposed, such as D' [26] or the Pearson's correlation coefficient r. Yet, the most commonly used measure is the squared Pearson's coefficient, r^2 [21], which has the advantage that all values are between 0.0 and 1.0, thus facilitating comparisons between pairs of SNPs in different subgenomic regions. Kim and Nielsen [24] showed that increased LD levels (summed over all SNP pairs in a region) are observed on each side of a beneficial mutation, whereas the amount of LD between loci that are located on different sides of the beneficial allele remains low. Software tools that rely on LD, such as Omega [38] and OmegaPlus [1], implement the ω statistic [24], which is significantly less computationally intensive than the aforementioned CLR tests, and performs better in terms of power to detect selection [12]. Nevertheless, ω-statistic implementations exhibit prohibitively large memory requirements since a substantial amount of entire SNPs, typically in the order of thousands, needs to be kept in memory in order to hide memory access latencies and achieve high performance in LD computations [3].

3 RELATED WORK

The development of statistical methods to obtain high-quality SNP data from next-generation sequencing data flourished in recent years [33]. This enabled major advancements in modeling and statistical analyses for population genetics, and triggered the release of various software solutions for selective sweep detection. In this section, we review selection inference approaches and custom-hardware accelerators reported to scale to a large number of whole genomes and SNPs, and point out disadvantages that can potentially deem them to be unsuitable for future dataset sizes [18].

3.1 Methods and tools

Nielsen et al. [32] released SweepFinder, a SFS-based software that implements a composite likelihood ratio (CLR) test where the numerator is the likelihood of a selective sweep and the denominator accounts for neutrality, i.e., the observed frequency of mutations per genomic location is the result of stochastic processes (no selective sweep occurred). While SweepFinder can analyze entire genomes and maintain low memory requirements, it suffers from long execution times due to heavily relying on floating-point operations and only employing a single core. As already mentioned, SweepFinder is also reported to be numerically unstable when the sample size exceeds 1,027 sequences [36], due to unhandled floating-point exceptions.

DeGiorgio et al. [14] released SweepFinder2, which employs the same statistical framework as SweepFinder and exhibits increased sensitivity. Particular code modifications yield a significantly more stable solution for computing likelihood scores, but performance is not improved as the software still deploys a single processing core. In fact, both SweepFinder and SweepFinder2 exhibit similarly long processing times, with the latter requiring more than 5 hours to scan 1,000 datasets that simulate subgenomic regions (approx. 2,000 loci each) from 20 samples, when RAiSD [2] completes the analysis in under 6 minutes.

Pavlidis et al. [36] released SweeD, a software tool that also relies on the aforementioned statistical framework. SweeD avoids the aforementioned numerical stability issues via a modified set of mathematical operations that allow to correctly analyze thousands of whole genomes. Furthermore, it employs multiple CPU cores to reduce execution time. Yet, the excessive computational requirements, due to the floating-point-intensive CLR test, remain. Furthermore, the tool scales poorly with the number of CPU cores; it achieves speedups of as low as 8x on a 20-core processor for the analysis of 1,000 datasets that simulate subgenomic regions (approx. 8,000 loci each) from 100 samples.

Alachiotis et al. [1] proposed a dynamic programming algorithm for the ω statistic [24], and released the parallel software OmegaPlus, which relies on the LD signature of a selective sweep. Calculating LD scores is considerably less compute-intensive than the CLR tests implemented in SweepFinder/SweepFinder2 and SweeD. However, computing LD scores, which is a memory-bound operation [3], is the limiting factor for performance due to the typically high number of pairwise calculations in whole-genome scans. Furthermore, OmegaPlus exhibits prohibitively large memory footprints because the entire dataset is loaded to memory prior to processing. For instance, the tool lacks the capacity to analyze the first human chromosome (data available by the 1000 Genomes project [45]) on a high-end, off-the-shelf personal computer with 32 GB of main memory due to excessive memory requirements for parsing the input file (65.8 GB).

In addition to the aforementioned stand-alone software implementations, several summary statistics for selective sweep detection have been proposed [15, 16, 47]. These are inexpensive calculations on SNP data, e.g., counting the number of polymorphic sites in a genomic region, typically applied at whole-genome scale using a sliding-window approach with fixed window size and step. Summary statistics serve as tests for neutrality due to the fact that their distributions differ distinctively between the presence and absence of positive selection. Note however that the fixed window size is of critical importance to the detection process, as accepting/rejecting neutrality highly depends on the window size [37]; this is why more advanced tests, such as SweepFinder and OmegaPlus, evaluate windows of varying sizes.

Several excellent surveys on detecting selective sweeps have been published, with varying focus, from detecting sweeps on ancient DNA [29] to assessing the challenges that whole-genome data pose to population genetics [39]. Crisci et al. [12] reviewed widely-used approaches to detect positive selection, including SweepFinder, SweeD, and OmegaPlus, concluding that OmegaPlus outperforms the other methods in both equilibrium and non-equilibrium evolutionary scenarios. Alachiotis and Pavlidis [2] performed a similar study, including SweepFinder2 and RAiSD, finding that RAiSD yields more accurate scans at a fraction of the time required by the other tools. Note that RAiSD and RAiSD-X (the hardware-accelerated system presented here) yield identical results.

3.2 Hardware accelerators

Various computationally demanding bioinformatics applications and kernels have been successfully ported to FPGAs [28, 46, 49] and/or GPUs [40, 50, 51], reporting considerable performance improvements. To the best of the author's knowledge, Alachiotis et al. [4] presented the first attempt to address the problem of inferring positive selection from SNP data at the hardware level, targeting a mid-range Xilinx Kintex UltraScale FPGA. The authors report up to 751x, 62x, and 20x faster analyses of simulated genomes than the software tools SweepFinder2 (1 thread), OmegaPlus (40 threads), and SweeD (40 threads), respectively, when run on a Dell PowerEdge R530 rack server with two 10-core Intel Xeon processors. As already mentioned, however, this accelerator system executes a brute-force, software-only algorithm to find SNP patterns in the data, which yields system performance highly sensitive to the number and location of patterns. RAiSD-X alleviates this drawback with a hardware-accelerated, hash-based implementation that finds patterns orders of magnitude faster, yielding a method-independent solution which can be applied for any SNP-based statistic that relies on patterns, e.g., detecting the dominant pattern in a subgenomic region as an indicative measure of introgression [54].

Bozikas et al. [7] described a multi-FPGA system that is capable of accurately computing LD on millions of sequences. This was achieved by avoiding on-chip buffering of SNP data, inevitably yielding an I/O-bound architecture that underutilized device resources. The authors report speedups of up to 56x when four Virtex 6 FPGAs are compared with the sequential execution of the open-source software PLINK1.9 [9] on an Intel Xeon processor (sequential execution). Alachiotis and Weisz [5] presented an LD accelerator that relies on prefetching and the caching of entire SNPs on chip to deliver high performance, implicitly imposing an upper bound on the sample size. The authors report speedups of up to 159x using one FPGA device in comparison with the same reference software (PLINK1.9 [9]). While both LD acceleration approaches employ FPGA technology to boost performance, LD alone can not serve as a test for neutrality. Thus, the aforementioned architectures can not be used as-is for positive selection inference.

4 MULTI-SIGNATURE DETECTION METHOD

In the following section, we give the definition of the μ statistic (Section 4.1), and provide an example of two consecutive score calculations (Section 4.2).

4.1 Method definition

The μ statistic [2] considers all three sweep signatures simultaneously, eliminating the need to analyze a dataset more than once, and consequently the need for the post-execution combination of the results. It detects sweep signatures by relying on the enumeration of occurrences of SNPs in sliding windows, rather than compute-intensive tests [32]. This alleviates the need for increased compute and memory capacity, which state-of-the-art software tools ([1, 14]) typically exhibit, and predominantly relies on integer arithmetic, allowing to reduce hardware resources and increase parallelism on an FPGA.

To compute the μ statistic assume a two-dimensional $S \times D_{sz}$ MSA-like structure D (see Figure 4) that consists solely of SNPs, where D_{sz} is the number of SNPs and S is the sample size. Let D_{ln} be the length of the genomic region in nucleotide bases that corresponds to the D_{sz} SNPs, and W_{sz} be the size of a window W in SNPs. Additionally, let s_i denote SNP i in D, and l_i denote its location. The final statistic value is computed as follows:

$$\mu = D_{ln} \times (\mu^{\text{VAR}} \times \mu^{\text{SFS}} \times \mu^{\text{LD}}), \tag{1}$$

based on three factors, one for each sweep signature.

Genetic variation in W is measured as follows:

$$\mu^{\text{VAR}} = \frac{l_{W_{sz}-1} - l_0}{D_{ln} \times W_{sz}}, \tag{2}$$

where l_0 and $l_{W_{sz}-1}$ are the locations of the first and the last SNPs in the window, respectively. Equation 2 assumes high values, indicating reduced genetic variation, when the W_{sz} SNPs in W correspond to a large genomic region.

The expected shift in the SFS is computed as follows:

$$\mu^{\text{SFS}} = \frac{\sum_{s_i \in W}[M(s_i) \leq m] + \sum_{s_i \in W}[M(s_i) \geq S - m]}{W_{sz}}, \tag{3}$$

where [ ] is the Iverson bracket notation (it returns 1 if the logical proposition expressed by the statement in the brackets is true; otherwise returns 0), $M(s_i)$ is the number of derived alleles at SNP s_i, and m is the maximum number of derived alleles per SNP. Equation 3 assumes high values with an increased total number of SNPs that comprise at most m or at least $S - m$ derived alleles. Note that, Equation 3 in the initial RAiSD release assumed high values with an increased total number of SNPs that comprised exactly one or $S - 1$ derived alleles. Introducing the m parameter in RAiSD-X ensures that the numerator receives non-zero values in the case of complete absence of SNPs with one or $S-1$ derived alleles, which would otherwise led to μ^{SFS}, and thus μ, becoming zero.

The expected pattern of LD among SNPs is calculated based on the rationale that a reduced number of different SNPs is expected in a region of high LD, and vice versa. To account for the expected low LD across the beneficial mutation, assume that W is split into two subwindows, W_L and W_R, with $W_{sz}/2$ SNPs each, as illustrated in Figure 3. The third sweep signature is therefore captured as follows:

$$\mu^{\text{LD}} = \frac{\sum_{s_i \in W_L}[s_i \in P_L][s_i \notin P_R] + \sum_{s_i \in W_R}[s_i \notin P_L][s_i \in P_R]}{\sum_{s_i \in W_L}[s_i \in P_L] \times \sum_{s_i \in W_R}[s_i \in P_R]}, \tag{4}$$

where P_L and P_R are the sets of SNP patterns in the W_L and W_R subwindows, respectively. In Equation 4, the numerator assumes high values with an increasing number of exclusive SNP patterns per subwindow, while the denominator assumes low values with a reduced number of SNP patterns in each of the subwindows.

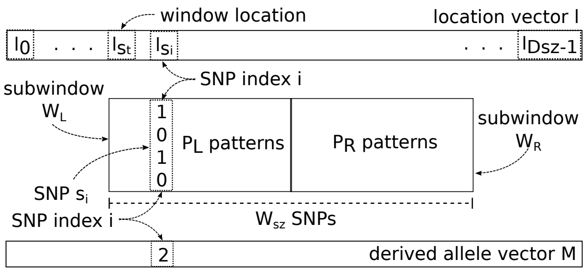

Figure 3. The window W is organized into two subwindows, L and R, to facilitate cross-LD evaluation based on the P_L and P_R SNP patterns.

4.2 Example

Figure 4 shows an example of two consecutive calculations of the μ statistic for an MSA with $S = 4$ samples and $D_{ln} = 20$ sites. The structure D comprises $D_{sz} = 12$ SNPs, indexed from 0 to 11 (the SNP indices are not shown in the figure for the sake of clarity), while the size of the window W is $W_{sz} = 8$ SNPs. The sliding-window algorithm proceeds with a step of 1 SNP for a total of $D_{sz} - W_{sz} + 1 = 5$ iterations. For window W_0, we have $l_0 = 1$ and $l_7 = 13$, whereas for W_1 we have $l_0 = 2$ and $l_7 = 15$. We can therefore compute Equation 2 as $\mu^{VAR} = (13-1)/(20 * 8) = 0.075$ for window W_0, and $\mu^{VAR} = (15-2)/(20 * 8) = 0.08125$ for window W_1.

To compute Equation 3 and Equation 4, each SNP in W is represented by a binary vector, adopting the assumption of the infinite-sites model [25], henceforth denoted ISM. Under this model, an infinite number of sites is assumed, and each new mutation consequently appears on a site where no mutation has previously occurred. When a SNP is represented by a binary vector, each unset bit (‘0’) usually indicates the ancestral allele while each set bit (‘1’) represents the derived/mutated allele. It should be noted that the assignment of ‘0’ to the ancestral allele is arbitrary. One could easily have used ‘1’ to represent the ancestral state and ‘0’ for the derived one. Note for instance that the SNPs at genomic locations 6 and 7 differ, yet they are represented by the same binary vector under ISM. Similarly, the SNPs at genomic locations 9 and 11 are identical, yet one is represented by the bit-wise complement of the other. Computing the number of mutations per SNP (M, see Figure 4) only requires to count the number of set bits per binary vector (population count). One can now compute Equation 3 by counting the number of M entries with values 1 and $S-1 = 3$, leading to $\mu^{SFS} = (2+2)/8 = 0.5$ for both windows.

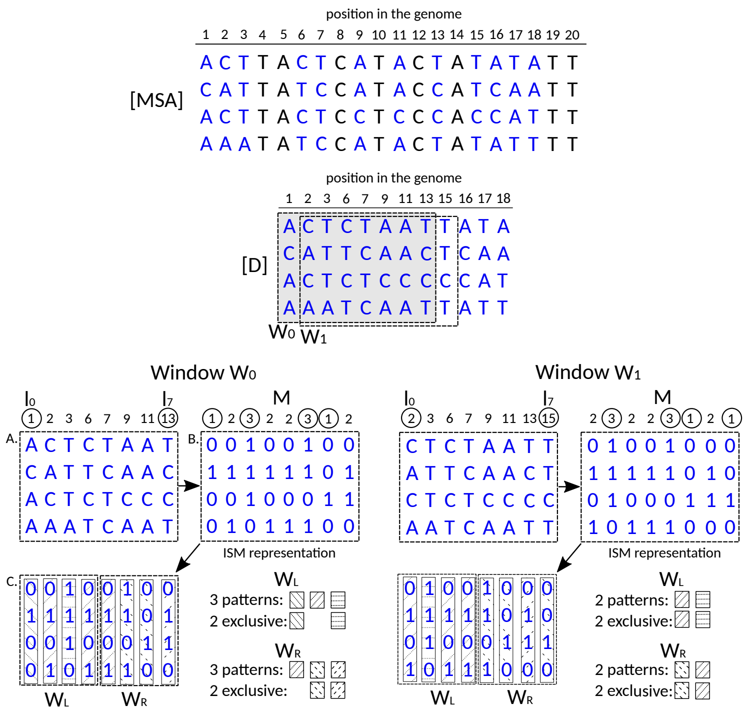

Figure 4. Two consecutive sliding-window calculations of the μ statistic on structure D, which consists only of SNPs (all monomorphic columns in the MSA are discarded). A) Only the SNP positions (vector l) are used for computing μ^{VAR}. B) Computing μ^{SFS} requires the number of derived alleles per SNP (vector M). C) The total number of patterns in the left and right subwindows, as well as those appearing exclusively in each subwindow, are used for computing μ^{LD}.

Finally, computing Equation 4 requires to split the window into two non-overlapping subwindows W_L and W_R. For convenience, the window size W_{sz} is always an even number, to allow equal-size subwindows. Equation 4 is computed by counting the number of patterns in W_L and W_R, the number of exclusive patterns in W_L and W_R, as well as the number of exclusive SNPs in W_L and W_R. What constitutes a pattern is defined in terms of perfect LD, i.e., $LD = 1.0$ when measured based on the squared Pearson's correlation coefficient r^2. In other words, two SNPs are the same pattern when their binary representations are identical or one is the bit-wise complement of the other. Thus, both W_0 subwindows comprise a total of 3 SNP patterns each, with one pattern ("0101") appearing in both W_L and W_R, leaving 2 exclusive patterns per subwindow. After the window shift, both W_1 subwindows now comprise a total of 2 SNP patterns each, and all the patterns in both W_L and W_R appear exclusively in their respective subwindows. Equation 4 is therefore computed as $\mu^{LD} = (2 + 2)/(3 + 3) = 0.667$ and $\mu^{LD} = (2 + 2)/(2 + 2) = 1.0$ for windows W_0 and W_1, respectively. Using Equation 1, the final μ statistic values for W_0 and W_1 are computed as $\mu = 20 \times (0.075 \times 0.5 \times 0.667) = 0.50025$ and $\mu = 20 \times (0.08125 \times 0.5 \times 1.0) = 0.8125$, respectively.

5 SYSTEM DESIGN

In the following section, we present the design of RAiSD-X. We initially revise the Out-of-Core algorithm (OoC, Section 5.1) that executes on the host processor to parse data from storage in chunks, and introduce the hardware-accelerated Hashing Algorithm based on Mutations (HAM, Section 5.2) that finds SNP patterns in the data. Thereafter, we introduce an interleaved data layout and a Set of Update Rules (SUR, Section 5.3) that collectively allow to compute the μ statistic dynamically with reduced computational complexity, facilitating hardware design and improving performance. Finally, we present the enhanced Decoupled Access-Execute Reconfigurable accelerator architecture (DAER, Section 5.4) that boosts HAM performance and evaluates sliding windows employing SUR.

5.1 Out-of-Core (OoC) algorithm

The primary operation of the host processor is to fetch input data from storage space in parameterized-size chunks. This ensures that the memory footprint does not increase with the number of samples or SNPs, while the parameterized size allows to adapt the algorithm to various processor architectures and/or accelerator platforms. A SNP chunk of size 1 MB, for instance, can reduce cache misses, or allow to store an entire SNP chunk on FPGA on-chip memory throughout execution. More importantly, the described OoC approach is both application- and method-agnostic, and can be deployed to maintain the memory footprint of tools that process large-scale SNP data below a user-defined threshold.

The OoC algorithm is depicted in Figure 5. The input dataset is loaded from storage space on a SNP-by-SNP basis. SNPs that are stored in row-major order (VCF [27] format) are fetched with minimal parsing overhead, relying on an iterative routine that sequentially loads one byte/character per iteration until a complete SNP is parsed. When SNPs are stored in column-major order (ms [22] / FASTA formats), however, the next allele in the SNP is placed several bytes away from the one loaded last. The exact byte distance between same-SNP alleles depends on the file format and the number of SNPs. Hence, prior to loading the next character, the file position indicator needs to be set accordingly.

When an entire SNP is loaded, e.g., SNP i in Figure 5, a filtering step (`SNP Filtering`) applies a series of checks to verify correctness and quality. The correctness checks ensure that the loaded SNP is indeed a SNP, i.e., it comprises at least two allele states as a result of at least one mutation, and that the SNP size matches the dataset's sample size. An optional quality check allows to discard SNPs that do not pass a user-defined quality threshold (QUAL field in VCF specification). The verified SNP is then converted to a compact binary representation (`Binary Conversion`) that reduces memory footprint and facilitates processing. The allele encoding scheme is dictated by the mutation model, which can either be the ISM [25] or a Finite Sites Model [48]. As already mentioned, the former assumes at most one mutation per site (1 bit per state), whereas the latter allows all possible DNA states (A, C, G, and T) or any of the DNA ambiguity codes that are associated with every possible combination of the four DNA states at every site (4 bits per state). Upon conversion to binary, a pattern matching routine (`Pattern Match`) compares the verified SNP with all existing SNP vector patterns in the `PATTERN POOL` data structure. In the case that the entire pool is scanned and no matching pattern is found, the incoming SNP is stored in the pool as a new pattern. In any case, the pattern index y is stored along with the corresponding SNP location $loc\ i$ in the `REGION` data structure, which describes the genomic region to be processed. Both structures maintain a nxt pointer that indicates the next available position for a new entry. The `PATTERN POOL` data structure exhibits increased versatility and bounded memory allocation through a flexible management mechanism that adapts the number of patterns n that the pool can accommodate to the desired memory footprint for the sample size S.

The OoC algorithm yields a generic solution that is applicable to a variety of SNP analyses. We observe, however, that memory requirements can be further reduced when the OoC algorithm is used for selective sweep detection without inducing an overhead. This is due to the fact that the number of derived alleles that is required by Equation 3 (see also Figure 3) is already calculated at

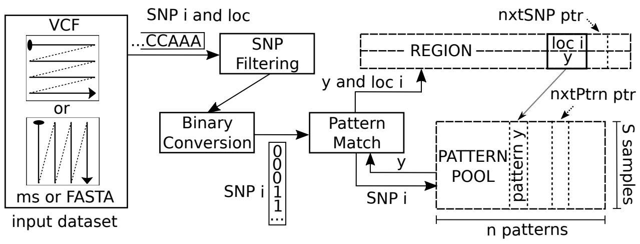

Figure 5. The OoC algorithm for parsing large-scale SNP data (source: [4]). Patterns are stored in their entirety in the `PATTERN POOL` data structure. An accurate description of the genomic region is maintained through the `REGION` structure, which comprises pointers to SNP patterns in the pool.

the `SNP Filtering` stage, which enumerates the number of derived alleles per SNP as part of its correctness checks, and thus it can be stored as an additional field in the `REGION` structure.

5.2 Hashing Algorithm based on Mutations (HAM)

For every SNP that passes the `SNP Filtering` stage, the `Pattern Match` routine performs a brute-force search of the `PATTERN POOL` with $O(n \times l)$ worst-case time complexity, where n is the number of patterns in the pool, and l is the pattern size in k-bit words. While the pattern size l is fixed throughout execution ($l = \lceil \frac{S}{k} \rceil$ for S samples under ISM), the number of patterns n increases as the pool grows in size. Consequently, only a limited number of patterns can be added to the pool before the `Pattern Match` routine becomes the limiting factor for performance. Figure 6 shows the total time required by `Pattern Match` to detect all the patterns in three simulated datasets of different sizes when the memory footprint of `PATTERN POOL` increases from 0.5MB to 10MB. As can be observed in the figure, allocating more memory increases the total time required for detecting patterns, with the worst-case execution times observed when the pool is large enough to host all the patterns in the data. Extensive profiling of RAiSD (version 1.7) with a 1MB `PATTERN POOL`, using datasets of various sizes (between 20 and 1000 samples) and formats (ms [22] and VCF [27]), revealed that the `Pattern Match` operation alone can occupy up to 65% of the total execution time. A larger pool size, however, allows to construct a `REGION` structure that corresponds to a wider genomic region, thereby reducing the number of OoC iterations required in order to complete an analysis. This improves the computation-to-synchronization ratio for the DAER accelerator that computes the μ statistic, thereby boosting its performance, since it is the `REGION` structure that is transferred (input data) to the accelerator for this computation. The effect of the number of OoC iterations on the performance of the DAER accelerator that computes the μ statistic is discussed in more detail in Section 6.

To this end, we introduce the HAM algorithm, which considerably lowers the overhead introduced by `Pattern Match` when the `PATTERN POOL` size grows. The underlying idea is to organize the allocated memory space into equally sized partitions, with each partition hosting patterns with the same number of derived alleles. For every SNP that passes the `SNP Filtering` stage, the `Pattern Match` routine now performs a brute-force search of at most two partitions, as described below, each containing a number of patterns p that is considerably lower than the total number of patterns n. Therefore, the worst-case time complexity is $O(p \times l)$ instead of $O(n \times l)$, with $p << n$, allowing to increase the `PATTERN POOL` size without deteriorating performance.

The HAM algorithm, which is part of `Pattern Match`, is depicted in Figure 7, while its pseudocode is provided in Algorithm 1. For a number of genomes S, the total number of partitions is $S - 1$, since a SNP is polymorphic by definition. A partition is described by a memory address

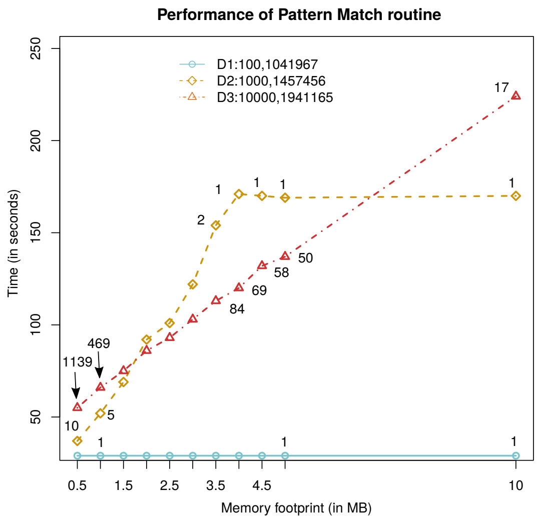

Figure 6. Time performance of `Pattern Match` for three simulated datasets, $D1-3$ (described by the sample size followed by the number of SNPs), when the `PATTERN POOL` size increases. The figure also contains the number of OoC iterations for some of the runs. The worst-case performance is observed when the pool size is sufficiently large to host all the patterns in the data, i.e., a single OoC iteration is required. In this case, allocating additional memory for the `PATTERN POOL` neither improves nor deteriorates the performance of `Pattern Match`, hence the observed plateau of the curves.

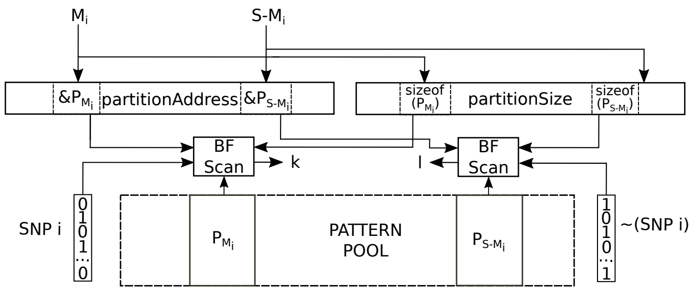

Figure 7. The HAM algorithm for finding SNP patterns with reduced worst-case time complexity. It improves performance of the OoC's `Pattern Match` stage by employing a hash-based approach that only requires to brute-force search a fraction of the `PATTERN POOL`. For a SNP i with M_i derived alleles, only partitions P_{M_i} and P_{S-M_i} need to be exhaustively searched.

(`partitionAddress` array) and a size (`partitionSize` array), defined in terms of number of patterns. The population counter conveniently serves as the hash function to compute an index into the `partitionAddress` and `partitionSize` arrays, which is already computed for SNP i in the `SNP Filtering` stage as part of the necessary correctness checks (see Figure 5). This is the number of derived alleles M_i, which is used to retrieve partition information for two partitions, one that hosts patterns with M_i derived alleles (partition P_{M_i}), and one that hosts patterns with $S-M_i$ derived alleles (partition P_{S-M_i}). A dedicated routine (`BF Scan` in Figure 7, where `BF` stands for brute-force) initially compares SNP i against all the patterns in partition P_{M_i}, and the respective pattern index k is returned if the SNP is matched to a pattern, in which case k corresponds to y in Figure 5.

ALGORITHM 1: Pseudocode of the OoC algorithm that employs HAM for pattern matching. The brute-force scan of each partition (*BF_SCAN* operation) is performed by the PROCESS units in RAiSD-X.

```
Data: $partitionAddress$ and $partitionSize$ arrays
Result: Location of $SNP\_i$ in the $PATTERN\_POOL$
$matchIndex \leftarrow$ -1 ; // Initialization
while PATTERN_POOL != FULL do
    $SNP\_i \leftarrow$ Next SNP from file;
    if $SNP\_i$ passes SNP_FILTERING checks then
        $c \leftarrow POPCOUNT(SNP\_i)$;
        $PM1 \leftarrow partitionAddress[c]$;
        $PS1 \leftarrow partitionSize[c]$;
        $matchIndex \leftarrow BF\_SCAN(PM1, PS1)$;
        if $matchIndex$=-1 then
            $c \leftarrow POPCOUNT(\sim SNP\_i)$ ; // Bit-wise complement
            $PM2 \leftarrow partitionAddress[c]$;
            $PS2 \leftarrow partitionSize[c]$;
            $match \leftarrow BF\_SCAN(PM2, PS2)$;
        end
        if $matchIndex$=-1 then
            Add $SNP\_i$ to Partition $PM1$ ;
            Increment $PS1$ ;
            return $PS1 - 1$ ;
        else
            return $matchIndex$;
        end
    end
end
```

Otherwise, the SNP's bitwise complement is compared against partition P_{S-M_i}, accounting for the fact that the assignment of '0' to the ancestral allele is arbitrary. Similarly, if the SNP is matched to a pattern in partition P_{S-M_i}, BF Scan returns the respective pattern index l, corresponding to y in Figure 5 in this case. Otherwise, SNP i is added as a new pattern in partition P_{M_i}, and the new pattern index is returned. In RAiSD-X, the host processor performs the population count operation and indexes the partitionAddress and partitionSize arrays, while the brute-force search of the partitions (BF Scan operation) is performed by the PROCESS units (described in more detail in Section 5.4).

The astute reader will have noticed that, for a fixed memory footprint, the maximum partition size decreases with an increasing sample size, requiring considerably more OoC iterations to complete an analysis. Therefore, it is a prerequisite for performance to allocate additional memory for PATTERN POOL when employing HAM in order to maintain (if not improve) a favorable computation-to-synchronization ratio for the DAER accelerator. Otherwise, while OoC performance improves due to reducing the worst-case time complexity per pool scan, the high number of OoC iterations, and consequently the high number of accelerator synchronization events, will result in diminishing performance improvements for computing the μ statistic.

5.3 Interleaved memory layout and Set of Update Rules (SUR)

When the PATTERN POOL is full, i.e., a partition has reached its maximum size, the SNP loading process is temporarily paused and the REGION structure is transferred to the DAER hardware accelerator, which executes a sliding-window algorithm on three data vectors, the location vector

L, the mutation vector M, and the pattern vector P. Each window requires a new triplet of values $T_i = (L_i, M_i, P_i)$ to be fetched from external memory, which collectively represent the i^{th} SNP in the subgenomic region described by the REGION structure (Figure 8A). To avoid strided memory accesses, we employ a new memory layout that interleaves vectors L, M, and P per SNP (Figure 8B). Consequently, L_i, M_i, and P_i are placed in adjacent memory locations in the REGION structure, and accessed sequentially, which is essential for the efficient utilization of the memory bandwidth. Furthermore, the number of *FETCH* units required to operate in tandem to compute a μ-statistic value is reduced from three (one per L, M, and P vectors) down to one, since a single *FETCH* unit, operating as a simple DMA engine, can now load the next T_i. Dictated by the DAER framework [10], a dedicated *FETCH* unit is assigned to each memory port. Therefore, interleaving data in memory allows to instantiate an accelerator core per available memory port, thereby increasing parallelism and boosting aggregate system performance.

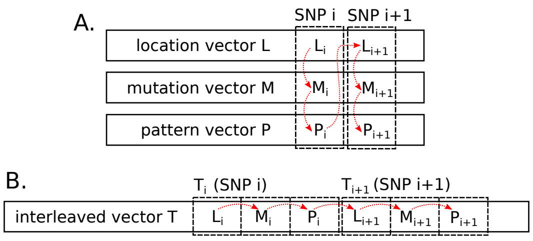

Figure 8. A) The initial memory layout of REGION structure, employed by RAiSD [2] and our first accelerator system [4], which requires accessing data at a stride. B) The interleaved memory layout employed by RAiSD-X, which allows to access contiguous data.

To efficiently compute the μ statistic in hardware using the interleaved memory layout, we introduce SUR, a series of rules that update a number of registers according to each T_i fetched from memory per sliding window iteration. Given a window size of W_{sz} SNPs and a step size of 1 SNP, SUR allows to process a REGION structure of D_{sz} SNPs with worst-case time complexity of $O(D_{sz} \times W_{sz})$, whereas a naive implementation that performs $(W_{sz}/2) \times (W_{sz}/2)$ comparisons per window to enumerate the exclusive patterns, as required by Equation 4, exhibits worst-case time complexity of $O(D_{sz} \times W_{sz}^2)$. When $W_{sz} = 10$ SNPs, for instance, computing the μ statistic by performing $(W_{sz}/2) \times (W_{sz}/2)$ comparisons per window to analyze a simulated dataset with 20 samples occupies 19% of RAiSD's total execution time. When the window size increases to 500 SNPs, RAiSD spends 95% of its total execution time on the sliding-window steps. Reducing the time complexity allows to increase the window size, which is defined in RAiSD by a constant (10), thereby potentially improving detection accuracy. Assessing the effect of W_{sz} on detection accuracy, however, requires a thorough examination of a vast amount of simulated populations that evolved under different evolutionary pressures, which is beyond the scope of the present work.

To apply SUR, we employ a fixed-size circular queue, CQ, which holds the fraction of vector T that corresponds to the current window at any point in time, as illustrated in Figure 9. We henceforth follow the convention that a window is defined by the index of its last SNP in the REGION structure, e.g., T_i for window W_i. Deploying a circular queue is an efficient approach to slide the window, since it does not require to move data per iteration [19, 41]. Sliding is achieved by simply updating three pointers. The first two pointers, denoted `start` and `end` in Figure 9, point to the first and last SNPs in the current window, respectively. Evidently, the window size W_{sz} defines the CQ size, as the queue must be sufficiently large to store all the SNPs in a window. We introduce one additional slot in CQ (and a third pointer denoted `extra` in Figure 9), between the first and last SNPs in a window, to prevent overwriting the first SNP of the previous window when the next sliding step occurs, i.e., when T_i is entered into the queue. This simplifies the set of updates rules employed for Equation 4, reducing time complexity and resource utilization. For window W_{i-1}, `start` points to $T_{(i-1)-(W_{sz}-1)}$, end points to T_{i-1}, and `extra` points to $T_{(i-2)-(W_{sz}-1)}$, whereas for window W_i, `start` points to $T_{i-(W_{sz}-1)}$, end points to T_i, and `extra` points to $T_{(i-1)-(W_{sz}-1)}$.

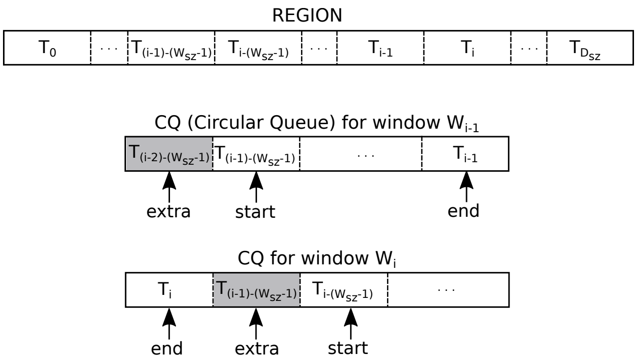

Figure 9. A circular queue, CQ, stores the part of the `REGION` structure that corresponds to the current window at any point in time. The window extends between the `start` and end pointers, whereas `extra` points to the first SNP in the previous window. Sliding is achieved by updating the `start`, end, and `extra` pointers.

To compute μ^{VAR}, the values $L_{i-(W_{sz}-1)}$ and L_i are fetched from CQ and assigned to l_0 and $l_{W_{sz}-1}$ (see Equation 2), respectively. Recall that $T_i = (L_i, M_i, P_i)$, where i is the SNP index in the `REGION` structure. For the μ^{SFS} factor, we introduce two registers, $R1$ and $R2$, which hold the number of singletons and the number of SNPs with $S-1$ mutations in W_i, respectively. Each register is initialized by the first window in the `REGION` structure, i.e., $W_{W_{sz}-1}$, with W_{sz} comparisons. For every subsequent window W_i, $R1$ and $R2$ are updated as follows:

$$R1 = \begin{cases} R1-1 & if\ M_{i-W_{sz}} = 1\ and\ M_i \neq 1 \\ R1+1 & if\ M_{i-W_{sz}} \neq 1\ and\ M_i = 1 \\ R1 & \text{otherwise,} \end{cases} \tag{5}$$

and

$$R2 = \begin{cases} R2-1 & if\ M_{i-W_{sz}} = S-1\ and\ M_i \neq S-1 \\ R2+1 & if\ M_{i-W_{sz}} \neq S-1\ and\ M_i = S-1 \\ R2 & \text{otherwise,} \end{cases} \tag{6}$$

where S is the sample size. After $R1$ and $R2$ have been updated to reflect the new window, Equation 3 is evaluated as $\mu^{\text{SFS}} = (R1+R2)/W_{sz}$, since $\sum_{s_i \in W}[M(s_i) = 1] = R1$ and $\sum_{s_i \in W}[M(s_i) = S-1] = R2$.

To compute μ^{LD}, recall that a window W is split into two non-overlapping subwindows, W_L and W_R, with $W_{sz}/2$ SNPs each. We instantiate four dedicated memory blocks, as illustrated in Figure 10, to store the patterns in the left subwindow (Le), the patterns in the right subwindow (Ri), the exclusive patterns in the left subwindow (xLe), and the exclusive patterns in the right subwindow (xRi) of the window W_i residing in CQ at any point in time. All memory blocks are sufficiently large to store the maximum number of patterns in a subwindow, i.e., $W_{sz}/2$ SNPs, and additionally hold the number of occurrences per pattern in each subwindow. Note that, a pattern is

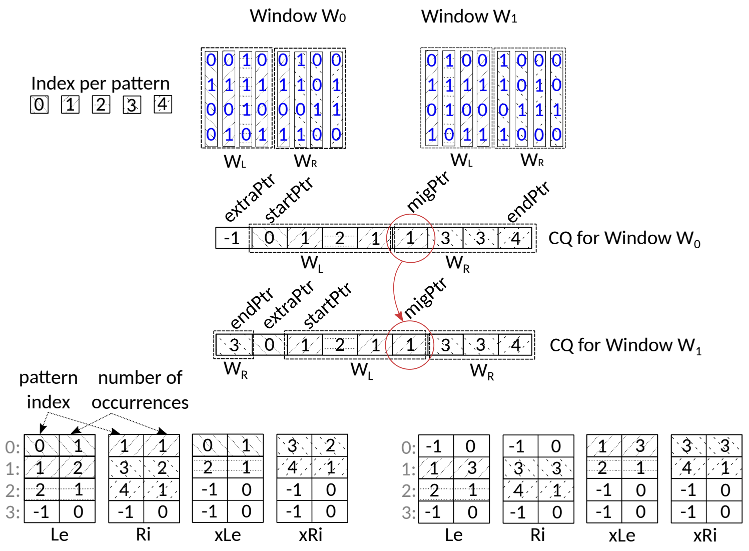

Figure 10. The content of the CQ and the memory blocks Le, Ri, xLe, and xRi for the two consecutive sliding-window calculations of the μ statistic illustrated in Figure 4. Note the dedicated mig pointer that points to the first SNP in W_R of the previous window, which is the SNP that migrates from the previous right subwindow to the current left subwindow with every sliding-window step.

represented in a memory block by its index in the PATTERN POOL, i.e., P_i for SNP i in the REGION structure. The figure shows the content of the memory blocks for the two consecutive sliding-window calculations of the μ statistic illustrated in Figure 4. For the sake of simplicity, we assume that patterns were entered into the PATTERN POOL at adjacent pattern slots, thus ignoring the number of mutations per SNP that is used by HAM for hashing. Therefore, the indices for the five patterns in the figures are in the range [0..4]. We also introduce four more registers, $R3$ through $R6$, which correspond to the four operands in Equation 4: $\sum_{s_i \in W_L}[s_i \in P_L] = R3$, $\sum_{s_i \in W_R}[s_i \in P_R] = R4$, $\sum_{s_i \in W_L}[s_i \in P_L][s_i \notin P_R] = R5$, and $\sum_{s_i \in W_R}[s_i \notin P_L][s_i \in P_R] = R6$. The dedicated memory blocks and the registers are initialized by window $W_{W_{sz}-1}$, with $O(W_{sz}^2)$ time complexity. For every subsequent window W_i, both the memory blocks and the registers are simultaneously updated with $O(W_{sz})$ worst-case time complexity. A total of three SNPs collectively affect the state of each subwindow per sliding-window step: i) the first SNP in the left subwindow of W_{i-1} (pointed to by extra), ii) the last SNP in window W_i (pointed to by end), and iii) the first SNP in the right subwindow of window W_{i-1} (pointed to by mig), which migrates to the left subwindow with every step (see Figure 10).

The flowcharts depicted in Figure 11 describe the SUR rules and the order they are applied in order to update Le and $R3$ (patterns in subwindow W_L, Figure 11A), Ri and $R4$ (patterns in subwindow W_R, Figure 11B), xLe and $R5$ (exclusive patterns in subwindow W_L, Figure 11C), and xRi and $R6$ (exclusive patterns in subwindow W_R, Figure 11D). Prior to applying SUR for a window W_i, the four memory blocks are searched for the pattern indices that the aforementioned pointers (extra, mig, and end) indicate. The notations X_{ind}^{ptr} and X_{cnt}^{ptr} represent the results of this search, with X_{ind}^{ptr} denoting the index in memory block $X \in \{Le, Ri, xLe, xRi\}$ for the pattern indicated by pointer ptr, and X_{cnt}^{ptr} denoting the respective number of occurrences. When the pattern (that ptr points to) exists in X, the search returns $X_{ind}^{ptr} \geq 0$ and $X_{cnt}^{ptr} > 0$, otherwise the respective values are -1 and 0. A dedicated stack data structure per memory block stores the indices of the invalid

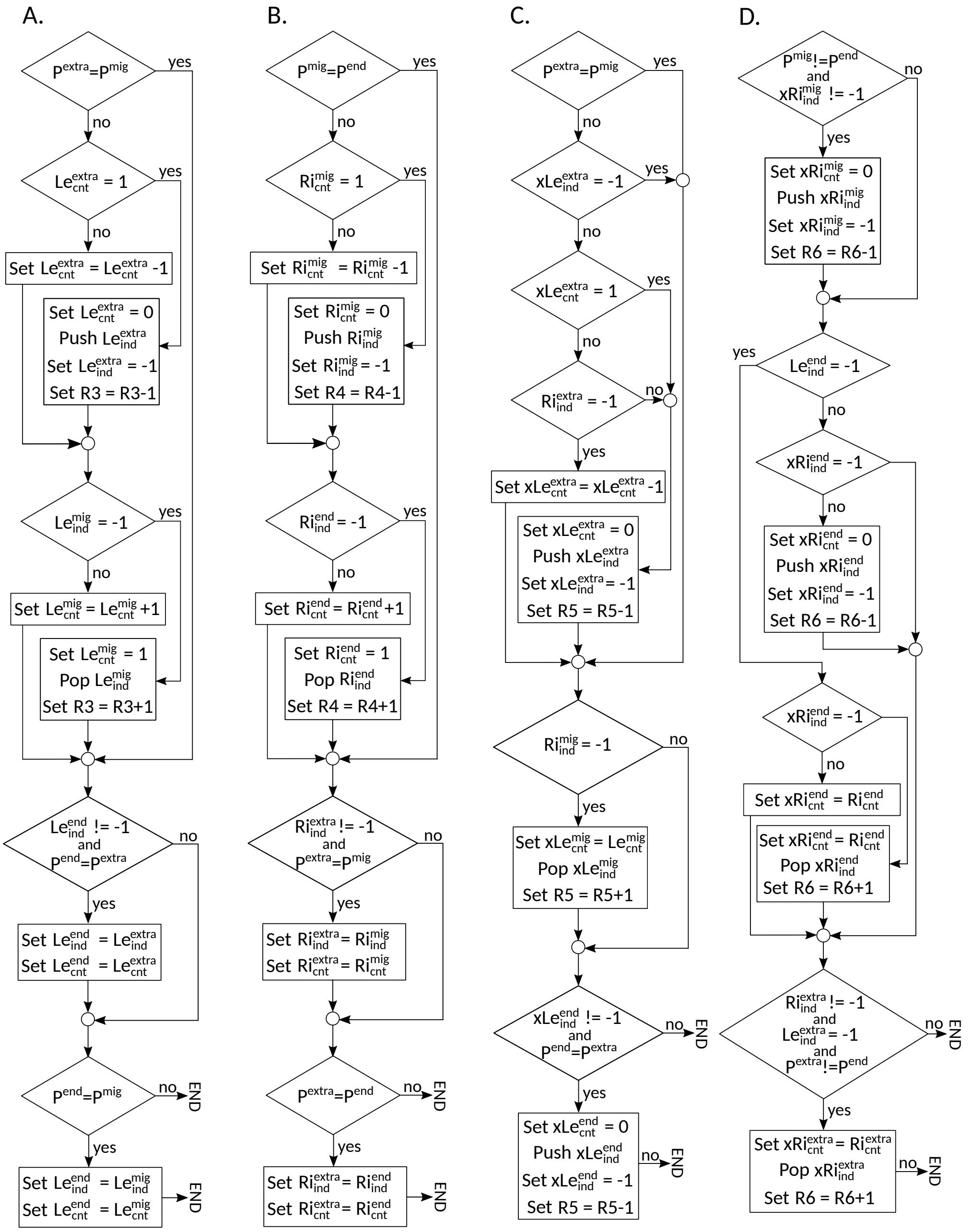

Figure 11. Flowcharts that describe the SUR rules employed for updating the four memory blocks and the four registers used for the evaluation of μ^{LD} (Equation 4). A) Steps to update the number of patterns in W_L, B) Steps to update the number of patterns in W_R, C) Steps to update the number of exclusive patterns in W_L, and D) Steps to update the number of exclusive patterns in W_R. Note that the push and pop operations refer to a dedicated stack per flowchart, and a pop operation, e.g., Pop Le^{mig}_{ind}, implies that the respective block entry is initialized with the pattern index that *mig* indicates.

memory-block entries at any point in time. New indices are pushed to the stack (push operation in Figure 11) when patterns slide out of the corresponding subwindow, and existing indices are removed from the stack (pop operation in Figure 11) when a new pattern enters the corresponding subwindow. For the memory block Le that holds the status of the left subwindow of W_1 in Figure 10, for instance, we have $Le^{extra}_{ind} = -1$, $Le^{extra}_{cnt} = 0$, $Le^{mig}_{ind} = 1$, $Le^{mig}_{cnt} = 3$, $Le^{end}_{ind} = -1$, $Le^{end}_{cnt} = 0$, and the corresponding stack holds the values 0 and 3, with the latter being at the top of the stack since the former was pushed to the stack first by W_0. The sole sequential dependency in employing SUR for μ^{LD} is that searching the memory blocks for the particular patterns should precede the application of the rules. Scanning and then updating a memory block and the corresponding register proceeds in parallel for all blocks, yielding overall time complexity of $O(W_{sz})$ per sliding step.

5.4 Decoupled Access/Execute Reconfigurable (DAER) architecture

The DAER [10] accelerator architecture (Figure 12) adopts a decoupled access/execute paradigm [11, 44] that facilitates system design, as well as code orchestration and development. It exhibits pairs of FETCH and PROCESS units that operate in tandem to implement a wide range of applications. A series of code preparation steps are required in order to decouple memory accesses from processing, and resolve dependencies. DAER resolves dependencies using a distributed memory-access scheme that serves requests concurrently through a network of FIFO-based interconnected FETCH units. Coupling the FETCH-unit network with multiple PROCESS units yields a dataflow engine that accommodates task-based applications with streaming and/or arbitrary access patterns. FETCH units communicate with a host processor to load execution parameters and memory traces, as well as with external on-board memory to retrieve/store data. PROCESS units receive input from external memory through FIFO-based links, and implement the required logic and/or arithmetic operations. FIFO-based links additionally facilitate the passing of intermediate results between neighboring processing units in a pipeline, as well as the synchronization among PROCESS units operating concurrently. The aforementioned code preparation steps can be applied on any algorithm that is implemented in a high-level programming language. The source code is primarily translated to a dataflow description, followed by an annotation step that employs high-level synthesis directives for interfacing and hardware generation. Employing a decoupled access/execute paradigm yields a distinct separation between parsing SNP data and applying SNP-based statistics. The resulting accelerator architecture exhibits a well-defined interface between the OoC algorithm and the PROCESS units via FIFO-based links that are controlled by FETCH units. Thus, repurposing the entire system for a different SNP-based statistic only requires to adapt the PROCESS units, whereas the rest of the system remains largely intact.

A PROCESS unit in RAiSD-X has two modes of operation which alternate in accordance with the computational demands of the OoC algorithm. When incoming SNPs are matched to patterns, the PROCESS unit is deployed for the brute-force search of the PATTERN POOL partitions that the HAM algorithm dictates per incoming SNP. In this mode of operation, which we dub BPS (Brute-force Partition Search), the PROCESS unit scans partition P_{M_i} (see Figure 7), and only proceeds to partition P_{S-M_i} (see Figure 7) if the previous scan did not match the incoming SNP to a pattern. If the second scan does not match the incoming SNP to a pattern either, the PROCESS unit notifies the FETCH unit to store the incoming SNP in partition P_{M_i} of the PATTERN POOL, which resides in the HMC. This is achieved through the $WrEn$ signal in Figure 12, which is generated by the BPS logic. When the PATTERN POOL is full, the OoC algorithm deploys the PROCESS unit to apply the required SUR steps for computing a μ-statistic score per sliding window. We refer to this mode of operation as SWS (Sliding-Window Steps). Both operating modes are served by the same FETCH unit, based on different configuration parameters for address generation, which are stored in a dedicated

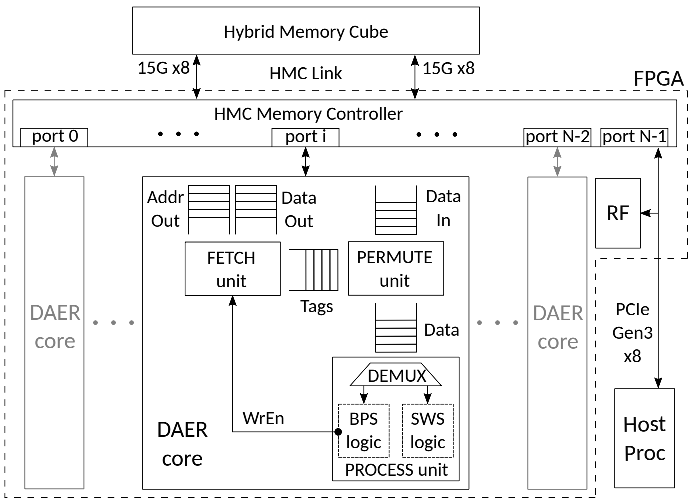

Figure 12. RAiSD-X system overview. Per OoC iteration, the host processor transfers a new SNP to the FPGA, which is temporarily stored in RF (Register File). Partition data are loaded from the HMC through FETCH units, and compared with the SNP by the BPS logic in each PROCESS unit. If the SNP is a new patten, the FETCH unit stores it in the correct partition of the `PATTERN POOL`. When the `PATTERN POOL` is full, the PROCESS units are deployed in SWS mode to scan the `REGION` structure and compute the μ statistic.

Register File (RF in Figure 12) by the host processor. In BPS mode, the FETCH unit retrieves patterns from the same partition in the `PATTERN POOL`, which are stored contiguously in memory. The interleaved memory layout of the `REGION` structure allows to fetch SNP data (represented by triplets of values, see Section 5.3) from contiguous memory locations when operating in SWS mode as well, thereby eliminating the need for a different FETCH-unit architecture. This reduces resources per FETCH-PROCESS pair, henceforth referred to as DAER core (Figure 12), and allows to instantiate multiple DAER cores to improve the aggregate hardware-platform performance. In BPS mode, DAER cores operate in tandem to accelerate the brute-force scan of the same partition in `PATTERN POOL`, whereas in SWS mode, each DAER core is assigned a fraction of the `REGION` structure and applies SUR steps independently. The number of DAER cores is bounded by the number of available memory ports in the system (N in Figure 12), since each port is exclusively assigned to a FETCH unit.

6 IMPLEMENTATION

To develop RAiSD-X, we initially introduced a series of proof-of-concept software functions into the latest release of the RAiSD source code (version 1.7, *https://github.com/alachins/raisd*), which allowed to extensively test for correctness (note that RAiSD-X and RAiSD produce identical results). More specifically, we modified the source code to construct the `REGION` structure employing the interleaved memory layout, thereby eliminating the need for an intermediate memory-layout transformation step. The `Pattern Match` routine (see Figure 5) was replaced with a HAM-enabled equivalent, and the SUR steps were implemented in the sliding-window algorithm that computes the μ statistic on the interleaved `REGION` structure. Thereafter, we applied a series of code transformations, dictated by the DAER framework (Section 5.4), to prepare the source code for hardware generation. A high-level synthesis tool (Xilinx Vivado HLS [53]) was employed to obtain the hardware description for the custom accelerators, i.e., the BPS logic for the HAM-enabled `Pattern`

Match, and the SWS logic for SUR. The complete FPGA system was implemented using Xilinx Vivado 2016.3 [17, 52].

We used a Linux-based desktop PC, the SC6-mini by Micron Technology, for implementation and verification purposes. It exhibits an Intel Core i7-5930K 6-core CPU running at 3.5GHz (host processor), 32GB of DDR4 main memory, and an AC-510 SuperProcessor module with 4GB of Hybrid Memory Cube (HMC) and a Xilinx Kintex UltraScale XCVU060 FPGA (hardware platform). The maximum theoretical bandwidth that the HMC interface exposes to the user logic on the FPGA is 30GB/sec per direction, through 10× 128-bit-wide memory ports operating at 187.5MHz. Communication between the host processor and the FPGA, and subsequently the HMC, is achieved through PCI Express, occupying one of the HMC ports, as depicted in Figure 12. A dedicated permutation unit (denoted PERMUTE in the figure) is inserted between every HMC port and the respective FIFO that provides input to the PROCESS unit in a DAER core, to permute the data loaded from the HMC according to a series of tags provided by the FETCH unit. This is because the HMC controller serves requests out of order, whereas the PROCESS unit expects data in the order that the associated memory requests were issued by the FETCH unit.

Table 1 provides resource utilization and performance (datapath latency in clock cycles, and throughput in results/cycle) for RAiSD-X when one and nine DAER cores are instantiated, as well as for the optimized implementation of our previous accelerator platform [4] (henceforth referred to as Accel1) with one and three accelerator instances. The table also provides resource utilization and performance for the complete FPGA systems, i.e., including the supportive infrastructure (controllers) for the HMC and PCI Express. Recall that, the optimized design of Accel1 occupies three HMC ports per accelerator instance, restricting the number of accelerator instances to three. Enabled by the interleaved memory layout and SUR, each DAER core in RAiSD-X occupies a single HMC port, which allows to instantiate up to nine DAER cores, and consequently boost the aggregate accelerator-platform performance.

Table 1. Resource utilization (%) and performance of RAiSD-X with one and nine DAER cores (DRC) on the Kintex UltraScale FPGA, in comparison with the optimized implementation of our first accelerator platform [4] (Accel1) with one and three accelerator instances (ACC). The table also provides resource utilization and performance for the complete FPGA systems, i.e., including the HMC and PCI Express controllers. Note the varying bandwidth requirements of the DAER cores based on the different mode operations: SWS for the sliding-window operations, and BPS for the brute-force pattern matching within the PATTERN POOL partitions.

Resources (Total)	Accel1	Accel1 Full System		RAiSD-X	RAiSD-X Full System	
/ Performance	ACC × 1	ACC × 1	ACC × 3	DRC × 1	DRC × 1	DRC × 9
LUTs (331,680)	7.84%	25.86%	41.79%	4.11%	24.75%	62.24%
FFs (663,360)	4.59%	20.23%	30.93%	1.81%	17.32%	31.35%
BRAMs(1,080)	1.85%	23.19%	33.47%	1.00%	12.95%	29.33%
DSPs (2,760)	0.58%	0.58%	1.74%	0.51%	0.51%	4.57%
I/O BW (MB/sec)	214	214	642	367(SWS) 2,794(BPS)	367(SWS) 2,794(BPS)	3,303(SWS) 25,146(BPS)
Latency (clk cycles)	187	187	187	422	422	422
Throughput (res/clk)	1/14	1/14	3/14	1/30	1/30	9/30
Frequency (MHz)	187.5	250	250	250	250	250

In RAiSD-X, the host processor performs the first two steps of the OoC algorithm, i.e., SNP Filtering and Binary Conversion (see Figure 5), as well as the preliminary part of HAM, which entails the indexing of the partitionAddress and partitionSize arrays (see Figure 7). When the addresses and sizes of the selected partitions are stored in the RF register (see Figure 12), through PCI Express, the DAER cores operate in parallel to brute-force search the two partitions (the PROCESS units operate in BPS mode). Based on the result of the partition scans, the host processor

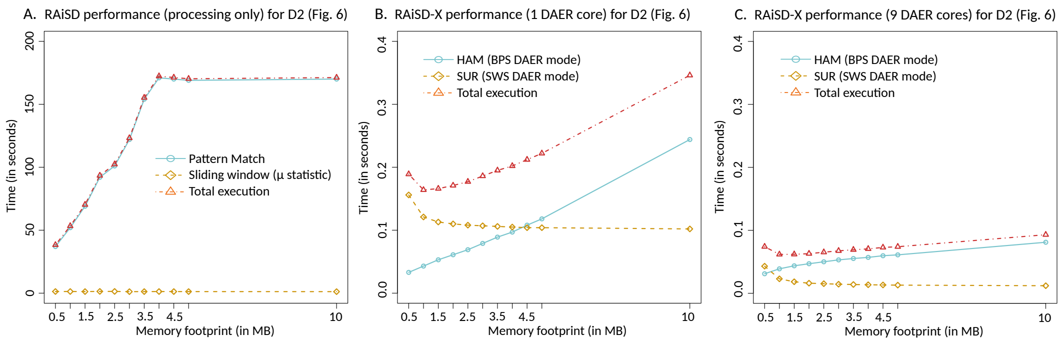

Figure 13. The effect of the PATTERN POOL memory size on the performance of RAiSD (A) and RAiSD-X with 1 (B) and 9 (C) DAER cores. As the memory footprint increases, the number of OoC iterations, and consequently the number of accelerator calls to DAER cores operating in SWS mode, decreases. The performance of finding patterns decreases with an increasing PATTERN POOL size in both RAiSD (software) and RAiSD-X (hardware). The sliding-window computation is not affected by the memory size when performed in software, whereas accelerator performance improves as the number of OoC iterations (accelerator calls) decreases.

constructs the REGION structure directly on the HMC without the intervention of the hardware (through PCI Express). When the PATTERN POOL is full, the host processor stores the address and the size of the REGION structure into the RF, and the DAER cores are deployed in SWS mode to compute the μ statistic values and their respective locations. The results are directly transferred to the host processor through the PCI Express link, thereby allowing to employ the HMC ports solely for loading input data. These steps are repeated (as dictated by the OoC algorithm) until the entire input dataset is parsed and processed.

As briefly mentioned in Section 5.2, the number of OoC iterations can potentially affect the performance of the DAER cores when operating in SWS mode (calculation of μ statistic scores). When the PATTERN POOL is not sufficiently large to fit all the patterns in the data, multiple calls to DAER cores operating in SWS mode are required, thereby incurring additional accelerator initiation overhead and deteriorating accelerator performance for the sliding-window computation. Figure 13 illustrates this effect for the simulated dataset D2 of Figure 6, which comprises 1,000 samples and 1,457,456 SNPs. As can be observed in Figures 13B (1 DAER core) and 13C (9 DAER cores), HAM execution times increase and SUR execution times decrease with an increasing PATTERN POOL memory footprint, yielding the fastest overall execution for this particular dataset when a 1MB PATTERN POOL is used. When more than one OoC iteration is required in order to analyze a given dataset, the optimal PATTERN POOL memory footprint varies with dataset characteristics such as the sample size and the number of SNPs, and is therefore dataset-dependent. Hence, to yield a generic hardware solution that performs well without considerably increasing the memory requirements, which is one of RAiSD's main goals, RAiSD-X allocates the minimum memory size required by the HAM algorithm to operate. Given that the overall RAiSD (software only) performance is dominated by the Pattern Match routine as the sample size increases, and the PATTERN POOL memory footprint has negligible effect on the sliding-window computation (Figure 13A), opting for the minimum operating PATTERN POOL memory footprint in RAiSD-X allows for a fair performance evaluation (Section 7), since both RAiSD and RAiSD-X exhibit comparable memory requirements, and RAiSD performs best with small PATTERN POOL sizes due to its brute-force pattern searching algorithm. Note that, all RAiSD-X results in this study are obtained using a 0.5MB PATTERN POOL, except for the analysis of the real data presented in Section 7.3.

7 PERFORMANCE EVALUATION

7.1 Experimental setup

We initially assess performance of the HAM algorithm and the SUR set of update rules at the software level, by backporting them to RAiSD (version 1.7), as well as at the hardware level, by examining the performance of the PROCESS units when operating in BPS (for HAM) and SWS (for SUR) modes. Thereafter, we compare RAiSD-X with Accel1 (our previous accelerator system [4]) and a series of state-of-the-art software tools: i) the parallel tool OmegaPlus [1], serving as the reference implementation for the LD-based sweep signature, ii) the sequential tool SweepFinder2 [14] and the parallel tool SweeD [36], for the SFS-based signature, and iii) the sequential tool RAiSD [2] that captures multiple sweep signatures with the μ statistic. As a test platform for the software benchmarks, we deploy a Dell PowerEdge R530 rack server with two 10-core Intel Xeon E5-2630v4 CPUs (20 threads per CPU) running at 2.2GHz (base), and 128GB of DDR4 main memory. For all performance comparisons, we report total execution times per software or hardware solution, including disk access (file parsing to fetch SNPs), processing, and generation of output reports. Although this does not represent a direct comparison of processing capacity between the involved technologies (multi-core CPU versus FPGA+HMC), it provides a realistic notion of the expected performance and scalability gains from the deployment of a complete FPGA-accelerated solution in population genetics analyses.

7.2 Simulated genomes

We initially employ a series of simulated datasets with increasing sample size (20, 50, and 100 sequences, available at *https://tinyurl.com/y56mkxnp*) and a varying number of SNPs (in the order of some thousands per set of SNPs, between approximately 6,500 and 8,500 SNPs). The exact number of SNPs is determined by the simulation software and depends on the evolutionary history of the population and the strength of selection. We use the ms [22] tool for neutral simulations, and the mssel tool (kindly provided by R.R. Hudson) for datasets with selection. To resemble realistic execution scenarios, which typically require a large number of software invocations to conduct a statistical analysis and calculate a cut-off threshold to distinguish between selection and neutrality, each dataset comprises 1,000 neutral sets of SNPs, and 1,000 sets of SNPs with selection at the center of the simulated genomic region.

Table 2 provides a breakdown of RAiSD execution times per dataset, along with execution times of the proposed solutions when implemented in software (backported to RAiSD version 1.7) and in hardware (DAER cores). As can be observed in the table, RAiSD-X is up to 151 times faster than the sequential execution of RAiSD for the pattern matching stage, and up to 169 times faster for the sliding-window operations. When considering the entire execution, which includes the disk access time for loading data and storing the final report, RAiSD-X is up to 4.3x faster than RAiSD.

Table 3 shows performance of RAiSD and RAiSD-X for the dataset with 20 samples when the window size increases to 500 SNPs. As observed in the table, RAiSD performance deteriorates dramatically, whereas RAiSD-X remains practically unaffected by the increasing window size, achieving up to 75 times overall faster execution. The rest of the results presented in this article have been obtained using a window size of 10 SNPs (default RAiSD value), unless specified otherwise. Note also that the Accel1 hardware accelerator is designed to operate on 10-SNP windows.

Table 4 provides execution times for the sequential analysis of the 2,000 sets of SNPs per dataset using SweepFinder2 (SF2), SweeD (SD), OmegaPlus (OP), and RAiSD, as well as for the FPGA-accelerated analysis of the same datasets with Accel1 and RAiSD-X, deploying three and nine accelerator cores, respectively. Note that each Accel1 accelerator core occupies three memory ports, one per μ statistic factor (sweep signature), which restricts the maximum number of accelerator

Table 2. Breakdown of RAiSD execution times (in seconds) for the OoC and the sliding-window algorithms, and execution times of HAM and SUR when backported to RAiSD version 1.7, as well as when implemented in hardware with 1 and 9 DAER cores for the three simulated datasets with increasing sample size (RSD: RAiSD v1.7 on the Dell rack server, HAM+SUR: RAiSD v1.7 with HAM+SUR on the Dell rack server, DAER1: RAiSD-X with 1 FETCH-PROCESS pair on the SC6-mini, DAER9: RAiSD-X with 9 FETCH-PROCESS pairs on the SC6-mini). The total execution time includes the disk access time for loading the ms files and storing the final report.

Sample	Out-of-Core algorithm				Sliding-window algorithm				Total execution time			
size	RSD	HAM	DAER1	DAER9	RSD	SUR	DAER1	DAER9	RSD	HAM+SUR	DAER1	DAER9
20	33.2	3.5	2.7	0.9	11.2	10.1	0.9	0.1	59.8	28.9	16.6	13.9
50	82.8	5.4	3.2	0.9	13.2	11.9	1.1	0.1	130.7	51.9	34.6	31.4
100	151.3	8.4	3.7	1.0	16.9	13.8	1.3	0.1	248.8	102.8	68.2	64.3

Table 3. Breakdown of RAiSD execution times (in seconds) for the OoC and the sliding-window algorithms, and execution times of HAM and SUR when backported to RAiSD version 1.7, as well as when implemented in hardware with 1 and 9 DAER cores for the 20-sample simulated dataset with increasing window size (RSD: RAiSD v1.7 on the Dell rack server, HAM+SUR: RAiSD v1.7 with HAM+SUR on the Dell rack server, DAER1: RAiSD-X with 1 FETCH-PROCESS pair on the SC6-mini, DAER9: RAiSD-X with 9 FETCH-PROCESS pairs on the SC6-mini). The total execution time includes the disk access time for loading the ms files and storing the final report.

Window	Out-of-Core algorithm				Sliding-window algorithm				Total execution time			
size	RSD	HAM	DAER1	DAER9	RSD	SUR	DAER1	DAER9	RSD	HAM+SUR	DAER1	DAER9
10	33.2	3.5	2.7	0.9	11.2	10.1	0.9	0.1	59.8	28.9	16.6	13.9
100	33.3	3.4	2.7	0.9	75.8	15.6	0.9	0.1	124.8	34.7	16.6	13.9
500	33.3	3.4	2.7	0.9	1008.1	37.8	1.0	0.2	1057.2	56.9	16.7	14.1

cores that can be deployed on the SC6-mini to three. As can be observed in Table 4, RAiSD-X is up to 1,755, 1,051, 1,271, and 4.3 times faster than SweepFinder2, SweeD, OmegaPlus, and RAiSD, respectively, as well as up to 2.3 times faster than Accel1. Table 4 also provides a qualitative comparison of the implemented methods, based on detection accuracy, which is measured as the percentage of reported sweep locations (best-score location per SNP set with selection) at a distance less than 1% of the size of the simulated genomic region. All runs assumed a genomic region size of 100,000 nucleotide bases. Thus, selection occurred at genomic location 50,000, with a detection outcome considered accurate if the best-score location is in the range [49,001-50,999]. For the dataset with a sample size of 20 sequences, for instance, RAiSD-X accurately reported 626 sweep locations (out of 1,000), whereas SF2, SD, and OP reported 297, 298, and 488, respectively. Note that, the column titled "μ statistic" provides the attained detection accuracy of RAiSD-X, Accel1, and RAiSD, since all three generate identical results relying on the μ statistic.

Table 4. Sequential execution times (in seconds) and detection accuracy of SweepFinder2 (SF2), SweeD (SD), OmegaPlus (OP), RAiSD, Accel1, and RAiSD-X. The column titled "μ statistic" provides the attained detection accuracy for all μ-statistic-based implementations, i.e., RAiSD, Accel1, and RAiSD-X. The table also provides total times for the parallel execution of SweeD (SD-40) and OmegaPlus (OP-40) with 40 threads.

Sample	Execution time								Detection accuracy (%)			
size	SF2	SD	SD-40	OP	OP-40	RAiSD	Accel1	RAiSD-X	SF2	SD	OP	μ statistic
20	24,571	14,721	1,075	17,807	2,283	60	32	14	29.7	29.8	48.8	62.6
50	29,814	14,733	888	27,567	4,241	131	61	31	42.3	42.5	67.0	74.1
100	61,905	15,758	1,110	41,144	4,957	249	104	64	45.8	46.1	75.6	75.9

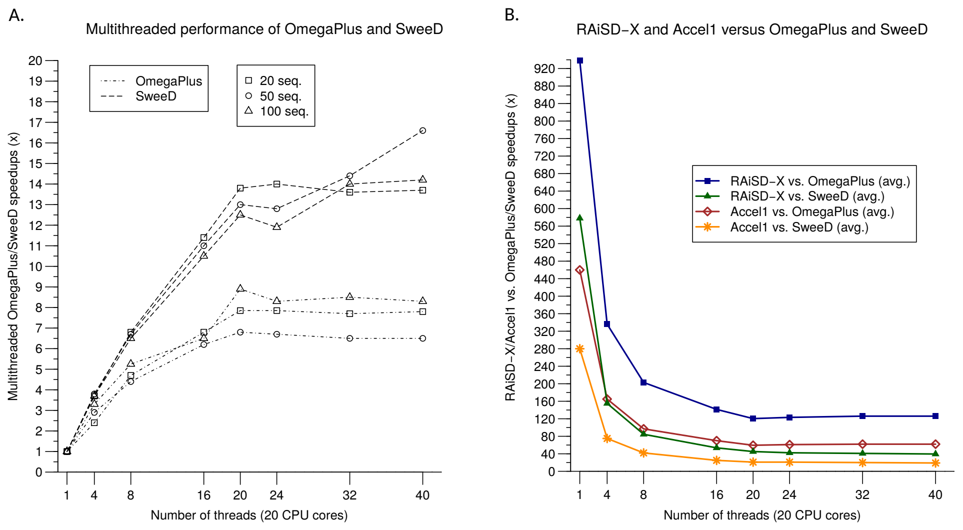

Figure 14. Performance evaluation of RAiSD-X, Accel1 [4], SweeD, and OmegaPlus based on datasets with 20, 50, and 100 sequences (2,000 sets of SNPs per dataset). A) Multithreaded versus sequential execution of SweeD and OmegaPlus. B) Average performance of the FPGA-accelerated systems Accel1 and RAiSD-X in comparison with SweeD and OmegaPlus (FPGA vs. multi-core CPU speedups).

When the aggregate processing capacity of modern processors is exploited by SweeD and OmegaPlus, through the deployment of multiple cores (none of the tools employ vector intrinsics), shorter overall execution times are achieved. Figure 14A illustrates how the tools scale with an increasing number of threads (up to 40) on 20 CPU cores. Still, RAiSD-X remains 124 and 40 times (on average over all runs) faster than OmegaPlus and SweeD, respectively, as illustrated in Figure 14B, which provides a performance comparison of RAiSD-X with the parallel execution of SweeD and OmegaPlus. The plot additionally shows the average attained speedups of Accel1 over OmegaPlus and SweeD, which are 62x and 20x, respectively.

It should be noted that, all software tools except for RAiSD require a user-defined parameter that determines how exhaustively to scan a dataset. This is the $-s$ parameter for SweepFinder2, and the $-grid$ parameter for both SweeD and OmegaPlus. This parameter specifies how many genomic locations along a chromosome should be evaluated per run. To yield comparable results, each tool evaluated 1,000 genomic locations. Evidently, the computational load increases proportionally with the number of evaluation points, introducing a trade-off between detection accuracy and analysis time, the examination of which is beyond the scope of this paper. The accuracy measurements provided in Table 4 are obtained using the same number of evaluation points for all tools (1000). Neither RAiSD, nor the FPGA-accelerated systems Accel1 and RAiSD-X require an additional parameter for this purpose, since the number of sliding-window steps is determined by the number of SNPs in the dataset.

7.3 The human genome (1000 Genomes project)

To assess performance on real data, we analyzed the human chromosomes 18 to 22, available by the 1000 Genomes project [45], which sequenced 2,504 genomes. Due to the fact that humans are diploid organisms, the experimental sample size for sweep detection is $2,504 \times 2 = 5,008$ sequences. We initially examined what fraction of RAiSD execution time is spent on file parsing, the OoC algorithm for pattern matching, and the μ statistic computations for the analysis of chromosome 22. We observed that, when the sliding window size is set to 10 SNPs, up to 90% of the total time is spent on file parsing. This is due to the fact that the VCF file format, which is widely employed for real data, is highly generic and can be used for a variety of biological studies, therefore it includes a considerable amount of irrelevant information for sweep detection. To further improve performance, RAiSD-X exhibits a dedicated file parser for compressed VCF files, which allows to reduce the time spent on file parsing and achieve higher speedups. Table 5 provides a time breakdown of RAiSD and RAiSD-X for the analysis of chromosome 22 when the window size increases to 1,000 SNPs.

Table 5. Breakdown of RAiSD and RAiSD-X execution times (in seconds) for the analysis of the human chromosome 22 as the window size increases to 1000 SNPs. RAiSD-X employs a dedicated parser for compressed VCF files. Note that, RAiSD-X allocates 100MB of memory for the `PATTERN POOL` (required by HAM because of the high sample size which reduces the partition size), whereas RAiSD allocates 1MB for the same purpose.

Window	VCF Parsing		Pattern Matching		μ statistic		Total Execution		Speedup (x)
size	RAiSD	RAiSD-X	RAiSD	RAiSD-X	RAiSD	RAiSD-X	RAiSD	RAiSD-X	RAiSD-X vs. RAiSD
10	314	141	31.7	0.11	1	0.012	346.7	141.1	2.5x
100	318	142	33.9	0.12	10.7	0.038	362.6	142.2	2.5x
500	313	150	36.7	0.20	173.7	1.204	523.4	151.4	3.5x
1000	315	150	41.5	0.22	648.9	2.064	1,005.4	152.3	6.6x

Table 6 provides the number of SNPs per chromosome, total execution times (including file parsing) for RAiSD (1 thread), OmegaPlus and SweeD when 1 and 40 threads (20 cores) are used, as well as the FPGA-based systems Accel1 and RAiSD-X, and the respective speedups. SweepFinder2 failed to analyze the data due to the large sample size, terminating abruptly on a failing assertion in all execution attempts (SweepFinder2.c:738: get_pstar). The remaining software tools (apart from RAiSD) calculated scores at $10,000$ evaluation points, whereas the FPGA-based systems and RAiSD evaluated two orders of magnitude more points along each chromosome. This is due to the sliding-window algorithm, which evaluates a SNP-dependent number of windows, X, with $X = D_{sz} - W_{sz} + 1$, where D_{sz} is the number of SNPs, and W_{sz} is the window size, set to 10 SNPs for all runs in this study.

The number of iterations of the OoC algorithm (denoted "OoC It." in Table 6) is a result of the out-of-core operation of RAiSD, Accel1, and RAiSD-X, which is deployed to efficiently process large-scale SNP data by maintaining a bounded memory footprint, regardless of the chromosome size. This number translates to the total number of accelerator calls for the FPGA-accelerated systems (PROCESS units operating in SWS mode in RAiSD-X). The high number of RAiSD-X accelerator invocations is an inevitable consequence of the large sample size, which defines the number of partitions that the HAM algorithm employs in the `PATTERN POOL`. It was empirically determined that allocating 100 MBs of memory for the `PATTERN POOL`, given the sample size of 5,008 sequences, yields the highest RAiSD-X performance despite the high number of OoC iterations. Recall that, unlike RAiSD and Accel1, a `PATTERN POOL` in RAiSD-X is considered full when any of the partitions becomes full. A larger sample size leads to a smaller partition size, thus requiring more OoC iterations for the same dataset size.

Finally, we used RAiSD-X and RAiSD to analyze the entire human genome (we excluded the sex chromosomes). RAiSD-X required 2 hours and 54 minutes for the 22 autosomes, while RAiSD required 7 hours and 42 minutes. Given the high-quality SNP data that the 1000 Genomes project [45] made available, one can estimate the total required time to scan the 22 autosomes, which comprise a total of $77,832,252$ SNPs, using SweeD and OmegaPlus. Based on the observed execution times and

Table 6. Performance evaluation based on the analysis of the human autosomes 18 to 22 from 2,504 individuals [45]. The window size for RAiSD, Accel1, and RAiSD-X is 10 SNPs. All times are total execution times including the time for parsing the input file in VCF format and the time required to create the final reports.

	Chromosome	18	19	20	21	22
	Size (SNPs)	2,171,507	1,747,024	1,733,485	1,049,984	1,051,673
CPU	OP-1	3h50m	3h27m	3h8m	2h1m	2h5m
	OP-40	32.2m	29.1m	26.8m	16.5m	17.9m
	SD-1	3h3m	2h38m	2h36m	1h53m	1h41m
	SD-40	31.5m	25.5m	25.7m	17m	16.5m
	RAiSD	12.8m	10.3m	9.5m	7.9m	5.8m
FPGA	Accel1	6.9m	5.6m	5.6m	3.4m	3.4m
	RAiSD-X	4.8m	3.9m	3.9m	2.3m	2.3m
FPGA vs. CPU	Accel1 vs. OP-1	33.3x	36.9x	33.6x	35.6x	36.8x
	vs. OP-40	4.7x	5.2x	4.8x	4.9x	5.3x
	vs. SD-1	26.5x	28.2x	27.9x	33.2x	29.7x
	vs. SD-40	4.5x	4.6x	4.6x	5x	4.9x
	vs. RAiSD	1.9x	1.8x	1.7x	2.3x	1.7x
	RAiSD-X vs. OP-1	47.9x	53.1x	48.2x	52.6x	54.3x
	vs. OP-40	6.7x	7.5x	6.9x	7.2x	7.8x
	vs. SD-1	38.1x	40.5x	40x	49.1x	43.9x
	vs. SD-40	6.6x	6.5x	6.6x	7.4x	7.2x
	vs. RAiSD	2.6x	2.6x	2.4x	3.4x	2.5x
OoC It.	RAiSD	1,104	904	890	541	548
	Accel1	1,104	904	890	541	548
	RAiSD-X	45,246	39,386	37,384	23,558	22,630

the number of SNPs per chromosome, we calculated that when SweeD and OmegaPlus deploy 20 CPU cores (40 threads), they process 1,112.1 and 1,054.9 SNPs per second, respectively. Thus, SweeD would require 19 hours and 26 minutes for this analysis, while OmegaPlus would require 20 hours and 29 minutes. Note that, assessing the accuracy of the detection methods is only possible when the location of the selected locus is known *a priori*, i.e., when using simulated data (the location of the selected locus is an input parameter to the simulation tool). Since the true locations of selected loci in real genomes are not known, a qualitative comparison between detection methods based on real data would rely on the outcome of a reference detection method, raising a respective question for the accuracy of the reference method itself. Alachiotis and Pavlidis [2] report, for a series of genes that have been detected using RAiSD, what has been discovered in the literature (based on other methods).

8 CONCLUSIONS

In this work, we presented RAiSD-X, an optimized FPGA-based accelerator system for the fast and accurate detection of positive selection in large-scale SNP data. RAiSD-X extends our previous work in accelerating positive selection inference with the introduction of two new algorithms and the complete redesign of the employed Decoupled Access-Execute (DAE) accelerator architecture. The Hash-based Algorithm based on Mutations (HAM) reduced the worst-case time complexity for detecting patterns in the data, while the Set of Update Rules (SUR) for the sliding-window calculation of the μ statistic, in combination with a transformed memory layout, allowed to reduce FPGA resources per accelerator core. This paved the way for increasing parallelism on the device, via the instantiation of more FETCH-PROCESS units, while introducing additional functionality within the PROCESS units, thereby boosting the aggregate system performance. A PROCESS unit can assume two roles, depending on the stage of the application, i.e., either conducting brute-force SNP comparisons as required by HAM, or applying the necessary update rules as dictated by SUR. We compared RAiSD-X with state-of-the-art software implementations, observing up to three orders of magnitude faster execution times.

As future work, we intend to explore the potential of custom in-memory architectures for further performance improvements. Furthermore, given the highly parallel nature of the employed algorithms, it is interesting to explore how RAiSD-X performance scales when FPGA systems with High Bandwidth Memory [23] (bandwidth up to 500GB/sec) are employed, as well as multiple FPGA boards. Distributing the workload to multiple FPGAs at the granularity at which RAiSD-X currently partitions the μ statistic calculations, however, will yield a less favorable computation-to-communication ratio. Therefore, coarser-grained parallelization needs to be explored, e.g., by partitioning the workload at the SNP-set level, rather than at the SNP level, as currently performed in RAiSD-X.

REFERENCES

[1] Nikolaos Alachiotis et al. 2012. OmegaPlus: a scalable tool for rapid detection of selective sweeps in whole-genome datasets. *Bioinf.* 28, 17 (2012), 2274–2275.

[2] Nikolaos Alachiotis and Pavlos Pavlidis. 2018. RAiSD detects positive selection based on multiple signatures of a selective sweep and SNP vectors. *Communications biology* 1, 1 (2018), 79.

[3] Nikolaos Alachiotis, Thom Popovici, and Tze Meng Low. 2016. Efficient computation of linkage disequilibria as dense linear algebra operations. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International.* IEEE, 418–427.

[4] Nikolaos Alachiotis, Charalampos Vatsolakis, Grigorios Chrysos, and Dionisios Pnevmatikatos. 2018. Accelerated Inference of Positive Selection on Whole Genomes. In *Field Programmable Logic and Applications (FPL), 2018 International Conference on.* IEEE, 1–4.

[5] Nikolaos Alachiotis and Gabriel Weisz. 2016. High Performance Linkage Disequilibrium: FPGAs Hold the Key. In *ACM/SIGDA FPGA2016.* ACM, 118–127.

[6] Md Tauqeer Alam et al. 2011. Selective sweeps and genetic lineages of Plasmodium falciparum drug-resistant alleles in Ghana. *J. of Infectious Diseases* 203, 2 (2011), 220–227.

[7] Dimitrios Bozikas et al. 2017. Deploying FPGAs to Future-proof Genome-wide Analyses based on Linkage Disequilibrium. In *FPL2017.* IEEE, 1–4.

[8] J M Braverman et al. 1995. The hitchhiking effect on the site frequency spectrum of DNA polymorphisms. *Genetics* 140, 2 (June 1995), 783–96.

[9] Christopher C Chang, Carson C Chow, Laurent CAM Tellier, Shashaank Vattikuti, Shaun M Purcell, and James J Lee. 2015. Second-generation PLINK: rising to the challenge of larger and richer datasets. *Gigascience* 4 (2015).

[10] George Charitopoulos, Charalampos Vatsolakis, Grigorios Chrysos, and Dionisios Pnevmatikatos. 2018. A Decoupled Access-Execute Architecture for Reconfigurable Accelerators. In *Proceedings of the Computing Frontiers Conference.* ACM.

[11] Tao Chen and G Edward Suh. 2016. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *MICRO2016.* IEEE, 1–12.

[12] Jessica L Crisci, Yu-Ping Poh, Shivani Mahajan, and Jeffrey D Jensen. 2013. The impact of equilibrium assumptions on tests of selection. *Frontiers in genetics* 4 (2013).

[13] Natasja G De Groot and Ronald E Bontrop. 2013. The HIV-1 pandemic: does the selective sweep in chimpanzees mirror humankind's future? *Retrovirology* 10, 1 (2013), 53.

[14] Michael DeGiorgio et al. 2016. Sweepfinder2: increased sensitivity, robustness and flexibility. *Bioinformatics* 32, 12 (2016), 1895–1897.

[15] F Depaulis and M Veuille. 1998. Neutrality tests based on the distribution of haplotypes under an infinite-site model. *Molecular biology and evolution* 15, 12 (Dec. 1998), 1788–1790. http://www.ncbi.nlm.nih.gov/pubmed/9917213

[16] J C Fay and C I Wu. 2000. Hitchhiking under positive Darwinian selection. *Genetics* 155, 3 (July 2000), 1405–13. http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1461156&tool=pmcentrez&rendertype=abstract

[17] Tom Feist. 2012. Vivado Design Suite. *Xilinx, Inc., White Paper* (2012), 30.

[18] Genomics England. 2016. The 100,000 genomes project. 100 (2016), 0–2.

[19] Phillip B Gibbons and Srikanta Tirthapura. 2002. Distributed streams algorithms for sliding windows. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 63–72.

[20] Sharon R Grossman, Ilya Shylakhter, Elinor K Karlsson, Elizabeth H Byrne, Shannon Morales, Gabriel Frieden, Elizabeth Hostetter, Elaine Angelino, Manuel Garber, Or Zuk, et al. 2010. A composite of multiple signals distinguishes causal variants in regions of positive selection. *Science* 327, 5967 (2010), 883–886.

[21] WG Hill and Alan Robertson. 1968. Linkage disequilibrium in finite populations. *Theoretical and Applied Genetics* 38, 6 (1968), 226–231.

[22] Richard R Hudson. 2002. Generating samples under a Wright-Fisher neutral model of genetic variation. *Bioinformatics (Oxford, England)* 18, 2 (2002), 337–8.

[23] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*. IEEE, 1–4.

[24] Yuseob Kim and Rasmus Nielsen. 2004. Linkage disequilibrium as a signature of selective sweeps. *Genetics* 167, 3 (July 2004), 1513–1524. https://doi.org/10.1534/genetics.103.025387

[25] Motoo Kimura. 1969. The number of heterozygous nucleotide sites maintained in a finite population due to steady flux of mutations. *Genetics* 61, 4 (1969), 893.

[26] RC Lewontin. 1964. The interaction of selection and linkage. I. General considerations; heterotic models. *Genetics* 49, 1 (1964), 49.

[27] Heng Li et al. 2009. The sequence alignment/map format and SAMtools. *Bioinformatics* 25, 16 (2009), 2078–2079.

[28] Atabak Mahram and Martin C Herbordt. 2015. NCBI BLASTP on high-performance reconfigurable computing systems. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7, 4 (2015), 33.

[29] Anna-Sapfo Malaspinas. 2016. Methods to characterize selective sweeps using time serial samples: an ancient DNA perspective. *Molecular ecology* 25, 1 (2016), 24–41.

[30] J Maynard Smith and J Haigh. 1974. The hitch-hiking effect of a favourable gene. *Genetical research* 23, 1 (Feb. 1974), 23–35.

[31] Aaron McKenna et al. 2010. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome research* 20, 9 (2010), 1297–1303.

[32] Rasmus Nielsen et al. 2005. Genomic scans for selective sweeps using SNP data. *Genome Research* 15, 11 (Nov. 2005), 1566–1575. https://doi.org/10.1101/gr.4252305

[33] Rasmus Nielsen, Joshua S Paul, Anders Albrechtsen, and Yun S Song. 2011. Genotype and SNP calling from next-generation sequencing data. *Nature reviews. Genetics* 12, 6 (2011), 443.

[34] Rasmus Nielsen, Scott Williamson, Yuseob Kim, Melissa J Hubisz, Andrew G Clark, and Carlos Bustamante. 2005. Genomic scans for selective sweeps using SNP data. *Genome research* 15, 11 (Nov. 2005), 1566–75. https://doi.org/10.1101/gr.4252305

[35] Tomoko Ohta. 1996. The neutral theory is dead. The current significance and standing of neutral and nearly neutral theories. *BioEssays* 18, 8 (1996), 673–677.

[36] Pavlos Pavlidis et al. 2013. SweeD: likelihood-based detection of selective sweeps in thousands of genomes. *Molecular biology and evolution* (2013), mst112.

[37] Pavlos Pavlidis and Nikolaos Alachiotis. 2017. A survey of methods and tools to detect recent and strong positive selection. *Journal of Biological Research-Thessaloniki* 24, 1 (2017), 7.

[38] Pavlos Pavlidis, Jeffrey D Jensen, and Wolfgang Stephan. 2010. Searching for footprints of positive selection in whole-genome SNP data from nonequilibrium populations. *Genetics* 185, 3 (July 2010), 907–22. https://doi.org/10.1534/genetics.110.116459

[39] John E Pool, Ines Hellmann, Jeffrey D Jensen, and Rasmus Nielsen. 2010. Population genetic inference from genomic sequence variation. *Genome research* 20, 3 (2010), 291–300.

[40] Carlos Reaño, Javier Prades, and Federico Silla. 2018. Exploring the Use of Remote GPU Virtualization in Low-Power Systems for Bioinformatics Applications. In *Proceedings of the 47th International Conference on Parallel Processing Companion*. ACM, 8.

[41] David Salomon. 2004. *Data compression: the complete reference*. Springer Science & Business Media.

[42] Daniel R Schrider and Andrew D Kern. 2016. S/HIC: robust identification of soft and hard sweeps using machine learning. *PLos Genet.* 12, 3 (2016), e1005928.

[43] Stephan C Schuster. 2007. Next-generation sequencing transforms today's biology. *Nature methods* 5, 1 (2007), 16.

[44] James E Smith. 1982. Decoupled access/execute computer architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 10. IEEE Computer Society Press, 112–119.

[45] Peter H Sudmant et al. 2015. An integrated map of structural variation in 2,504 human genomes. *Nature* 526, 7571 (2015), 75.

[46] A Surendar. 2017. FPGA based parallel computation techniques for bioinformatics applications. *International Journal of Research in Pharmaceutical Sciences* 8, 2 (2017), 124–128.

[47] F. Tajima. 1989. Statistical Method for Testing the Neutral Mutation Hypothesis by DNA Polymorphism. *Genetics* 123, 3 (Nov. 1989), 585–595. http://www.genetics.org/cgi/content/abstract/123/3/585

[48] Simon Tavaré. 1986. Some probabilistic and statistical problems in the analysis of DNA sequences. *Lectures on mathematics in the life sciences* 17, 2 (1986), 57–86.

[49] B Sharat Chandra Varma, Kolin Paul, and M Balakrishnan. 2016. *Architecture exploration of FPGA based accelerators for BioInformatics applications*. Springer.

[50] Anuradha Welivita, Indika Perera, and Dulani Meedeniya. 2017. An interactive workflow generator to support bioinformatics analysis through GPU acceleration. In *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 457–462.

[51] Lars Wienbrandt, Jan Christian Kässens, Matthias Hübenthal, and David Ellinghaus. 2019. 1000× faster than PLINK: Combined FPGA and GPU accelerators for logistic regression-based detection of epistasis. *Journal of computational science* 30 (2019), 183–193.

[52] Xilinx. [n.d.]. Vivado Design Suite. User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_3/ug910-vivado-getting-started.pdf

[53] Xilinx. [n.d.]. Vivado Design Suite, High Level Synthesis. User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_3/ug902-vivado-high-level-synthesis.pdf

[54] Duo Xu et al. 2017. Archaic hominin introgression in Africa contributes to functional salivary MUC7 genetic variation. *Molecular biology and evolution* 34, 10 (2017), 2704–2715.

Received February 2007; revised March 2009; accepted June 2009