Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

# JACC-FPGA: A hardware accelerator for Jaccard similarity estimation using FPGAs in the cloud

Javier E. Soto [a], Cecilia Hernández [b,c], Miguel Figueroa [a,*]

[a] Electrical Engineering Department, University of Concepción, Chile
[b] Computer Science Department, University of Concepción, Chile
[c] Center of Biotechnology and Bioengineering (CeBiB), Chile

ARTICLE INFO

*Article history:*
Received 18 February 2022
Received in revised form 16 June 2022
Accepted 6 August 2022
Available online 12 August 2022

*Keywords:*
Hardware acceleration
Cloud computing
Field-programmable gate arrays
Streaming algorithms
Sketches
Genomics

ABSTRACT

Genomic similarity is a key metric in genomics, used in important tasks such as genome clustering and metagenomic profiling. One commonly-used approach is to treat each genome as a set of k-mers and to compute the Jaccard coefficient between each genome pair. However, computing the Jaccard coefficient between genomes in a large dataset is a computationally-challenging task. In this paper, we present an algorithm and accelerator architecture that uses an FPGA-as-a-service paradigm to compute the Jaccard similarity between pairs of genomes in large datasets using sketches and hardware acceleration in the cloud. The algorithm can compute the similarity between all genome pairs in the dataset, or it can use a selection criterion to reduce the amount of computation when only genome pairs with a Jaccard coefficient above a user-supplied threshold are of interest. After building the sketches, our heterogeneous accelerator can compute more than 96 million Jaccard coefficients per second running on an AWS EC2 f1.2xlarge instance with a Xilinx XCVU9P FPGA, which is 58 times faster than a state-of-the art software implementation that exploits SIMD instructions and thread-level parallelism on a compute-optimized EC2 c5.9xlarge instance with 36 hardware threads. The accelerator also computes similarities 27 times faster than a straightforward GPU-accelerated implementation of the Jaccard coefficient using sketches, and 4 times faster than an optimized GPU implementation of our algorithm, both running on an EC2 g5.4xlarge instance tailored with an NVIDIA A10G GPU. Furthermore, using a Jaccard coefficient threshold of 0.8 reduces the execution time of similarity computation to approximately one third in the hardware-accelerated and parallel software implementations, compared to computing the complete similarity matrix.

## 1. Introduction

The computation of genomic similarity is essential in different biological and biomedical applications, such as clustering [1,2], assemblers [3–5], and metagenomic profiling [6,7]. In particular, the Jaccard coefficient is a popular metric used to compute the genomic similarity between two DNA datasets. Each dataset is represented as a multiset of *k-mers*, that is, substrings of *k* DNA bases. Each of the DNA bases is a nucleotide denoted with the character ‘A’, ‘C’, ‘G’ or ‘T’. Jaccard estimates the similarity between multisets as the fraction of the k-mers that are shared between them. More specifically, the Jaccard coefficient is computed as the ratio between the cardinality of the intersection of the two multisets and the cardinality of their union. Recently published tools, such as Mash [1] and Dashing [8], use efficient algorithms and parallelism techniques to compute distance metrics based on the Jaccard coefficient. However, the current trend of continuous growth in the number of genomes and the size of the data, makes genome-similarity estimation a computationally-challenging problem.

In fact, modern sequence technologies, such as NGS (Next Generation Sequencing), have enabled the production of a large amount of publicly available genomic data. For instance, the National Center for Biotechnology Information (NCBI) Reference Sequence (RefSeq) collection [9], which provides a “comprehensive, integrated, non-redundant, well-annotated set of sequences”, has grown linearly in the past years. As of June 2022, the dataset contains sequences from 1,19,373 different organisms, with more than 6 TiB of uncompressed data [10]. Other available collections feature even more data, like the Unified Human Gastrointestinal Protein [11], which has a catalog of 2,04,938 reference genomes. The Genome Variation Map [12] grew from 8,884 samples in September 2017 to 69,004 samples in January 2022, consisting of 977, 482, 968 variants from 43 species. Estimations show that

* Corresponding author.
*E-mail addresses:* javsoto@udec.cl (J.E. Soto), cecihernandez@udec.cl (C. Hernández), miguel.figueroa@udec.cl (M. Figueroa).

https://doi.org/10.1016/j.future.2022.08.005
0167-739X/© 2022 Elsevier B.V. All rights reserved.

only human genomic data will require between 2 and 40 exabytes of storage by 2025 [13].

Due to space and time restrictions, the continuous growth of genomic datasets poses difficult challenges to the commodity computer systems used to process and analyze the data. To address some of these problems, some genomic applications use space-efficient algorithms based on sketches, which are probabilistic data structures that can accurately estimate properties of the dataset, such as cardinality and element frequencies, with sublinear memory usage [1,8]. Furthermore, sketches of the same size can represent datasets of different sizes, and algorithms can compute properties of the datasets directly from the sketch data [1,8,14–16]. To reduce execution time, the bioinformatics community has proposed efficient algorithms for modern computing architectures that exploit parallelism at the data, thread and process level, such as Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and cloud processing [17]. Even when using commodity processors, software implementations of genomics algorithms [8,18,19] use thread-level parallelism and Single Instruction/Multiple Data (SIMD) parallel operations to reduce computation time. Examples include sequence alignment [18,19] and genomic similarity computation using the Jaccard coefficient [8]. Furthermore, implementations that exploit the SIMD paradigm on GPU-based accelerators have become common. The literature reports GPU implementations for sequence alignment [20,21], k-mer counting [22], and metagenomic classification [23], which feature speedups of up to more than two order of magnitude [23] compared to traditional architectures. FPGA-based accelerators have also been used to speed up processes like seeding on sequence alignment [24–26], k-mer counting [17,27], and pairwise overlapping [28] in sequence assembly. Recently, the availability of FPGA accelerators in cloud computing platforms, such as Amazon Web Services (AWS) [29], has made this technology available to more genomics applications [24,28,30,31]. The availability of these platforms has enabled a new computing paradigm that uses the FPGA as a service for genomics and related areas [32,33].

This paper describes JACC-FPGA, a cloud-based algorithm and accelerator architecture that uses the FPGA-as-a-service paradigm, leveraging the properties of HyperLogLog (HLL) sketches to estimate genome similarity with the Jaccard coefficient. The accelerator supports two basic operations: (1) building sketches from the genome data and computing single-genome cardinality, and (2) computing pairwise similarity for a set of genomes using the sketch data. Our solution can compute the similarity matrix for all pairs of genomes in the dataset, or it can use a selection criterion to reduce execution time when we only want to retrieve those pairs with a Jaccard coefficient above a user-supplied threshold. The accelerator exploits fine- and course-grained parallelism at the spatial and temporal levels to reduce computation time and maximize memory bandwidth. The host-side software handles data input/output and performs noncritical steps in cardinality and similarity estimation. Our heterogeneous genomic similarity estimator currently runs on an Amazon Web Services EC2 F1 instance. The main contributions of our work are:

- A streaming algorithm that uses parallelism and the FPGA-as-a-service paradigm to build sketches from genome data and to estimate set cardinality and similarity between sets with sublinear memory using the sketches.
- The architecture of a hardware accelerator and host-side processing that implements the streaming algorithm.
- A selection criterion and algorithm that, based on individual set-cardinalities, reduces the number of similarity computations to retrieve only genome pairs with a Jaccard coefficient that exceeds a user-defined threshold.
- A cloud-based implementation of the algorithms using FPGA and GPU acceleration. Compared to implementations on cloud infrastructure with a similar cost, the FPGA-based implementation reduces the Jaccard coefficient computation time from sketch data by a factor of 58 compared to a state-of-the-art software implementation that exploits SIMD instructions and thread-level parallelism on a cloud machine with 36 hardware threads. It also reduces the computation time by a factor of 27 compared to a straightforward GPU-accelerated implementation of sketch-based similarity computation using Jaccard, and by a factor of 4 compared to an optimized GPU implementation that uses our streaming algorithm.

The source code for the JACC-FPGA hardware accelerator is publicly available at https://github.com/vlsi-udec/jacc-fpga.

The rest of the paper is organized as follows: Section 2 reviews related work, Section 3 presents our methods and algorithms, and Section 4 describes the FPGA accelerator architecture. Section 5 presents our experimental evaluation, including datasets, cloud-based infrastructure, selection of architectural parameters, and results. Finally, Section 6 shows our conclusions and discusses future work.

## 2. Related work

Mash is a well-known genomic similarity software presented by Ondov et al. [1], which defines the Mash distance using the Jaccard coefficient. Mash extends MinHash [34] sketches to estimate pairwise Jaccard similarities between sets. Jaccard is also related to the ANI (Average Nucleotide Identity) metric, where a ANI of 97% is considered as a similarity threshold for correct classification of prokaryotes [35]. Zhao [14] presented BinDash, a MinHash sketch that uses a rolling one-permutation hash to store only 14 bits per hash. BinDash uses half of Mash's memory footprint and processes the same data in about 10% of the time. Recently, Baker and Langmead [8] presented Dashing, a faster alternative to Mash that estimates the Jaccard similarity between sets using HLL sketches and computes the cardinality of intersections with the inclusion–exclusion principle. Dashing aggressively exploits SIMD- and thread-level parallelism and achieves high performance compared to Mash, with a speedup in execution time from 9.4 to 2.3 times depending on the sketch size. Running on a server with four Intel E7-4830 processors and a total of 112 hardware threads, Dashing can process 87,000 genomes in 6 minutes [8].

Dashing and Mash are important tools for genome comparison and functional prediction [36]. Some applications that use Dashing include research in disease analysis [36,37], computing evolutionary distances [38], and genome similarity [39]. Moustafa et al. [36] include Dashing and Mash as tools for pairwise distance in comparative genomics to decipher the spread and pathogenesis of bacterial infectious diseases. Lipworth et al. [37] use Dashing and binning to identify potential pathogen genetic factors involved in the increase of infections related to Escherichia Coli and Klebsiella bacteria. Criscuolo [38] uses pairwise similarity to estimate phylogenetic inference between genomes, and suggest that Dashing and Mash are fundamental tools for defining an evolutionary distance metric.

Genomic algorithms have consistently been an attractive target for parallel algorithms and hardware acceleration, because of the fast growth of the volume of genomic data. Dedicated hardware accelerators such as GPUs and FPGAs can exploit fine-grained parallelism at a scale beyond the capabilities of traditional SIMD instructions and multiple hardware threads available on commodity processors. GPUs mainly exploit SIMD parallelism with fixed-width data and floating-point arithmetic at a massive level using hardware-managed threads. FPGAs enable tailoring the processor architecture to the algorithm to exploit coarse- and fine-grained parallelism in the hardware. Designing for FPGAs is more time-consuming than programming GPUs, but they can achieve significantly higher performance and power efficiency [17].

GPU-accelerated genomics algorithms include genomic assemblers [21], deletion finders on samples [40], and read aligners [41,42]. Zeni et al. [42] present a high-performance long-read aligner using a six-GPU accelerator, which achieves a speedup of 30.7x over a state-of-the-art software implementation. Goenka et al. [41] present a similar approach for genome alignment, but implement it on an 8-GPU AWS instance, achieving a 14x speedup and a 2.3x cost reduction compared to a traditional software implementation.

FPGAs have been used as accelerators of streaming algorithms in applications that require high performance and data throughput [15,43–46]. Because of their fine-grained configurable architecture with limited on-chip memory, FPGA-based accelerators favor sketch-based algorithms, which can exhibit large data parallelism and a small memory footprint. McVicar et al. [27], designed an FPGA-based accelerator to count k-mers using a Bloom filter, which achieves a speedup of up to 17.6x over a multithreaded software implementation. Also, Saavedra et al. [45] presented an accelerator that uses a Countmin-CU sketch to detect heavy hitters in genomic and network traffic data, achieving a speedup of more than 700x compared to a desktop computer. Other applications of sketch-based FPGA accelerators include Tong and Prasanna [43], who combine Countmin and K-ary sketches to detect heavy hitters and heavy change in network traffic, and Soto et al. [47], who estimate network-flow entropy at more than 204 Gbps using Countmin-CU, HLL sketches, and a hashed priority queue.

Several genomics algorithms have been implemented using FPGAs in recent years, mainly on the AWS platform. Fujiki et al. [30] designed an architecture to accelerate the seed-extension step on the alignment process. They tested their implementation on an AWS F1 f1.2xlarge instance with the well-known BWA-MEM2 alignment algorithm, achieving a speedup of 2.3x over a software implementation. Subramaniyan et al. [24] designed a hardware accelerator for the same seed-and-extend step using an AWS f1.4xlarge instance. They reduce the seeding time by 3.3 times and the whole alignment execution time by 2.1 times. Ham et al. [33] used an f1.2xlarge instance to accelerate genomic analysis stages of the GATK4 workflow, achieving a speedup of up 19.3 over a multithreaded software implementation. Saavedra et al. [15] proposed an algorithm and FPGA accelerator to mine discriminative k-mers, achieving a speedup of up to 78x compared to a software implementation that exploits SIMD and thread-level parallelism on a 12-thread processor. Wu et al. [31] presented an accelerator for the realignment process, which is one-third of the execution time on genomic diagnostics of acute cancers, achieving a speedup of 81x over a software implementation of the same algorithm. In another work, Guo et at. [28] presented a pairwise overlap detector for long reads on an FPGA and a GPU. Their work accelerates the first stage of the assembly process, which frequently uses the Overlap-Layout-Consensus paradigm. Their GPU accelerator achieves a 7x speedup over a baseline software implementation, while the FPGA achieves a 28x speedup.

## 3. Methods

This section describes the approach, data structures, and parallel algorithms used in our FPGA-as-a-service solution to estimate the pairwise Jaccard similarities using sketches.

We represent each genome in a dataset as a multiset of k-mers, where each k-mer is an overlapping subsequence of $k$ bases. Like in similar approaches [1,8], our representation uses canonical k-mers: for each k-mer in the dataset, its canonical k-mer is computed by comparing its forward and reverse complement representations, and choosing the representation that is lexicographically smallest.

The Jaccard similarity between two genomes represented by the multisets $X$ and $Y$ is defined as follows:

$$J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}, \tag{1}$$

where $|\cdot|$ denotes set cardinality.

Like in Dashing [8], our proposed method estimates the pairwise Jaccard coefficients using the inclusion–exclusion principle to avoid computing the intersection of the sets:

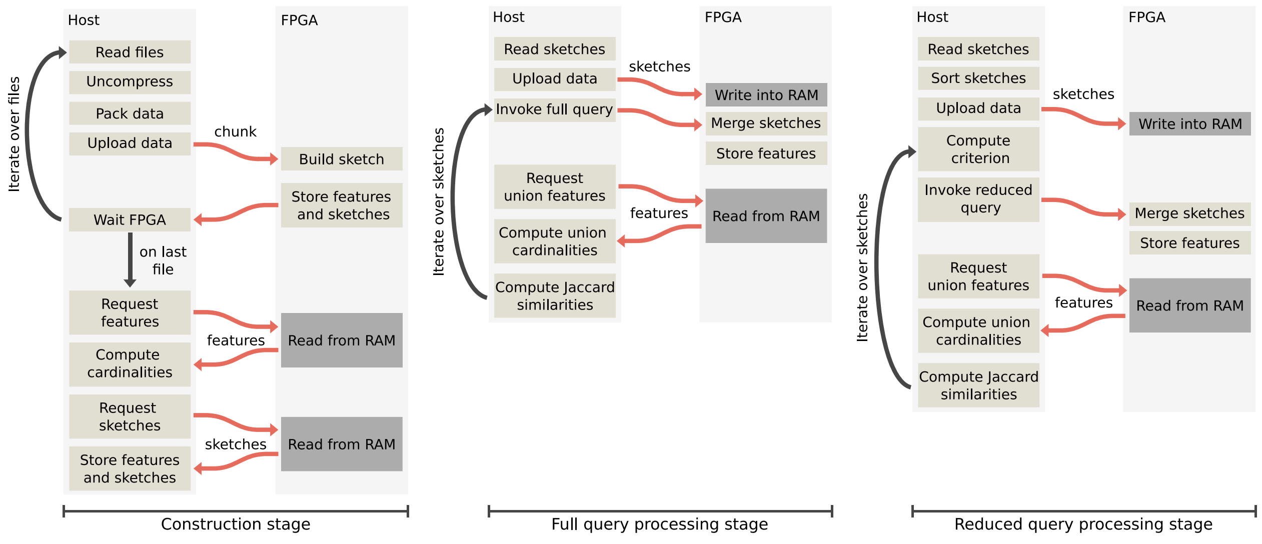
$$J(X,Y) = \frac{|X| + |Y| - |X \cup Y|}{|X \cup Y|}. \tag{2}$$

Fig. 1(a) shows the data flow of the algorithm, indicating the tasks performed on the host and the FPGA accelerator. The main functions are the construction stage and the two query processing stages. The construction stage uses thread parallelism on the host side to read the collection of compressed genome data files, merging them into data chunks, and asynchronously transferring them to the FPGA. A dedicated processor on the FPGA builds sketches for multiple genomes in parallel, saves them to its off-chip memory (RAM), and computes features used for cardinality estimation on the fly. The host transfers the data, uses the features to compute the cardinality of each genome, and sorts and saves the sketches and cardinalities.

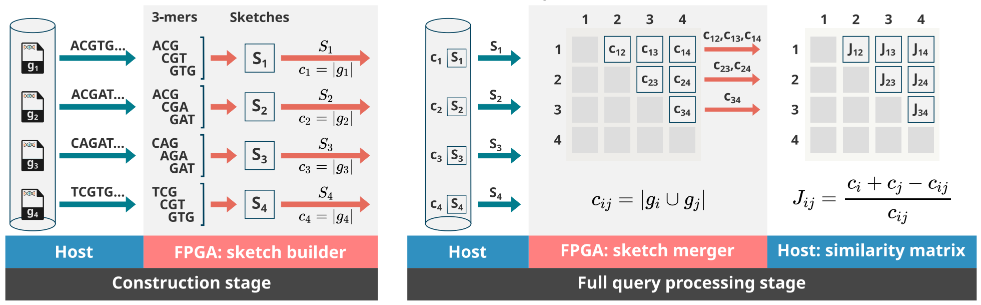
In the query stage, the host can make two types of queries to the accelerator. The first query, labeled as *full query processing stage* in Fig. 1(a), computes the similarity matrix of the complete genome collection in the dataset using the sketch data. The second query, labeled *reduced query processing stage*, is designed to only obtain genome pairs with a Jaccard similarity above a user-supplied threshold. For the second query, in Section 3.3.2 we propose a method that discards *a priori* those genome pairs that, based on their cardinality, cannot comply with the similarity threshold, thus avoiding the computation of the complete similarity matrix. On each iteration, the host uses the selection criterion to select a subset of genomes and query the accelerator for the similarities between them. The FPGA performs a parallel computation of the features used to estimate union cardinality for the selected subset using the sketch data, and transfers them back to the host. The host uses the features to compute the Jaccard similarities and select those above the threshold.

Fig. 1(b) shows a simplified example of the data flow for 4 genome files using k-mers of length 3. During the construction stage, the host streams the data from the genome files to the FPGA, which extracts the k-mers and builds the 4 sketches, sending their data and cardinalities back to the host (in practice, the last step of cardinality estimation is performed on the host). In the full query processing stage, the host uploads the sketches to the FPGA, which merges the sketch data to compute the joint cardinalities. Finally, the host uses Eq. (2) to compute the Jaccard similarities using the individual and joint cardinalities.

The rest of this section describes the algorithms and data structures that implement each of the stages described above, beginning with a brief summary of the HLL sketch. To improve readability, we have moved the formal description of the less relevant algorithms to Appendix.



(a) Data flow of the algorithm.



(b) Simplified example of sketch construction and full query with 4 genome files.

**Fig. 1.** (a) Data flow of the algorithm. In the construction stage, multiple host threads read and decompress the genome data files and the host uploads the data to the FPGA, which builds an HLL sketch for each genome and computes the features used for cardinality estimation. The sketches and features are sent back to the host, which computes the cardinalities and stores the data. In the full query processing stage, the host uploads all sketch data to the FPGA, which then computes the cardinality features for the union of each pair of genomes and sends them back to the host. The host computes the Jaccard similarity for each genome pair. In the reduced query processing stage, the host first sorts the sketches by cardinality and then uses the selection criterion to upload to the FPGA only those sketches that can yield pairwise similarities above the user-provided threshold. (b) Simplified example of sketch construction and full query using 4 genome files and k-mers of length 3. For simplicity, the cardinalities are shown as if completely computed on the FPGA, but the final step of cardinality computation takes place on the host.

## 3.1. Background: Cardinality estimation with the HLL sketch

HLL is a sketch that can estimate the number of distinct elements in a multiset with sublinear space complexity and low estimation standard error [48]. The sketch represents the value of each element as the position of the most-significant one in the binary representation of the element (the number of leading zeros plus one), thus logarithmically reducing the space needed to store the value. Moreover, HLL distributes the information over $m$ elements, or buckets, of an array using a hash function to uniformly map the inputs across the buckets, thus improving its estimation of the number of distinct elements. The algorithm uses a precision parameter $p$, such that $m = 2^p$.

To populate the sketch, the HLL insertion algorithm first applies a 64-bit hash function to each input (in our case, the canonical k-mers extracted from the genome data file). The algorithm uses the $p$ most-significant bits of each hash value to determine the bucket index corresponding to the input. The algorithm then computes the number of leading zeros plus 1 in the $64 - p$ least-significant bits of the hash, and compares this number to what is already stored in the selected bucket, storing the maximum value back in the bucket. This process is repeated for every element in the multiset. Algorithm 6 in Appendix A.1 formally describes the HLL insertion algorithm.

To estimate the cardinality of the multiset, HLL computes the harmonic mean of the buckets, defined as:

$$Z = \sum_{j=0}^{m-1} 2^{-A[j]}, \tag{3}$$

where $m = 2^p$ and $A$ is the array that stores the buckets. Then, the cardinality is estimated as:

$$C_{HLL} = \alpha_A \frac{m^2}{Z}, \tag{4}$$

where $\alpha_A$ is a constant associated with the sketch size.

The estimator in Eq. (4) produces inaccurate results on sets with a small number of distinct elements compared to the size of the sketch. Heule et al. [49] proposed a correction to the estimation when $C_{HLL} \leq 2.5m$:

$$C_{HLL} = m \log\left(\frac{m}{n_z}\right), \tag{5}$$

where $n_z$ is the number of unused buckets, which store a value equal to zero. Algorithm 7 in Appendix A.1 formally describes the HLL estimation algorithm.

HLL also defines the union operation between two sketches, which is performed by computing the maximum value between the corresponding buckets in the sketches. Running the estimation algorithm on the resulting buckets yields an estimate of the cardinality of the union of the two input multisets.

**Algorithm 1:** Host: producer thread

```
Input: thread index $t$, block size $K$, queue $q$, file stack $files$
while $len(files) > 0$ do
    $f \leftarrow files.pop()$;
    while not $end\_of\_file(f)$ do
        $data \leftarrow f.read(K)$;
        $q.push(t, data)$;
    $q.push(t, eof\_mark)$;
```

**Algorithm 2:** Host: consumer thread

```
Input: number of threads $T$, block size $K$, blocks per chunk
       per thread $C$, queue $q$
Let
    $offset = 0$
    Initialize $chunk$ as array of size $C \times T \times K$
    for $i \leftarrow 0$ to $T - 1$ do
        Initialize $queue[i]$ as a queue of blocks
while reading files do
    if not $q.empty\ ()$ then
        $t, data \leftarrow q.pop()$
        $queue[t].push(data)$
    if $all\_nonempty(queue)$ then
        for $i \leftarrow 0$ to $T - 1$ do
            $chunk[offset + i \times K] \leftarrow queue[i].pop()$
        $offset \leftarrow offset + T \times K$
        if $offset == C \times T \times K$ then
            $parallel\_sketch\_kernel\ (chunk)$
```

## 3.2. Construction stage

The construction stage builds HLL sketches to represent the genomes in the dataset, treating each genome as a set of canonical k-mers. Fig. 2 illustrates the sketch construction process on the host and the accelerator. The host side uses a multithreaded process based on the consumer/producer pattern to overlap disk access, data transfer to the accelerator, and data processing on the FPGA. The host runs $T$ producer threads ($T = 8$ in the figure) to read and decompress FASTA/FASTQ genome data files in parallel using zlib [50], and one consumer thread that packs and transfers data to the accelerator. The number of files read simultaneously is limited by the number of hardware threads available in the host and its total disk bandwidth, and is limited to $T = 8$ in the AWS f1.2xlarge instance used in our current implementation. Algorithm 1 shows a producer thread, which reads one file at a time and pushes blocks of data into a shared queue, which is read by the consumer thread to simultaneously transfer data from multiple files to the accelerator. We use the Jiffy [51] library to implement the multiple-producer/single-consumer queue. Each producer thread inserts an end-of-file marker when it reaches the end of its current file before inserting data from the next genome.

Algorithm 2 describes the consumer thread. The thread reads blocks from the shared queue and separates the data onto multiple queues, one for each producer. Every time there is at least one block in each queue, the consumer builds a chunk slice that contains data from all producers. When a chunk of $T \times C$ blocks has been assembled, the thread copies it to a dedicated buffer to transfer it to the accelerator using an asynchronous OpenCL call. Along with the data, the consumer also enqueues the kernel call to the accelerator to process the chunk.

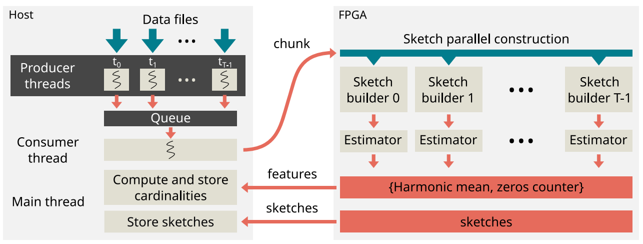


**Fig. 2.** Process diagram and data flow of sketch construction. $T$ producer threads in the host read and decompress data from the genome files in parallel. A consumer thread packs the data in chunks, interleaving sequences from all producer threads, and transfers them to the accelerator memory. The accelerator has $T$ sketch-builder modules that read the data and build an HLL sketch in parallel for each genome file. Internally, each sketch builder operates $B$ parallel sketches to maximize throughput. After reading the data from a file, the sketch builder computes the sketch features (harmonic mean and number of empty buckets) and transfers them along with the sketch data back to the host.

**Algorithm 3:** Accelerator: parallel sketch kernel

```
Input: data block $chunk$
Let
    k-mer length $k = 31$
    number of threads $T = 8$
    number of internal HLL sketches $B = 64/T$
    for $i \leftarrow 0$ to $T - 1$ do
        Init $line\_buffer[i]$ bit array of size $2(k + B - 1)$
        Init $sketch[i]$ as HLL_sketch array of size $B$
foreach $chunk\_word$ in $stream$ do
    for $t \leftarrow 0$ to $T - 1$ do in parallel
        $c \leftarrow chunk\_word[64t : 64(t + 1) - 1]$
        $sketch\_builder(B, k, t, c, line\_buffer[t], sketch[i])$
```

Fig. 2 shows that the accelerator features $T$ sketch builder modules. All sketch builders operate in parallel to populate HLL sketches using a different slice of the chunk data. Algorithm 3 describes the parallel sketch kernel in the accelerator. Because the FPGA memory bus is 512-bits wide, when the kernel performs a memory read, it retrieves a 512-bit word (*chunk_word*) from the chunk uploaded by the host. Because each base is encoded as an 8-bit ASCII character 'A', 'C', 'G' or 'T', *chunk_word* contains $512/8 = 64$ bases. The kernel passes the data to the $T$ parallel sketch builders, so that each builder simultaneously receives $B = 64/T$ bases to update its sketch. The kernel repeats the process until the entire stream of chunks from the host has been processed.

Algorithm 4 shows the operation of each sketch builder. To understand the algorithm, it is important to note that each sketch builder maximizes performance by actually populating $B$ parallel HLL sketches, which are later merged to transfer a single sketch and features back to the host. Thus, the accelerator exploits spatial parallelism not only through multiple sketch builders, but also by using multiple HLL sketches within each builder to construct a single sketch. Every time the kernel performs a sketch update, each of the $T$ sketch builders receives $B$ bases, which allows it to construct $B$ canonical k-mers. Our current implementation uses $k = 31$, but the design is parameterized, so it is straightforward to generate an accelerator for k-mers of different lengths.

Algorithm 4 shows that the sketch builder first reads a slice of $B$ bases from of each chunk word, and reencodes the bases using 2 bits per base. A line buffer is used as a shift register to store the $k + B - 1$ most-recently read bases, and the builder extracts $B$ k-mers from the buffer. Each of the $B$ HLL sketches within

**Algorithm 4:** Accelerator: sketch builder

```
Input: number of HLL $B$, k-mer length $k$, thread index $t$,
       64-bit chunk slice $c$, line buffer $line\_buffer$, HLL
       sketch array of 8 elements $sketch$
1  $b \leftarrow two\_bit\_encode(c)$
2  $line\_buffer \leftarrow (line\_buffer << 2B) \mid b$
3  if $line\_buffer \neq eof\_mark$ then
4      for $i \leftarrow 0$ to $B-1$ do in parallel
5          $k\text{-}mer \leftarrow line\_buffer[2i : 2i+2k-1]$
6          $k\text{-}mer_{canon} \leftarrow canonical\_kmer(k\text{-}mer)$
7          $sketch[i].insert(k\text{-}mer_{canon})$
8  else
9      for $i \leftarrow 0$ to $B-1$ do reduction in parallel
10         $o\_sketch \leftarrow union(o\_sketch, sketch[i])$
11     $hmean \leftarrow harmonic\_mean\ (o\_sketch)$
12     $nempty \leftarrow count\_zeros\ (o\_sketch)$
13     $store\_in\_memory\ (hmean, nempty, o\_sketch)$
```

the builder receives one of these k-mers as input, computes the canonical k-mer using Algorithm 8 Appendix A.2, and then it updates its own sketch using Algorithm 6. When the builder detects an end-of-file marker, it merges its sketches by linearly traversing the sketch buckets and computing

1. the maximum between each corresponding set of $B$ buckets, which yields the unified sketch for the genome data,
2. the first feature, which is the harmonic mean $Z$ shown in Eq. (3), and
3. the second feature, which is the number of empty buckets in the unified sketch, denoted as $n_z$ in Eq. (5).

### 3.3. Parallel query processing

In this section, we first present the parallel algorithm used in the accelerator to compute the similarity matrix for all genome pairs in the dataset. Then, we describe a method that, given a minimum similarity threshold, uses the cardinality of each genome multiset to discard *a priori* those genome pairs whose similarity cannot exceed the threshold.

#### 3.3.1. Parallel similarity matrix algorithm

Pairwise similarities in a genome dataset are usually presented as an upper triangular matrix of dimensions $N \times N$, where $N$ is the number of genome files present in the dataset. A straightforward single-threaded software solution computes the similarity matrix using a doubly nested-loop to generate all genome pairs. A multithreaded software implementation can reduce the computation time by computing multiple rows from the similarity matrix in parallel. In both cases, each thread selects one genome, or pivot, and computes its similarity to the other genomes in sequence. Our hardware-accelerated solution uses a similar principle to the multithreaded approach, with up to $V$ pivots. The maximum number of pivots is limited by the hardware resources available on the FPGA.

Algorithm 5 shows the parallel computation of the similarity matrix on the accelerator. Before invoking this kernel, the host uploads the complete set $s$ of $N$ sketches computed in the first stage. The accelerator can store sketches for up to $V$ pivots in the FPGA on-chip memory. In the context of the accelerator, a pivot can be accessed multiple times in a single kernel call. In contrast, the rest of the sketches are read from external memory in a streaming fashion, because the FPGA does not possess sufficient

**Algorithm 5:** Accelerator: similarity matrix

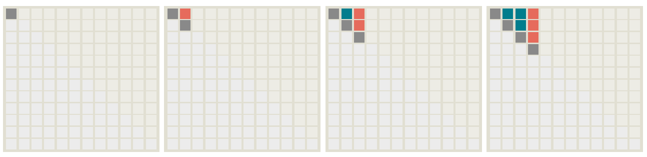
```
Input: number of pivots $V$, number of genome sketches $N$,
       number of sketches read in parallel $D$, sketch set $s$
Output: similarity matrix $M$
1  Let
2      Initialize $P$ as an array of sketches of size $V$
3      Initialize $M$ as an $N \times N$ similarity matrix
4  for $i \leftarrow 0$ to $N/V-1$ do
       // Load $V$ pivots and compute
       // pairwise similarities
5      for $j \leftarrow 0$ to $V-1$ do
6          $x \leftarrow i \times V + j$
7          do in parallel
8              $P[j] \leftarrow stream(s[x])$
9              for $k \leftarrow j$ to 0 do in parallel
10                 $y \leftarrow i \times V + k$
11                 $M[x, y] \leftarrow Jaccard\ (stream(s[x]), P[k])$
       // Iterate streaming $D$ sketches in
       // parallel and compute similarities
12     for $x \leftarrow ((i+1) \times V)/D$ to $N/D-1$ do
13         for $k \leftarrow 0$ to $V-1$ do in parallel
14             $y \leftarrow i \times V + k$
15             for $n \leftarrow 0$ to $D-1$ do in parallel
16                 $z \leftarrow x \times D + n$
17                 $M[x, y] \leftarrow Jaccard(stream(s[z]), P[k])$
18 return $M$
```

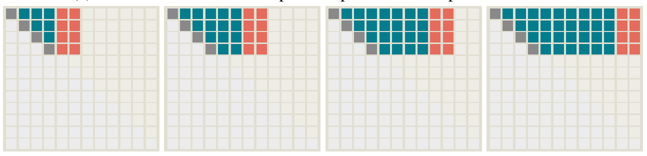
on-chip memory to store them. We can stream multiple sketches in parallel, limited by the external memory bus width.

Each iteration in the main loop of the kernel uses $V$ pivots to produce $V$ rows of the similarity matrix. In each iteration, the kernel performs two steps. In the first (lines 5–11), the accelerator reads the sketches of $V$ pivots and stores them in local memory, denoted by the array of sketches $P$. For all but the first pivot, while the accelerator is reading the sketch data, it computes the similarity between that sketch and the pivots already stored in the accelerator. The similarity computation uses Eq. (2), where $X$ is a pivot and $Y$ is the incoming sketch. The cardinalities $|X|$ and $|Y|$ were computed in the sketch construction stage, and the cardinality $|X \cup Y|$ is computed in a streaming fashion on the accelerator by merging sketches $X$ and $Y$ while the contents of sketch $Y$ are read. As a result, the first step of the main loop stores $V$ pivots and computes the $V(V-1)/2$ Jaccard similarities between them, which are the first nonzero elements of the corresponding $V$ rows of the similarity matrix.

In the second step (lines 12–15), the accelerator streams the sketches that are numbered above the pivots to complete the $V$ rows of the similarity matrix. Using the wide memory bus, the accelerator reads data from $D$ sketches in parallel, which allows it to simultaneously compute $V \times D$ Jaccard similarities (between the $D$ incoming sketches and the $V$ stored pivots). Although Algorithm 5 shows the complete algorithm as a single kernel, the host actually performs two kernel calls for each iteration of the loop in line 4: one for the loop in lines 5–11 and the second for the loop in lines 12–17.

Fig. 3 illustrates the progression of the similarity matrix as Algorithm 5 computes its coefficients, for $N = 12$, $V = 4$ and $D = 2$. In the figure, orange squares represent the Jaccard similarities as they are being computed in the current step, and teal squares are the similarities computed in previous steps. Gray squares in the diagonal represent the similarity between a genome and


(a) First iteration of main loop: reads pivots and computes similarities.


(b) First iteration: streams sketches and computes similarities.

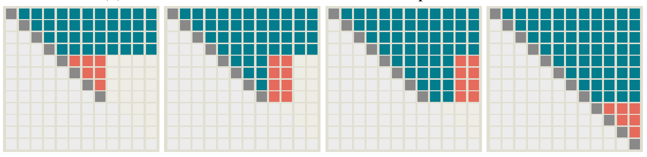

(c) Second (first three diagrams) and third (fourth diagram) iterations.

**Fig. 3.** Example of the execution of Algorithm 5 for a $12 \times 12$ similarity matrix. The example uses $V = 4$ pivots and reads $D = 2$ sketches simultaneously from external memory. Orange squares represent the computations in the current step, which occur in parallel in each step shown in the figure. Teal squares were computed in previous steps and gray squares in the diagonal are not computed. (a) The accelerator sequentially loads the first $V$ pivots and computes the similarity between them (lines 5–11). (b) The accelerator streams the sketches numbered higher than the pivots, $D$ sketches simultaneously, and computes their similarity to the pivots to complete $V$ rows of the matrix (lines 12–17). (c) The accelerator reads a new set of pivots, computes the corresponding row, and finally reads the last $V$ pivots, thus completing the similarity matrix.

itself, which is not computed. Fig. 3(a) shows the first iteration of the loop in lines 5–11 of Algorithm 5: The accelerator loads the first $V$ pivots one at a time, and computes the similarity in parallel between the incoming pivot and the ones already stored in its local memory. Fig. 3(b) shows consecutive iterations of the loop in lines 12–17, where in each iteration the accelerator simultaneously streams in $D$ sketches and computes their similarity to the stored pivots, producing $V \times D$ similarities in parallel. Finally, Fig. 3(c) shows the execution of the rest of the algorithm: First, the accelerator loads the second set of $V$ pivots and computes the similarities between them, then it performs two iterations of the loop in lines 12–17 to complete the $V$ rows of the matrix, $D$ columns at a time; finally, it loads the last set of pivots and completes the similarity matrix.

### 3.3.2. Pair selection criterion

In many applications that use genomic similarity [1,35], the goal is to find all genome sequence pairs whose similarity exceeds a predetermined threshold or, equivalently, whose distance falls below a given value. A straightforward approach to obtains all pairs of interest is to first compute the complete similarity matrix and then select the pairs that satisfy the similarity constraint. The total number of similarity computations required to obtain the complete matrix is $N \times (N-1)/2$, where $N$ is the number of genome files in the dataset. This quadratic complexity is computationally challenging with modern datasets that contain a large number of genomes.

In order to reduce the number of similarity computations, we propose an algorithm that *a priori* discards all those genome pairs that are incapable of producing a similarity that exceeds a given threshold. Because typically useful thresholds have a high value [35], this method can significantly reduce the number of similarities that need to be computed to obtains the pairs of interest. Our algorithm selects the candidate pairs using a criterion based on the cardinalities of the individual multisets associated to the genomes, which are computed in the first stage of the algorithm, as described in Section 3.2.

The intuition behind the selection criterion is based on the following ideas: As Eq. (1) shows, the Jaccard coefficient for any two multisets $X$ and $Y$ reaches its minimum value of 0 when $|X \cap Y| = 0$, that is, the two sets have no elements in common. In that case, $|X \cup Y| = |X| + |Y|$. Conversely, Eq. (2) shows that the Jaccard coefficient value is maximal when $|X \cup Y|$ is minimal, which happens when one of the sets is completely contained in the other, such that $|X \cup Y| = \max(|X|, |Y|)$. Thus, given two multisets $X$ and $Y$, their Jaccard coefficient complies with

$$0 \leq J(X,Y) \leq \frac{|X| + |Y| - \max(|X|,|Y|)}{\max(|X|,|Y|)}. \tag{6}$$

Note that when $X$ and $Y$ are identical, $|X| = |Y| = |X \cup Y|$ and $J(X,Y) = 1$. However, in general this will not be the case. Without loss of generality, let us assume that $|X| \leq |Y|$. In that case, from Eq. (6), we can state that

$$J(X,Y) \leq \frac{|X|}{|Y|}. \tag{7}$$

Knowing only the cardinalities of $X$ and $Y$, Eq. (7) establishes an upper bound for $J(X,Y)$. If we are only interested in those sets $X$ and $Y$ for which $J(X,Y) \geq h$, where $h$ is a predetermined threshold, then we need to compute $J(X,Y)$ only when

$$h \leq J(X,Y) \leq \frac{|X|}{|Y|}, \tag{8}$$

that is, $J(X,Y) \geq h$ only if

$$|Y| \leq \frac{|X|}{h}. \tag{9}$$

Equation (9) tells us that if $|Y| > |X|/h$, then $J(X,Y) < h$ and the sets $X$ and $Y$ do not comply with the similarity threshold. We use this criterion to reduce the number of Jaccard similarities computed by Algorithm 5. The host sorts the sketches by cardinality before uploading them to the external memory of the accelerator so that, when the accelerator reads the $V$ pivots in lines 5–11 and streams the sketches to compare them to the pivots in lines 12–17, it does so in order of increasing cardinality. If we are only interested in those pairs whose similarity exceed the threshold $h$, then the loop in lines 12–17 does not need to stream all the sketches. Instead, it streams only those for which their cardinality is lower than or equal to $c/h$, where $c$ is the highest cardinality in the set of pivots currently residing in the accelerator local memory. The host applies this criterion when it issues the kernel call that executes the loop in lines 12–17. Using Eq. (9), it determines the upper limit of the loop iteration, which controls how many sketches the accelerator must stream in line 17. In Section 5.3, we quantify the reduction in execution time achieved by using this criterion.

## 4. Accelerator architecture

We use a heterogeneous architecture comprised of a host computer and a Xilinx FPGA-based accelerator board that interfaces to the host via PCI-Express. The host reads and decompresses data, performs the final step of cardinality and Jaccard computation, and stores the sketches. The host communicates with the accelerator using an OpenCL extension provided by the Xilinx Vitis development platform, where hardware modules are presented to the software as OpenCL kernels. The *sketch construction kernel* computes and returns the HLL sketches along with their harmonic mean and number of empty buckets. The host uses this kernel to compute the HLL sketches and cardinalities of all the genomes

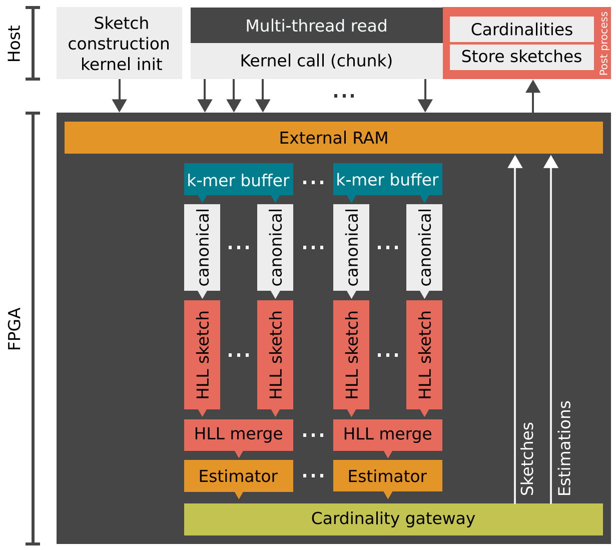


**Fig. 4.** Architecture of the sketch construction kernel, including the host and accelerator sides. The gray blocks in the host represent the execution of the loop in Algorithm 2, which transfers chunks of data to the accelerator using kernel calls. The post process block in the host stores the sketch data and computes the cardinalities of the genome multisets. On the hardware side, the accelerator consists of a set of $T$ parallel HLL sketch construction modules, each composed of multiple pipelines that process $B$ k-mers in parallel. During construction, each pipeline reads k-mers from external memory through a shared k-mer buffer, computes canonical k-mers and updates the HLL sketch. After reading a complete genome file, each sketch construction module merges its $B$ sketches and computes the harmonic mean and number of empty buckets from the merged sketch data. The cardinality gateway module writes the results back to external memory.

in the dataset. The *similarity computation kernel* merges pairs of HLL sketches and computes the same features. In this case, the host uses the features to compute the cardinality of the union of the genome pairs, and obtains the Jaccard similarity for each pair using Eq. (2). We designed the architecture of the accelerator at the register-transfer level (RTL) using the SystemVerilog hardware-description language.

### 4.1. Sketch construction kernel

During the sketch construction stage, the host first uploads a binary configuration stream to program the architecture of the accelerator on the FPGA and defines memory buffers to transfer data to and from the FPGA board. Then, the host executes Algorithms 1 and 2 to read and transfer genome data to the accelerator in successive kernel calls. When an end-of-file mark is detected, the accelerator stores the sketch data in off-chip memory and computes the harmonic mean and number of empty sketches, which are also stored in external memory. The host reads these results, computes the cardinalities of the genome sets, sorts them, and stores them with the sketch data on disk.

Fig. 4 shows the architecture of the accelerator and the processes taking place on the host side. As discussed in Section 3.2, the host uses $T$ parallel threads to read and uncompress FASTQ/FASTA files for each genome, and builds chunks of data that pack 8-bit ASCII representations of base values from the $T$ files. Through multiple calls to the sketch construction kernel, the hosts streams the packed data to the accelerator in chunk units. The FPGA in the accelerator contains $T$ sketch builder modules that operate in parallel, reading data from the chunks in external memory. The FPGA reads the chunks in lines of 512 bits, determined by the width of its memory bus. Thus, each line contains 64 bases, and each sketch builder receives $B = 64/T$ bases per clock cycle during a memory read. The k-mer buffer transforms each base to a 2-bit representation, ignoring characters other than 'A', 'C', 'G' and 'T'. This allows us to avoid processing the file on the host, using only valid DNA bases and ignoring the headers and any other metadata. According to Algorithm 3, the k-mer buffer implements a line buffer of $2(k + B - 1)$ bits that acts as a shift register, where $k$ is the length of the k-mer. On each clock cycle, the line buffer shifts in $2B$ new bits, corresponding to the new $B$ bases from the chunk line, and builds $B$ new k-mers, which are used to update the sketches using Algorithm 6, described in Appendix A.1. Adjacent k-mers overlap by $k - 1$ bases.

To exploit the parallelism of receiving $B$ new k-mers in each clock cycle, each sketch construction module is actually implemented as $B$ HLL sketches, each one receiving one of the $B$ k-mers on each clock cycle. The k-mer is transformed into its canonical representation before being added to the HLL sketch. The modules that compute the canonical representation implement Algorithm 8, described in Appendix A.2. All the operations are fully pipelined to maximize clock frequency.

When the sketch construction module detects an end-of-file-mark, it streams out the contents of its $B$ sketches and merges them using the union operation between HLL sketches. A state machine simultaneously reads the sketches and selects the maximum value between each of the corresponding $B$ buckets. This operation is performed by a pipelined tree of comparators with a latency of $\log_2 B$ cycles and a throughput of one result per clock cycle. The sketch arrays are implemented using on-chip memory blocks with an aspect ratio of $1K \times 32$ bits. Using two blocks in parallel, we configure a $1K \times 64$-bit memory, which implements a $1K \times 16$-bucket array with 4-bit buckets (Section 5.1 justifies our choice of 4-bit buckets). Thus, each iteration of the state machine produces 16 buckets of the merged sketch simultaneously. This approach reduces the total number of clock cycles to traverse the sketches by a factor of 16 and makes efficient use of on-chip memory resources. The merged sketch values are streamed to the *Estimator* and *Cardinality gateway* modules in Fig. 4.

The *Estimator* module computes the harmonic mean of the merged buckets using Eq. (3) and counts the number of empty (zero-valued) buckets. The module uses fixed-point arithmetic with 14 integer and 15 fractional bits (Q14.15) to compute the harmonic mean, which is sufficient for the $2^{14} \times 4$-bit sketches used in our current implementation. The merged sketch data and the computed features are written into the external memory using the *Cardinality gateway*, from where they can be transferred back to the host. The host retrieves the values and computes the cardinality of the set using Eqs. (4) and (5).

In our current implementation, the host supports a maximum of $T = 8$ parallel hardware threads, therefore the sketch construction kernel in the accelerator processes 8 genome files in parallel using 8 sketch construction modules, each composed of $B = 8$ parallel HLL sketches.

### 4.2. Similarity computation kernel

Fig. 5 shows the architecture of the accelerator module used to compute the similarity matrix, where processes run on the host side and the module resides on the accelerator. The same accelerator module is used to compute both the complete similarity matrix and the similarities above the user-defined threshold with the selection criterion described in Section 3.3.2. The host controls the method to use by assigning the appropriate values to the parameters passed to the kernel.

The architecture shown in Fig. 5 supports two kernel calls from the host. The first kernel implements the loop in lines 5–11 of Algorithm 5, which loads a set of $V$ pivots and computes the $V \times (V - 1)/2$ similarities between them. One execution of this

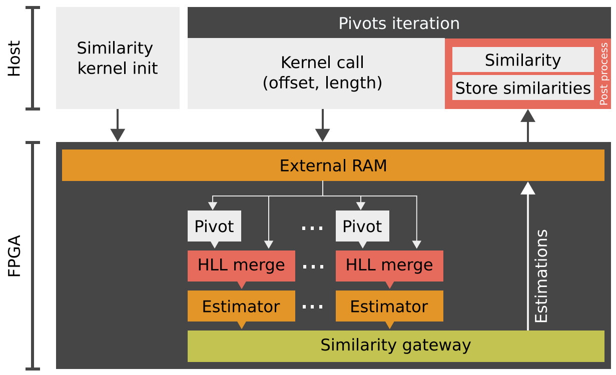


**Fig. 5.** Architecture of the similarity computation kernel. The host initializes the kernel by uploading the sketch data to the accelerator external memory, which was computed during the sketch construction stage, sorted by cardinality. For each iteration of the main loop in Algorithm 5, the host first executes a kernel call that reads $V$ pivot sketches into the accelerator, stores their data in on-chip memory (labeled "Pivots" in the figure) and computes the cardinality features of their union using the HLL merge and Estimator modules. Then, the host executes a second kernel call that streams in the rest of the sketches in the same rows as the pivots (or fewer if the host uses the selection criterion), and computes the cardinality features of the union between them and the pivots. Although for simplicity the figure shows only one HLL merge/Estimator module per pivot, the accelerator reads and computes features for $D$ sketches in parallel.

loop is illustrated in Fig. 3(a). Before executing the kernel call, the host uploads the sketch data to the accelerator external memory. To enable the threshold-based selection criterion, the host has previously sorted the sketches in order of increasing cardinality, as discussed in Section 3.3.2. The host calls the kernel, passing as parameters the number of pivots to read ($V$) and a pointer to the first pivot ($i \times V$ in the algorithm). The accelerator loads the pivot sketches from memory, one pivot at a time, and loads its data in on-chip memory blocks labeled as *Pivot* in Fig. 5. While loading the pivot data, the *HLL merge* block merges its local pivot and the incoming sketch data using the same hardware blocks described in Section 4.1. The merged buckets are not stored, but instead they are streamed to the *Estimator* module, which computes the harmonic mean and number of empty buckets using the same hardware modules discussed in Section 4.1. These two features are written back to external memory and transferred to the host. The host then computes the cardinality of the merged sketches (the union of the genome sets) and uses Eq. (2) to compute their Jaccard similarity, using the cardinalities of individual sets that were computed and stored during the sketch construction stage.

In order to accelerate the similarity computation between pivots, we want to simultaneously read as many buckets as possible from a single sketch. The sketches are stored using on-chip memory, which has a maximum configurable aspect ratio of $512 \times 64$ bits on a Xilinx UltraScale+ FPGA. As discussed in Section 5.1, we use sketches of $2^{14} \times 4$-bit buckets in our HLL sketches, so a 64-bit wide memory can read or write 16 buckets simultaneously. To implement a $2^{14}$-bucket sketch, we use two parallel memory blocks, so the total width of the sketch is 32 buckets. Therefore, the accelerator configures a 128-bit interface to external memory, which allows it to read 32 buckets in each clock cycle to load the pivots and compute the merge operations. As discussed in Section 5.2, the number of pivots in our current implementation is $V = 80$, limited by routing resources.

The second kernel computes the similarities between the pivots and the sketches that complete the $V$ rows of the similarity matrix where the pivots reside (see Fig. 3(b)). We will call these sketches *row sketches*. Each kernel call implements the loop in lines 12–17 of Algorithm 5, reading $D$ sketches in parallel in each iteration to merge them with the $V$ pivots. Fig. 3(b) illustrates one execution of the kernel. Note that, unlike the pivots, the row sketches are read from memory but not stored on-chip. Instead, the *HLL merge* modules merge the row sketches with the pivots and compute their cardinality features, in the same way used to compute the similarities between pivots. For simplicity, Fig. 5 shows only one *HLL merge*/*Estimator* path per pivot, but actually each pivot implements $D$ paths that allow it to simultaneously merge the pivot data with the $D$ sketches that are being read from memory. When it issues the kernel call, the host passes a pointer to the first row sketch to the accelerator, and the number of row sketches to read. When computing the entire similarity matrix, this number is chosen to complete the $V$ rows of the matrix, as shown in line 12 of Algorithm 5. When using the selection criterion to only compute the similarities above a threshold $h$, the number of row sketches to read is determined using Eq. (9) so that the last sketch has a cardinality of at most $|X|/h$, where $X$ is the pivot with the largest cardinality currently stored in the accelerator.

The value of $D$ is limited by the logic and routing resources on the FPGA and, more importantly, by the maximum width of the FPGA memory bus, which is 512 bits in our current platform. Since we read 32 buckets from each sketch simultaneously, which requires a 128-bit bus, the accelerator uses the 512-bit memory interface to read $D = 4$ sketches in parallel.

## 5. Results

This section describes the dataset collection, performance metrics, and infrastructure used in our experimental evaluation. We compare the performance of our FPGA-accelerated algorithm to Dashing [8], a state-of-the-art software implementation of Jaccard in genomics. We also developed two GPU-based implementations that use Jaccard to compute genome similarity. Finally, we compare our FPGA-based implementation of the threshold-based selection criterion to the two GPU accelerators, and to a parallel software implementation based on Dashing.

Our experiments use the complete collection of genomes of the Reference Sequence (RefSeq) available to download individually. The dataset comprises the 128,110 genome files available at the NCBI website [9] as of June 1, 2022, where the smallest genome has 220 DNA bases and the largest has $40,054,341,269$. The mean and median are $12,925,678$ and $4,158,070$ bases, respectively. In total, the dataset contains $1,655,908,634,893$ bases. For the smaller tests in Section 5.1, we use the same subset of 400 genome pairs used by Dashing [8], which feature Jaccard similarities varying in the complete range $[0, 1]$.

We evaluate the performance of sketch construction and similarity-matrix computation on four different algorithms and implementations:

**Dashing:** The state-of-the art algorithm for genome similarity mentioned above, which exploits thread-level and SIMD instructions to achieve high performance on modern multicore processors. We use Dashing v1.0, which is available at https://github.com/dnbaker/dashing.

**SF-GPU:** A straightforward GPU-based implementation of sketch construction and similarity computation using the NVIDIA Compute Unified Device Architecture (CUDA) platform. The sketch construction stage uses Algorithm 1 to read the genome files on the host, and a CUDA kernel extracts, encodes and hashes the k-mers from each genome. A second kernel builds the sketches and stores them in the GPU global (external) memory. Both kernels are composed of multiple CUDA streams, one per each producer thread on the host. In the similarity computation stage, we launch one CUDA thread for each row of the similarity matrix, which merges the sketches stored in global memory to compute the Jaccard coefficients in parallel.

**Table 1**
AWS EC2 instances used in our experiments. All instances use virtualized processors and the number of vCPUs refer to the number of parallel hardware threads supported by the machine. The f1.2xlarge instance uses a Xilinx XCVU9P UltraScale+ FPGA. The g5.4xlarge instance uses an A10G NVIDIA Ampere GPU. Prices and features are valid as of June 2022.

| Instance | CPU family | vCPU | RAM (GiB) | Accelerator hardware | Price (USD/hour) |
|---|---|---|---|---|---|
| f1.2xlarge | Intel Xeon | 8 | 122 | XCVU9P | 1.65 |
| g5.4xlarge | AMD EPYC | 16 | 24 | A10G | 1.62 |
| c5.9xlarge | Intel Xeon Platinum | 36 | 72 | – | 1.53 |
| c5.2xlarge | Intel Xeon Platinum | 8 | 16 | – | 0.34 |

**JACC-GPU:** An optimized GPU-based implementation that uses our algorithm to compute the similarity matrix. The sketch construction stage is the same as SF-GPU, as this approach yields essentially the same behavior as using Algorithms 1–4. In the similarity computation stage, we use the same pivot-based approach described in Algorithm 5, which reduces global memory traffic compared to SF-GPU. Each CUDA block reads one pivot sketch into on-chip shared memory, and the sketch-merge operation is performed in parallel on the block threads.

**JACC-FPGA:** The FPGA-based implementation of the accelerator described in Section 4, which uses Algorithms 1–5 to build the sketches and compute the similarity matrix.

When evaluating the performance of the threshold-based algorithm described in Section 3.3.2, we use a parallel software implementation developed by ourselves using the Bonsai HLL sketch library written by the authors of Dashing, which is available at https://github.com/dnbaker/bonsai. We compare this software implementation to accelerated SF-GPU, JACC-GPU, JACC-FPGA implementations, in which the host computes the selection criterion and invokes the accelerator kernels using Algorithm 5.

We performed all experiments in the cloud using Amazon Web Services Elastic Compute Cloud (AWS EC2) instances. Table 1 summarizes the characteristics and price of the machines used in our experiments, valid as of June 2022. For the FPGA, GPU and software implementations, we aimed to select machines with similar price per hour of computation. Our FPGA-accelerated implementation runs on an AWS EC2 F1 instance (f1.2xlarge), which features an accelerator board with a Xilinx XCVU9P Ultrascale+ FPGA. The host computer supports 8 parallel hardware threads on an Intel Xeon processor. Our GPU-based implementations run on a g5.4xlarge instance tailored with an NVIDIA A10G Ampere GPU and an AMD EPYC host with 16 parallel hardware threads. For our comparisons to a software implementation, we used a compute-optimized c5.9xlarge instance with a more advanced Intel Xeon Platinum processor that supports 36 parallel hardware threads. We also used a less-expensive c5.2xlarge instance, which features the same number of vCPUs as the F1 host. We include it to illustrate the impact of the host processing power, as discussed in Section 5.3. All instances use GP3 storage [52] with a maximum of 16,000 input/output (I/O) operations per second. In order to assign the same storage cost to all machines [53], we limit the peak storage throughput to 212 MiB/s, which is the maximum supported by the f1.2xlarge instance [54].

## 5.1. HLL sketch parameters

Our first experiment evaluates the impact of the sketch size on the estimation of the Jaccard coefficient, compared to its true value. The sketch size depends on the number of buckets, which is $2^p$, and the number of bits per bucket. We evaluate the results produced by our hardware implementation of the HLL sketch and by the implementation provided by Dashing. Our HLL implementation uses the estimation correction for low sketch occupancy described in Section 3.1. We also implemented the HLL++ sketch, which uses a more complex estimator and has been reported to produce more accurate results than HLL [49], but in our experiments it produced the same estimations of the Jaccard coefficient. Consequently, we only report the results obtained with HLL.

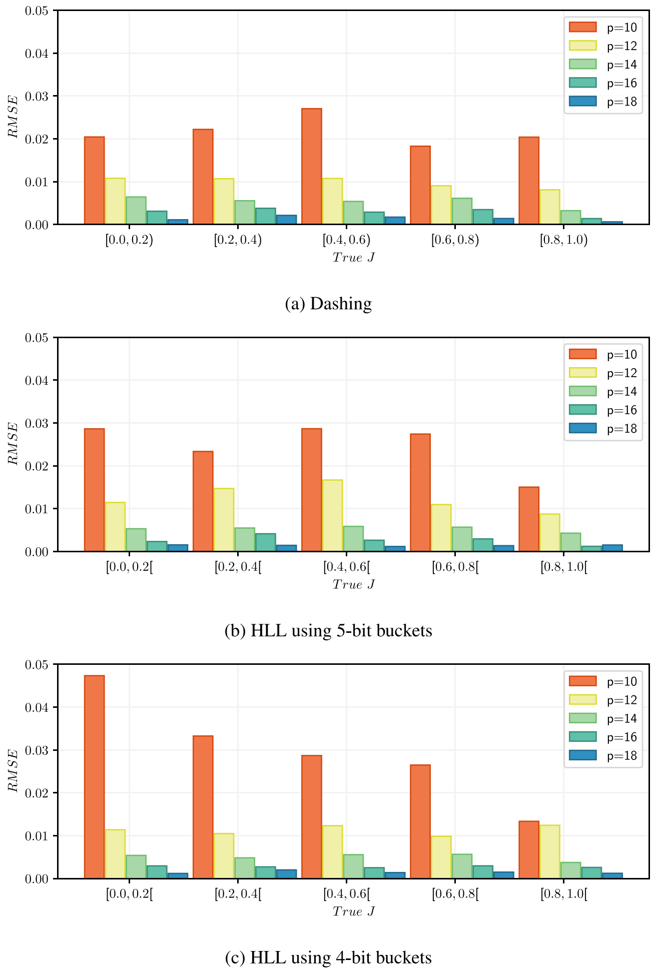
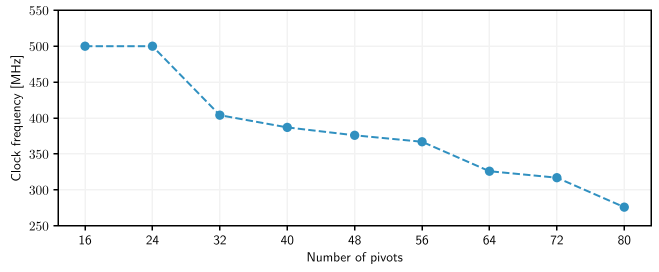
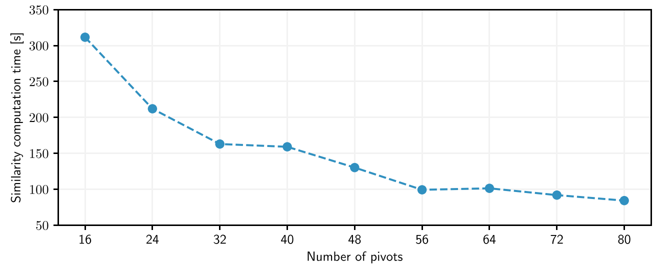


**Fig. 6.** Root-mean-square error (RMSE) of the Jaccard similarity using cardinalities estimated with HLL sketches, using exact cardinalities as the reference. We tested 5 values of $p$ between 10 and 18 for the sketches. The error was computed using the same 400 genomes pairs used by Dashing [8].

Fig. 6 shows the Root-Mean-Square Error (RMSE) of the Jaccard estimation compared to its true value, for Dashing and our hardware accelerator. We used 5 values of $p$ between 10 and 18. We use the same 400 genome pairs used by the authors of Dashing in their own evaluation [8]. These genome pairs encompass the complete range of Jaccard values between 0 and 1. The graphs in Fig. 6 show the RMSE for different values of $p$, for 5 ranges of the Jaccard value. Fig. 6(a) shows the RMSE achieved by Dashing. Because Dashing was conceived as a software solution, it uses a fixed number of 8 bits per bucket. In our FPGA-based accelerator, we can choose the number of bits per bucket, and in general we aim to choose the smallest possible value, as this allows us to maximize the number of buckets that we can simultaneously load on a single memory read operation, as discussed in Section 4.2. Figs. 6(b) and 6(c) show the RMSE obtained by our HLL implementations using 4 and 5 bits per bucket. As expected, the estimation error decreases as we increase the value of $p$. Moreover, even



(a) Clock frequency of the similarity computation kernel.



(b) Execution time of the similarity computation kernel.

**Fig. 7.** Performance of the similarity computation kernel versus the number of pivots, using the complete RefSeq dataset. (a) Clock frequency in MHz. (b) Kernel execution time in seconds.

though for values of $p \leq 12$ the estimation error is higher for a smaller number of bits per bucket, when $p > 12$ the RMSE with 4 bits per bucket is equivalent to that achieved by using 5 bits and by the 8-bit Dashing implementation. Moreover, the RMSE in this case is less than 0.01 across the entire range of Jaccard values. Therefore, we chose to use sketches of $2^{14}$ buckets with 4 bits per bucket in our current implementation, as this allows us to achieve greater parallelism when reading and merging sketch data during the similarity computation stage. However, our design is parameterized to easily synthesize sketches of different sizes, limited only by the constraints imposed by the hardware, such as memory bus width and routing resources.

### 5.2. Number of pivots

The number of pivots used by the kernel controls the parallelism achieved during the similarity computation stage of our algorithm, and directly affects its execution time. When computing $V$ rows of the similarity matrix, each kernel call computes $V \times L + V \times (V-1)/2$ similarities, where $V$ is the number of pivots and $L$ is the number of row sketches in the kernel call. However, because the computation is performed in parallel for the $V$ pivots, the execution time is proportional to $L + V$. Thus, a larger value of $V$ reduces the number of kernel calls and the time required to compute the similarity matrix.

Theoretically, the maximum useful number of pivots is 128, which is the number of clock cycles required by the accelerator to read a single sketch of $2^{14} \times 4$ bits, using the maximum memory bus width of 512 bits. However, increasing the number of pivots increases the logic and on-chip memory resources used by the accelerator, reducing the maximum clock frequency at which the implementation can operate. More importantly, because each row sketch read from external memory is merged with all the pivots in parallel, increasing the number of pivots makes it more difficult to route the data buses that distribute bucket values within the accelerator. With more than 80 pivots, the Vitis synthesis tool is unable to route the design in the XCVU9P FPGA available on the f1.2xlarge instance.

Fig. 7 shows the performance of the similarity computation kernel for a different number of pivots between 16 and 80, when computing the complete similarity matrix of the RefSeq dataset. The figure plots the kernel clock frequency and execution time on the f1.2xlarge instance. Ideally, doubling the number of pivots should reduce the execution time of the kernel to approximately one half. However, a design with more pivots requires more hardware resources and more complex routing, reducing the maximum clock frequency at which the design can run on the accelerator. Indeed, Fig. 7(a) shows a consistent reduction of the kernel clock frequency as we increase the number of pivots, starting at 500 MHz for 16 and 24 pivots, down to 276 MHz for 80 pivots. Nevertheless, Fig. 7(b) shows that the impact on execution time of the greater parallelism enabled by adding pivots is larger than the reduction in clock frequency. The execution time decreases less than linearly with the number of pivots, but the minimum time is still achieved with 80 pivots. However, as explained above, the XCVU9P FPGA does not provide sufficient routing resources to implement the design with more than 80 pivots. Consequently, we configure our accelerator to use the maximum of 80 pivots supported by our current hardware platform and design tools.

### 5.3. Performance

In this section, we evaluate the performance of our solution and compare it to equivalent software and GPU-accelerated implementations, running on different machines.

In our first experiment, we run the two stages of the algorithm separately on our hardware platforms and measure the execution time using the wall clock function on the host computer. We run the algorithm on the complete RefSeq dataset described at the beginning of Section 5 and, during the similarity computation stage, we compute the complete similarity matrix for the dataset. At the beginning of the sketch construction phase, the host sorts the genome files by size to improve the load balance of the producer threads of Algorithm 1. To provide a comparison between the performance of each accelerator and its host, we also run Dashing on the host machine of the f1.2xlarge and g5.4xlarge instances.

Table 2 reports the execution times in seconds. We perform 10 runs on each platform and report the median execution time. The median absolute deviation is less than 0.2% of the median value in all cases during the sketch construction stage, and less than 0.1% during similarity computation. During the first stage, JACC-FPGA on the f1.2xlarge instance builds the HLL sketches for all the genome multisets and computes their cardinality in approximately 39.8 min, which is about 2.9 times faster than Dashing performing the same task on the host computer of the f1.2xlarge instance. It is also twice as fast as Dashing running on the compute-optimized c5.2xlarge instance. Running on the 36-vCPU c5.9xlarge instance in 37.6 min, Dashing builds the sketches about 10% faster than JACC-FPGA. Likewise, ST-GPU and JACC-GPU on the g5.4xlarge instance build the sketches in the same 37.6 min as Dashing on the c5.9xlarge instance. The GPU-accelerated solutions build the sketches about 10% faster than Dashing running on their own g5.4xlarge host.

During the second stage, both Dashing and the FPGA/GPU-accelerated machines read from disk the sketches and cardinalities computed in the first stage, and merge all the sketch pairs to compute the cardinality of their union and, subsequently, their Jaccard similarity. As shown in Table 2, JACC-FPGA computes the similarity matrix in only 85 s. Comparatively, Dashing takes more than 6.5 h on the f1.2xlarge host, more than 4.3 h on the c5.2xlarge instance, and almost 1.4 h on the c5.9xlarge. Thus, the accelerator is 58 times faster than Dashing running on the fastest machine, and more than 277 times faster than the same

**Table 2**
Execution time of the sketch construction and similarity computation stages with the complete RefSeq dataset, using different algorithms and AWS EC2 instances. We report the performance of JACC-FPGA on an f1.2xlarge instance, SF-GPU/JACC-GPU on a g5.4xlarge instance. We report the performance of Dashing on the compute-optimized c5.2xlarge and c5.9xlarge instances, and also on the f1.2xlarge and g5.4xlarge hosts. The execution time is reported in seconds and, during similarity computation, the algorithm computes the complete similarity matrix.

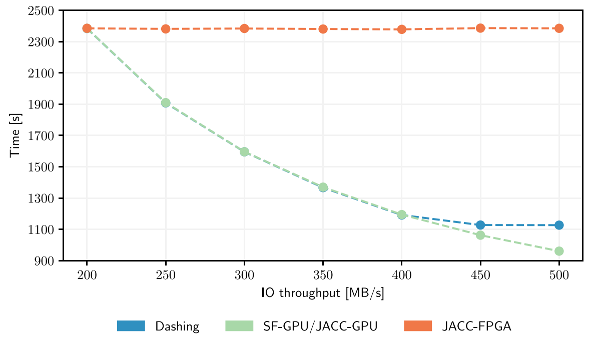
| Implementation | Instance | Sketch construction (s) | Similarity computation (s) |
|---|---|---|---|
| JACC-FPGA | f1.2xlarge | 2388 | 85 |
| JACC-GPU | g5.4xlarge | 2257 | 340 |
| SF-GPU | g5.4xlarge | | 2315 |
| Dashing | c5.9xlarge | 2257 | 4954 |
| Dashing | f1.2xlarge | 6847 | 23,571 |
| Dashing | g5.4xlarge | 2452 | 8325 |
| Dashing | c5.2xlarge | 4756 | 15,517 |

software running on the f1.2xlarge host computer. The SF-GPU implementation runs in 38.6 min while JACC-GPU, which uses the same pivot-based Algorithm 5 as JACC-FPGA to reduce traffic to external memory, computes the similarity matrix in just 5.7 min on the same hardware. Still, JACC-FPGA is more than 4 times faster than JACC-GPU and more than 27 times faster than ST-GPU. The large speedup achieved by JACC-FPGA compared to the other implementations is due to the large scale of temporal and spatial parallelism exploited by the accelerator architecture, which allows it to compute 320 sketch merges simultaneously (80 pivots merge with 4 row sketches each).

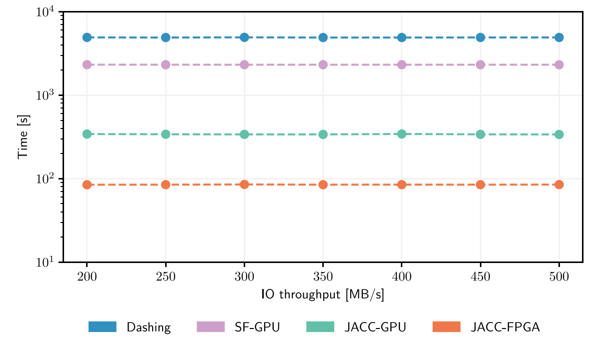
An important observation that explains the execution times during sketch construction in the fastest implementations (JACC-FPGA, JACC-GPU, and Dashing on the c5.9xlarge instance), is that the total size of the compressed files in the RefSeq dataset is 464.4 GiB. At the maximum disk throughput of 212 MiB/s, the minimum time required to read the dataset is 2,243 s. The execution times in the fastest instances are very close to this minimum, thus we can hypothesize that these implementations are currently I/O bound and their computation is almost completely overlapped with disk access. The slightly higher execution time of JACC-FPGA could be explained by considering that the maximum I/O throughput of the f1.2xlarge instance is also 212 MiB/s, while for g5.4xlarge and c5.9xlarge it is 594 MiB/s and 1,187 MiB/s, respectively. Therefore, the latter two instances are operating well below their maximum throughput, and only the storage limits their I/O bandwidth.

To validate the assumption that the stage construction is I/O bound, we performed an experiment where we gradually increased the maximum disk throughput from 200 MiB/s to 500 MiB/s, which also increases the storage cost [54]. Fig. 8(a) shows the execution time of the sketch construction stage on the three fastest implementations as a function of the disk throughput. As expected, the performance of JACC/FPGA remains constant, because the I/O throughput of the f1.2xlarge host is limited to 212 MiB/s. However, the other two implementations reduce their execution times as the bandwidth increases. Beyond 400 MiB/s, Dashing becomes compute-bound on the c5.9xlarge instance, and its sketch construction time does not decrease below 1,100 s. On the other hand, JACC-GPU remains I/O bound during the entire experiment and continues to improve its performance as the disk throughput increases. Fig. 8(b) shows a semi-logarithmic graph that plots the execution time of the distance computation stage as a function of the disk throughput. In this case, the execution time remains constant for all implementations. This is expected, as the sketch data read from disk during this stage amounts to less than 2 GiB, therefore all implementations are compute-bound.

Fig. 9 shows a breakout of the execution time on the f1.2xlarge host. In the sketch construction stage, the time was measured



(a) Sketch construction.



(b) Similarity computation.

**Fig. 8.** Performance of the sketch construction and similarity computation stages versus the maximum I/O throughput of the storage device attached to the host. During sketch construction, the execution time of JACC-GPU and Dashing on the c5.9xlarge instance decrease as the disk throughput increases. The execution time of JACC-FPGA does not decrease, because the I/O throughput of the f1.2xlarge host is limited to 212 MiB/s. During similarity computation, the execution time of the 3 implementations is compute-bound, and thus independent of the I/O throughput.

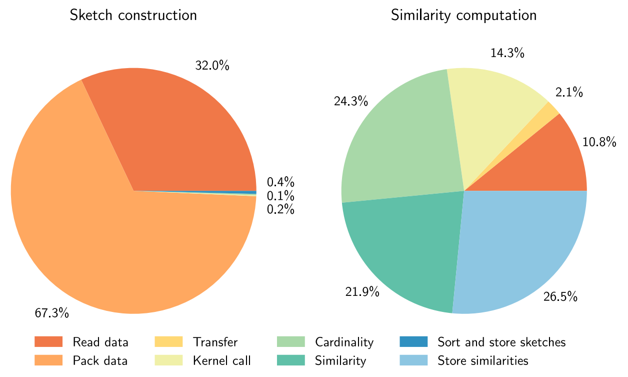


**Fig. 9.** Profile of the execution time of the two stages of our hardware-accelerated implementation of the algorithm, measured on the F1 host. Time was measured in the consumer thread for the sketch construction stage, and in the main thread for the similarity computation stage. The kernel calls include copying data to and from the accelerator buffers and waiting for the kernel to finish execution. Because the kernel calls are asynchronous, the execution time on the accelerator is mostly overlapped with tasks running on the host.

on the consumer thread. We use asynchronous kernel calls to the accelerator, which allows us to overlap virtually 100% of the execution time on the FPGA with processing on the host side. Indeed, less than 0.2% of the time is spent on kernel calls, which include transferring data to/from the accelerator buffer and waiting for the FPGA. Although we overlap disk reads with

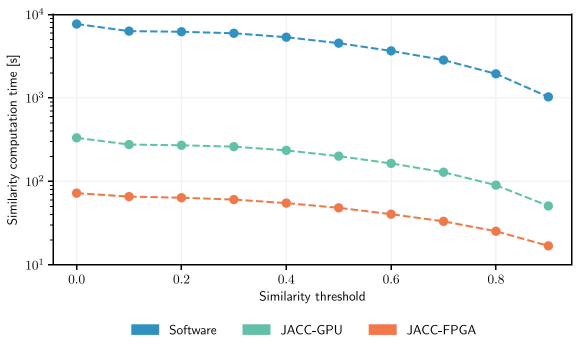


**Fig. 10.** Execution time of Algorithm 5 using the selection criterion described in Section 3.3.2, as a function of the similarity threshold. We programmed the software implementation using the Bonsai HLL sketch library used in Dashing, which exploits thread-level parallelism and vector extensions. The software implementation runs on the c5.9xlarge instance, JACC-GPU runs on the g5.4xlarge instance, and JACC-FPGA runs on f1.2xlarge. In all cases, we subtract the time used to initially read and sort sketch files and, in the case of JACC-GPU and JACC-FPGA, upload the sketch data to the accelerator.

host processing, the consumer threads spends 32% of the time waiting for data from the producer threads, which confirms that the sketch-construction stage is I/O bound. In the similarity computation stage, the time was measured in the main thread. Here, the host spends 10.8% of the time reading the sketches from disk and 26.5% storing the similarity matrix. Of the remaining 62.7% of the time, 14.3% is spent on kernel calls and 48.4% on the host. This is relevant, because, although the computation of the similarity matrix is transferred to the accelerator and parallelized effectively, a significant fraction of the time is still spent on the f1.2xlarge host, which is severely underpowered compared to the C5 and G5 instances. This is evident in the execution times of Dashing during similarity computation reported in Table 2, which show that the f1.2xlarge host is significantly slower than even c5.2xlarge, despite both having the same number of vCPUs.

### 5.4. Selection criterion performance

In this section, we evaluate the impact of using the selection criterion presented in Section 3.3.2 to reduce the number of similarity computations when we are only interested in retrieving the genome pairs with a Jaccard coefficient above a specified threshold. We measure the impact of the selection criterion on JACC-GPU, JACC-FPGA, and a multithreaded software implementation of Algorithm 5 running on the c5.9xlarge instance. We programmed this implementation in C++ using the Bonsai sketch library, which provides SIMD support for HLL sketches. Because the concepts of $V$ pivots and $D$ simultaneously-streaming sketches are intrinsic to the accelerator, we use $V = D = 1$ in the software implementation, but parallelize the same loops with OpenMP using the 36 available vCPUs. In all cases, we use the selection criterion to limit the number of iterations of the loop in line 12 of Algorithm 5.

Our software implementation reads the cardinalities and HLL sketches generated by Dashing for each genome of the RefSeq dataset, sorts them by cardinality, and then executes multiple queries to compute the Jaccard similarity for the genome pairs of interest, varying the threshold between 0 and 0.9. Reading and sorting the sketch files takes 81 s on the c5.9xlarge instance. In the case of the FPGA/GPU implementations, the host reads the sketches from a single file, which was generated and sorted during the sketch construction stage. The host uploads the sketch data to the accelerator and then executes the similarity queries using the same threshold values as the software implementation.

**Table 3**
FPGA resource utilization per kernel.

| Resource | Sketch construction | Similarity computation | Available |
|---|---|---|---|
| LUTs | 92,077 | 349,315 | 1,180,984 |
| Registers | 111,800 | 354,420 | 2,364,480 |
| BRAMs | 324 | 179 | 2,160 |
| DSP slices | 1,536 | 0 | 6,840 |

Because the sketch data is already sorted and consolidated onto a single file, the sketch read and upload operation takes about 11 s on the f1.2xlarge and g5.4xlarge instances.

Fig. 10 shows a semi-logarithmic graph of the execution time of each similarity query on the 3 implementations, as a function of the threshold $h$ used in the selection criterion of Eq. (9). In the graph, a threshold value $h = 0$ means that the algorithm does not exclude any genomes and computes the entire similarity matrix. As expected, the execution time on all implementations decreases when the threshold increases, because the selection criterion the number of iterations of the loop in line 12 of Algorithm 5, thus reducing the number of Jaccard coefficients that all implementations compute. For threshold values up to $h = 0.4$, our software implementation is slower than Dashing computing the complete similarity matrix from cached sketches, because Dashing exploits the parallelism available in the processor more efficiently than our implementation. For higher thresholds, the selection criterion yields better performance and, with $h = 0.8$, our software solution is 2.5 times faster than Dashing and 4 times faster than itself when computing the complete similarity matrix.

The impact on execution time in JACC-GPU and JACC-FPGA is similar to the software, but the benefits diminish for threshold values larger than 0.7. One of the reasons is that the selection criterion is computed for the pivot with the largest cardinality of each set, which introduces some overhead for the rest of the pivots that end up computing more similarities than needed. The second and more important reason is that the execution times in the accelerator for $h > 0.7$ are smaller than 30 s in JACC-FPGA and smaller than 100 s in JACC-GPU, making the fixed kernel call overhead more significant. Still, with $h = 0.8$, JACC-FPGA computes the similarities in 25 s, which is 2.9 times faster than computing the complete matrix. Similarly, JACC-GPU runs in 90 s with $h = 0.8$, which represents a speedup of 3.7x compared to computing the entire similarity matrix.

### 5.5. Hardware resource utilization

Table 3 shows the FPGA resources used by the kernels that implement the two stages of the algorithm. For the sketch construction kernel, the resources with the highest utilization are the DSP slices, which implement arithmetic operations such as multiplications and additions. The kernel implements 64 MurmurHash3 modules (8 sketch construction modules with 8 HLL sketches each), each requiring 24 DSPs: 10 for each 64-bit multiplication in Algorithm 9 and 4 in other operations. This accounts for 22.4% of the DSPs available on the XCVU9P FPGA. The sketch construction kernel uses 256 BRAM modules (on-chip memory blocks) to implement the sketches and 68 BRAMs in various memory interfaces and input/output buffers, which in total account for 15% of the BRAMs available on the FPGA. Other resources show a utilization below 8%.

The similarity computation kernel occupies 29.5% of the LUTs, which are mainly used to implement logic and arithmetic operations with small inputs to merge HLL sketches. The kernel also uses 15% of the registers, most of them to pipeline the merge operations and other computations. The kernel also uses 8% of the available BRAMs, 2 in each of the 80 pivots and 19 BRAMs in memory interfaces. Although the FPGA has considerable spare resources available to increase the number of pivots, the synthesis tools are unable to route the design on this FPGA with more than 80 pivots, as discussed in Section 5.2.

## 6. Conclusions

In this paper, we have presented a heterogeneous architecture and parallel algorithm that uses hardware acceleration to compute the similarity matrix between a set of genomes using the FPGA-as-a-service paradigm in the cloud. Our solution treats genome data as multisets of k-mers and uses HLL sketches to estimate the Jaccard similarities between the sets. Our architecture uses the massive hardware resources available on modern FPGAs to exploit coarse and fine-grained parallelism and pipelining to accelerate sketch operations such as construction, merge, and cardinality estimation. We use streaming algorithms to maximize performance by balancing the on-chip storage, large computational capacity, and limited external memory bandwidth of FPGA-based hardware accelerators. We also propose a selection criterion that, based solely on the cardinalities of individual genome multisets computed during sketch construction, can be used to reduce the amount of computation required to obtain the similarities of those genome pairs with a Jaccard coefficient above a user-defined threshold.

Our experimental results, tested on AWS EC2 instances with the RefSeq dataset, show that hardware acceleration is effective at speeding up both the construction of the HLL sketches and the computation of the similarity matrix. Compared to the performance of Dashing running on the host computer of the f1.2xlarge instance, the accelerator is 2.9 times faster when building the sketches and more than 277 times faster when computing the similarity matrix. However, processing power and input/output bandwidth limitations of the f1.2xlarge host computer limit the performance of the accelerator during the sketch construction stage of the algorithm. Consequently, during this stage our heterogeneous solution is 10% slower than Dashing running on a c5.9xlarge instance or a GPU-based implementation of our algorithms running on a g5.4xlarge instance, both of which have a similar price-per-hour to f1.2xlarge. Still, when computing the similarity matrix from the sketch data, the accelerator is 58 faster than Dashing and 4 times faster than the optimized GPU implementation. As the number of genomes in the dataset increases, we can expect that hardware acceleration will have an even stronger impact, because the complexity of computing the similarities is quadratic in the number of genomes.

Using our selection criterion to retrieve only those genome pairs whose Jaccard similarity exceeds a predefined threshold can significantly reduce the computation time on software and hardware-accelerated implementations. Our results show that, when using a threshold of 0.8, the similarity computation achieves speedups between 2.9x and 3.7x compared to computing the complete similarity matrix. The criterion is simple to apply, requiring only the cardinalities of each individual genome multiset, which are computed during the sketch construction stage. Moreover, without considering the probabilistic nature of cardinality estimation with HLL sketches, the criterion is deterministic in terms of set-operations. This guarantees that all pairs that comply with the threshold will be retrieved by the algorithm, provided that the sketches are designed to estimate cardinalities with sufficiently high precision.

Running on the Xilinx UltraScale+ XCVU9P FPGA on the f1.2xlarge instance, our kernels use less than 30% of its arithmetic, logic and on-chip storage resources. This utilization depends only on the size of the sketches and the degree of parallelism that the accelerator is allowed to exploit, subject to the memory bandwidth available. The spare hardware resources allow us to integrate other kernels on the same design for further processing of the genome data. Moreover, the Vitis programming platform allows dynamic reconfiguration of the FPGA logic, which also allows us to design more complex processing pipelines where complete kernels are replaced on-the-fly as the algorithms process the data.

Our work can be extended by using the FPGA-as-a-service paradigm to create a cloud service that performs cardinality estimation and similarity computation between user-provided sequences or between the user data and a previously-defined reference dataset. Such a service can be useful for clustering, mapping sequence reads, or to find the most similar elements in the dataset. We can also adapt the accelerator architecture to compute similarities between metagenomic samples, which can be useful to accelerate applications in taxonomic profiling.

**Algorithm 6:** HLL insertion algorithm

```
Input: sketch $A$, $k\text{-}mer$
Output: sketch $A$
1 $h \leftarrow MurMurHash3(k\text{-}mer)$
2 $v_1 \leftarrow < h_{63}, h_{64-p} >_2$
3 $v_2 \leftarrow < h_{63-p}, h_0 >_2$
4 $A[v_1] \leftarrow max\{A[v_1], ldz(v_2) + 1\}$
```

## CRediT authorship contribution statement

**Javier E. Soto:** Conceptualization, Software, Validation, Investigation, Writing – original draft, Visualization. **Cecilia Hernández:** Conceptualization, Methodology, Validation, Writing – review & editing, Supervision. **Miguel Figueroa:** Conceptualizaton, Methodology, Validation, Writing – review & editing, Supervision, Project administration, Funding acquisition.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgments

This work was supported by ANID through Fondecyt Regular grant 1220960, Basal Funds FB0001, and Doctorado Nacional scholarship.

## Appendix. Other algorithms

### A.1. HyperLogLog

Algorithm 6 shows the insertion operation, which updates the HLL sketch with a new k-mer. First, it computes a 64-bit MurMurHash3 function (Algorithm 9 in Appendix A.3) with the k-mer. It indexes the sketch with $p$ bits of the hash, where $p$ is the precision parameter. Then, it computes the number of leading zeros ($ldz$) of the rest of the hash, plus one, and updates the sketch if this number is higher than the current value.

Algorithm 7 shows the estimation operation, which is executed once all the k-mers in a set have been inserted. Line 5 computes the harmonic mean in Eq. (3). Lines 3 and 5 estimate

**Algorithm 7:** HLL estimation algorithm

```
   Input: sketch $A$
   Output: cardinality estimation $C_{HLL}$
 1 Let
 2     $m = 2^p$
 3 $\alpha_A = 0.7213/(1+\frac{1.079}{m})$ assuming $p \geq 7$
 4 $Z \leftarrow \sum_{i=0}^{m-1} 2^{-A[i]}$
 5 $C_{HLL} \leftarrow \alpha_A \frac{m^2}{Z}$
 6 if $C_{HLL} \leq 2.5m$ then
 7     $n_z \leftarrow zeros\_counter(A)$
 8     $C_{HLL} \leftarrow m \log\left(\frac{m}{n_z}\right)$
 9 return $C_{HLL}$
```

**Algorithm 8:** Canonical k-mer computation

```
   Input: $k\text{-}mer$
   Output: canonical k-mer $k_{canon}$
 1 Let
 2     $C_1 = 0x3333333333333333$
 3     $C_2 = 0x0F0F0F0F0F0F0F0F$
 4     $C_3 = 0x00FF00FF00FF00FF$
 5     $C_4 = 0x0000FFFF0000FFFF$
 6 $k_0 \leftarrow ((k\text{-}mer \gg 2) \wedge C_1) \vee ((k\text{-}mer \wedge C_1) \gg 2)$
 7 $k_1 \leftarrow ((k_0 \gg 4) \wedge C_2) \vee ((k_0 \wedge C_2) \gg 4)$
 8 $k_2 \leftarrow ((k_1 \gg 8) \wedge C_3) \vee ((k_1 \wedge C_3) \gg 8)$
 9 $k_3 \leftarrow ((k_2 \gg 16) \wedge C_4) \vee ((k_2 \wedge C_4) \gg 16)$
10 $k_4 \leftarrow (k_3 \gg 32) \vee (k_2 \gg 32)$
11 $k_5 \leftarrow \neg k_4$
12 $k_6 \leftarrow k_5 \gg 2$
13 $k_{canon} \leftarrow min\{k_6, k\text{-}mer\}$
14 return $k_{canon}$
```

the cardinality using Eq. (4). Lines 7 and 8 apply the correction in Eq. (5) when the estimation is less than $2.5 \times 2^p$, using the number of empty buckets (which contain a zero).

## A.2. Canonical k-mer

To compute the canonical representation of a k-mer, Algorithm 8 chooses between its original representation and its reverse complement, based on their lexicographical value. The k-mer is represented as a $2k$-bit word, where $k$ is the number of bases. Algorithm 8 is based on the Kraken [55] implementation of reverse complement computation for $k = 31$, with minor changes to improve its performance on the FPGA accelerator. The algorithm uses a set of bit-masks $C_1 - C_4$ and bitwise AND, OR and NOT operations to compute the reverse complement of the k-mer (inverting the order of its bases). It then selects the minimum value using an unsigned integer comparison.

## A.3. MurMurHash3 hash function

Algorithm 9 computes the 64-bit MurMurHash3 [56] function, also known as *fmix64* in some implementations. This algorithm can be used for k-mers of any length up to 64 bits. The function treats the canonical k-mer as an unsigned integer and uses multiplications and bitwise XOR/shift operations to compute the hash value.

**Algorithm 9:** MurMurHash3 64 bits

```
   Input: $k\text{-}mer$
   Output: hash value $h$
 1 $k_0 \leftarrow k\text{-}mer \gg 33$
 2 $k_1 \leftarrow k_0 \oplus k\text{-}mer$
 3 $k_2 \leftarrow 0xff51afd7ed558ccd \times k_1$
 4 $k_3 \leftarrow k_2 \gg 33$
 5 $k_4 \leftarrow k_3 \oplus k_2$
 6 $k_5 \leftarrow 0xc4ceb9fe1a85ec53 \times k_4$
 7 $k_6 \leftarrow k_5 \gg 33$
 8 $h \leftarrow k_6 \oplus k_5$
 9 return $h$
```

## References

[1] B.D. Ondov, T.J. Treangen, P. Melsted, A.B. Mallonee, N.H. Bergman, S. Koren, A.M. Phillippy, Mash: fast genome and metagenome distance estimation using MinHash, Genome Biol. 17 (1) (2016) 1–14.

[2] S. Behera, J.S. Deogun, E.N. Moriyama, MinIsoClust: Isoform clustering using minhash and locality sensitive hashing, in: Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics, 2020, pp. 1–7.

[3] K. Berlin, S. Koren, C.-S. Chin, J.P. Drake, J.M. Landolin, A.M. Phillippy, Assembling large genomes with single-molecule sequencing and locality-sensitive hashing, Nature Biotechnol. 33 (6) (2015) 623–630.

[4] S. Koren, B.P. Walenz, K. Berlin, J.R. Miller, N.H. Bergman, A.M. Phillippy, Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation, Genome Res. 27 (5) (2017) 722–736.

[5] M. Forc, W. Kuśmirek, R.M. Nowak, De Novo genome assembly for third generation sequencing data, in: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2018, Vol. 10808, International Society for Optics and Photonics, 2018, 108083D.

[6] D. Moi, L. Kilchoer, P.S. Aguilar, C. Dessimoz, Scalable phylogenetic profiling using MinHash uncovers likely eukaryotic sexual reproduction genes, PLoS Comput. Biol. 16 (7) (2020) e1007553.

[7] A. Criscuolo, On the transformation of MinHash-based uncorrected distances into proper evolutionary distances for phylogenetic inference, F1000Research 9 (2020).

[8] D.N. Baker, B. Langmead, Dashing: fast and accurate genomic distances with HyperLogLog, Genome Biol. 20 (1) (2019) 265, http://dx.doi.org/10.1186/s13059-019-1875-0.

[9] N.A. O'Leary, M.W. Wright, J.R. Brister, S. Ciufo, D. Haddad, R. McVeigh, B. Rajput, B. Robbertse, B. Smith-White, D. Ako-Adjei, et al., Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation, Nucleic Acids Res. 44 (D1) (2016) D733–D745.

[10] NCBI reference sequence (RefSeq) database release 212, 2022, URL https://ftp.ncbi.nlm.nih.gov/refseq/release/release-notes/RefSeq-release212.txt. (Accessed 1 June 2022).

[11] A. Almeida, S. Nayfach, M. Boland, F. Strozzi, M. Beracochea, Z.J. Shi, K.S. Pollard, E. Sakharova, D.H. Parks, P. Hugenholtz, et al., A unified catalog of 204,938 reference genomes from the human gut microbiome, Nature Biotechnol. 39 (1) (2021) 105–114.

[12] C. Li, D. Tian, B. Tang, X. Liu, X. Teng, W. Zhao, Z. Zhang, S. Song, Genome Variation Map: a worldwide collection of genome variations across multiple species, Nucleic Acids Res. 49 (D1) (2020) D1186–D1191, http://dx.doi.org/10.1093/nar/gkaa1005.

[13] Z.D. Stephens, S.Y. Lee, F. Faghri, R.H. Campbell, C. Zhai, M.J. Efron, R. Iyer, M.C. Schatz, S. Sinha, G.E. Robinson, Big data: Astronomical or genomical? PLoS Biol. 13 (7) (2015) 1–11, http://dx.doi.org/10.1371/journal.pbio.1002195.

[14] X. Zhao, BinDash, software for fast genome distance estimation on a typical personal laptop, Bioinformatics 35 (4) (2018) 671–673, http://dx.doi.org/10.1093/bioinformatics/bty651.

[15] A. Saavedra, H. Lehnert, C. Hernández, G. Carvajal, M. Figueroa, Mining discriminative K-mers in DNA sequences using sketches and hardware acceleration, IEEE Access 8 (2020) 114715–114732, http://dx.doi.org/10.1109/ACCESS.2020.3003918.

[16] J.E. Soto, T. Krohmer, C. Hernandez, M. Figueroa, Hardware acceleration of k-mer clustering using locality-sensitive hashing, in: 2019 22nd Euromicro Conference on Digital System Design, DSD, IEEE, 2019, pp. 659–662.

[17] N. Cadenelli, Z. Jakšić, J. Polo, D. Carrera, Considerations in using OpenCL on GPUs and FPGAs for throughput-oriented genomics workloads, Future Gener. Comput. Syst. 94 (2019) 148–159, http://dx.doi.org/10.1016/j.future.2018.11.028.
[18] T.D. Wu, J. Reeder, M. Lawrence, G. Becker, M.J. Brauer, GMAP and GSNAP for genomic sequence alignment: Enhancements to speed, accuracy, and functionality, Methods Mol. Biol. 1418 (2016) 283–334.
[19] Y. Gao, Y. Liu, Y. Ma, B. Liu, Y. Wang, Y. Xing, abPOA: an SIMD-based C library for fast partial order alignment using adaptive band, Bioinformatics 37 (15) (2020) 2209–2211, http://dx.doi.org/10.1093/bioinformatics/btaa963.
[20] N. Ahmed, H. Mushtaq, K. Bertels, Z. Al-Ars, GPU accelerated API for alignment of genomics sequencing data, in: 2017 IEEE International Conference on Bioinformatics and Biomedicine, BIBM, 2017, pp. 510–515, http://dx.doi.org/10.1109/BIBM.2017.8217699.
[21] S. Goswami, K. Lee, S. Shams, S.-J. Park, GPU-accelerated large-scale genome assembly, in: 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2018, pp. 814–824, http://dx.doi.org/10.1109/IPDPS.2018.00091.
[22] H. Li, A. Ramachandran, D. Chen, GPU acceleration of advanced k-mer counting for computational genomics, in: 2018 IEEE 29th International Conference on Application-Specific Systems, Architectures and Processors, ASAP, 2018, pp. 1–4, http://dx.doi.org/10.1109/ASAP.2018.8445084.
[23] R. Kobus, A. Müller, D. Jünger, C. Hundt, B. Schmidt, MetaCache-GPU: Ultra-fast metagenomic classification, in: 50th International Conference on Parallel Processing, Association for Computing Machinery, New York, NY, USA, 2021, http://dx.doi.org/10.1145/3472456.3472460.
[24] A. Subramaniyan, J. Wadden, K. Goliya, N. Ozog, X. Wu, S. Narayanasamy, D. Blaauw, R. Das, Accelerated seeding for genome sequence alignment with enumerated radix trees, in: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture, ISCA, 2021, pp. 388–401, http://dx.doi.org/10.1109/ISCA52012.2021.00038.
[25] Y. Turakhia, G. Bejerano, W.J. Dally, Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly, ACM SIGPLAN Not. 53 (2) (2018) 199–213.
[26] Y. Turakhia, S.D. Goenka, G. Bejerano, W.J. Dally, Darwin-WGA: A co-processor provides increased sensitivity in whole genome alignments with high speedup, in: 2019 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2019, pp. 359–372, http://dx.doi.org/10.1109/HPCA.2019.00050.
[27] N. Mcvicar, C.-C. Lin, S. Hauck, K-mer counting using bloom filters with an FPGA-attached HMC, in: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, 2017, pp. 203–210, http://dx.doi.org/10.1109/FCCM.2017.23.
[28] L. Guo, J. Lau, Z. Ruan, P. Wei, J. Cong, Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between FPGA and GPU, in: 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, 2019, pp. 127–135, http://dx.doi.org/10.1109/FCCM.2019.00027.
[29] Amazon, Amazon EC2 F1 instances, 2022, URL https://aws.amazon.com/ec2/instance-types/f1. (Accessed 15 February 2022).
[30] D. Fujiki, S. Wu, N. Ozog, K. Goliya, D. Blaauw, S. Narayanasamy, R. Das, SeedEx: A genome sequencing accelerator for optimal alignments in subminimal space, in: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2020, pp. 937–950, http://dx.doi.org/10.1109/MICRO50266.2020.00080.
[31] L. Wu, D. Bruns-Smith, F.A. Nothaft, Q. Huang, S. Karandikar, J. Le, A. Lin, H. Mao, B. Sweeney, K. Asanović, D.A. Patterson, A.D. Joseph, FPGA accelerated INDEL realignment in the cloud, in: 2019 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2019, pp. 277–290, http://dx.doi.org/10.1109/HPCA.2019.00044.
[32] X. Wang, Y. Niu, F. Liu, Z. Xu, When FPGA meets cloud: A first look at performance, IEEE Trans. Cloud Comput. (2020) 1, http://dx.doi.org/10.1109/TCC.2020.2992548.
[33] T.J. Ham, D. Bruns-Smith, B. Sweeney, Y. Lee, S.H. Seo, U.G. Song, Y.H. Oh, K. Asanovic, J.W. Lee, L.W. Wills, Genesis: A hardware acceleration framework for genomic data analysis, in: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture, ISCA, 2020, pp. 254–267, http://dx.doi.org/10.1109/ISCA45697.2020.00031.
[34] P. Indyk, R. Motwani, Approximate nearest neighbors: towards removing the curse of dimensionality, in: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, 1998, pp. 604–613.
[35] Y. Bussi, R. Kapon, Z. Reich, Large-scale k-mer-based analysis of the informational properties of genomes, comparative genomics and taxonomy, PLoS One 16 (10) (2021) e0258693.
[36] A.M. Moustafa, A. Lal, P.J. Planet, Comparative genomics in infectious disease, Curr. Opin. Microbiol. 53 (2020) 61–70.
[37] S. Lipworth, K.-D. Vihta, K. Chau, L. Barker, S. George, J. Kavanagh, T. Davies, A. Vaughan, M. Andersson, K. Jeffery, et al., Ten-year longitudinal molecular epidemiology study of Escherichia coli and Klebsiella species bloodstream infections in Oxfordshire, UK, Genome Med. 13 (1) (2021) 1–13.
[38] A. Criscuolo, A fast alignment-free bioinformatics procedure to infer accurate distance-based phylogenetic trees from genome assemblies, Res. Ideas Outcomes 5 (2019) e36178.
[39] C. Gostinčar, Towards genomic criteria for delineating fungal species, J. Fungi 6 (4) (2020) 246.
[40] C.-P. Cheng, K.-L. Lan, W.-C. Liu, T.-T. Chang, V.S. Tseng, DeF-GPU: Efficient and effective deletions finding in hepatitis B viral genomic DNA using a GPU architecture, Methods 111 (2016) 56–63, http://dx.doi.org/10.1016/j.ymeth.2016.07.020, Big Data Bioinformatics.
[41] S.D. Goenka, Y. Turakhia, B. Paten, M. Horowitz, SegAlign: A scalable GPU-based whole genome aligner, in: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, 2020, pp. 1–13, http://dx.doi.org/10.1109/SC41405.2020.00043.
[42] A. Zeni, G. Guidi, M. Ellis, N. Ding, M.D. Santambrogio, S. Hofmeyr, A. Buluç, L. Oliker, K. Yelick, LOGAN: High-performance GPU-based X-Drop long-read alignment, in: 2020 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2020, pp. 462–471, http://dx.doi.org/10.1109/IPDPS47924.2020.00055.
[43] D. Tong, V.K. Prasanna, Sketch acceleration on FPGA and its applications in network anomaly detection, IEEE Trans. Parallel Distrib. Syst. 29 (4) (2018) 929–942, http://dx.doi.org/10.1109/TPDS.2017.2766633.
[44] M. Tang, M. Wen, J. Shen, X. Zhao, C. Zhang, Towards memory-efficient streaming processing with counter-cascading sketching on FPGA, in: 2020 57th ACM/IEEE Design Automation Conference, DAC, 2020, pp. 1–6, http://dx.doi.org/10.1109/DAC18072.2020.9218503.
[45] A. Saavedra, C. Hernández, M. Figueroa, Heavy-hitter detection using a hardware sketch with the countmin-CU algorithm, in: 2018 21st Euromicro Conference on Digital System Design, DSD, 2018, pp. 38–45, http://dx.doi.org/10.1109/DSD.2018.00022.
[46] M. Chiosa, T.B. Preußer, G. Alonso, SKT: A one-pass multi-sketch data analytics accelerator, Proc. VLDB Endow. 14 (11) (2021) 2369–2382.
[47] J.E. Soto, P. Ubisse, Y. Fernández, C. Hernández, M. Figueroa, A high-throughput hardware accelerator for network entropy estimation using sketches, IEEE Access 9 (2021) 85823–85838, http://dx.doi.org/10.1109/ACCESS.2021.3088500.
[48] P. Flajolet, E. Fusy, O. Gandouet, F. Meunier, Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm, in: Discrete Mathematics and Theoretical Computer Science, Discrete Mathematics and Theoretical Computer Science, 2007, pp. 137–156.
[49] S. Heule, M. Nunkesser, A. Hall, HyperLogLog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm, in: Proceedings of the 16th International Conference on Extending Database Technology, EDBT '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 683–692, http://dx.doi.org/10.1145/2452376.2452456.
[50] P. Deutsch, J.-L. Gailly, RFC 1950 (Informational).
[51] D. Adas, R. Friedman, A fast wait-free multi-producers single-consumer queue, in: 23rd International Conference on Distributed Computing and Networking, ICDCN 2022, Association for Computing Machinery, New York, NY, USA, 2022, pp. 77–86, http://dx.doi.org/10.1145/3491003.3491004.
[52] Amazon EBS volume types, 2022, URL https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html. (Accessed 1 June 2022).
[53] Amazon EBS volume pricing, 2022, URL https://aws.amazon.com/ebs/pricing/. (Accessed 1 June 2022).
[54] Amazon EBS–optimized instances, 2022, URL https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-optimized.html. (Accessed 1 June 2022).
[55] D.E. Wood, S.L. Salzberg, Kraken: ultrafast metagenomic sequence classification using exact alignments, Genome Biol. 15 (3) (2014) 1–12.
[56] A. Appleby, Smhasher & murmurhash, 2022, URL https://github.com/aappleby/smhasher. (Accessed 15 February 2022).

**Javier E. Soto** received the B.Sc. (2014) and M.Sc. (2016) degrees in Electrical Engineering from Universidad de Concepción, Chile, where he is currently pursuing a Ph.D. degree in Electrical Engineering. His research interests include high-performance computing, hardware accelerators, embedded systems, and heterogeneous architectures.

**Cecilia Hernández** is an Assistant professor at Universidad de Concepción. She received the M.Sc. degree in Computer Science from the University of Washington and a Ph.D. degree in Computer Science from Universidad de Chile. Her research interests include compressed structures, data mining, parallel and distributed algorithms, and bioinformatics.

**Miguel Figueroa** is a Professor of Electrical Engineering at Universidad de Concepción, Chile. He obtained an undergraduate degree in Electronics Engineering (1990) and an M.Sc. degree in Electrical Engineering (1997) from Universidad de Concepción. He received the M.Sc. (1999) and Ph.D. (2005) degrees in Computer Science and Engineering from the University of Washington in Seattle, WA, USA. His current research interests include hardware accelerators for bioinformatics and network traffic monitoring, reconfigurable computing for high-performance embedded systems, VLSI circuits for low-power video processing and computer vision, and high-speed processing circuits for quantum cryptography and radioastronomy. He has authored and co-authored more than 100 scientific publications.