Genesis: A Hardware Acceleration Framework for Genomic Data Analysis

Tae Jun Ham* David Bruns-Smith† Brendan Sweeney† Yejin Lee* Seong Hoon Seo*
U Gyeong Song* Young H. Oh‡ Krste Asanovic† Jae W. Lee* Lisa Wu Wills§

*Seoul National Univerity †University of California, Berkeley ‡Sungkyunkwan University §Duke University

{taejunham, yejinlee, andyseo247, thddnrud, jaewlee}@snu.ac.kr
{bruns-smith, brs, krste}@berkeley.edu younghwan@skku.edu lisa@cs.duke.edu

***Abstract*—In this paper, we describe our vision to accelerate algorithms in the domain of genomic data analysis by proposing a framework called *Genesis* (genome analysis) that contains an interface and an implementation of a system that processes genomic data efficiently. This framework can be deployed in the cloud and exploit the FPGAs-as-a-service paradigm to provide cost-efficient secondary DNA analysis. We propose conceptualizing genomic reads and associated read attributes as a very large relational database and using extended SQL as a domain-specific language to construct queries that form various data manipulation operations. To accelerate such queries, we design a *Genesis* hardware library which consists of primitive hardware modules that can be composed to construct a dataflow architecture specialized for those queries.**

As a proof of concept for the *Genesis* framework, we present the architecture and the hardware implementation of several genomic analysis stages in the secondary analysis pipeline corresponding to the best known software analysis toolkit, GATK4 workflow proposed by the Broad Institute. We walk through the construction of genomic data analysis operations using a sequence of SQL-style queries and show how *Genesis* hardware library modules can be utilized to construct the hardware pipelines designed to accelerate such queries. We exploit parallelism and data reuse by utilizing a dataflow architecture along with the use of on-chip scratchpads as well as non-blocking APIs to manage the accelerators, allowing concurrent execution of the accelerator and the host. Our accelerated system deployed on the cloud FPGA performs up to 19.3× better than GATK4 running on a commodity multi-core Xeon server and obtains up to 15× better cost savings. We believe that if a software algorithm can be mapped onto a hardware library to utilize the underlying accelerator(s) using an already-standardized software interface such as SQL, while allowing the efficient mapping of such interface to primitive hardware modules as we have demonstrated here, it will expedite the acceleration of domain-specific algorithms and allow the easy adaptation of algorithm changes.

***Index Terms*—genome sequencing, genomic data analysis, hardware accelerator, FPGA, SQL**

I. Introduction

As the democratization of wet lab sequencing technology drives down sequencing cost, the cost and runtime of data analysis are becoming more significant [40]. The Human Genome Project [33] released the first human genome assembly in 2001, following 15 years of work at a total cost of \$3B. As depicted in Figure 1, the price of sequencing a single human genome has decreased from more than \$100, 000, 000 in 2001 to about \$1000 in 2019, far outpacing Moore's Law [42]. Just in July 2019, Veritas Genetics announced a further price reduction of whole-genome sequencing to below \$600 and predicts that the cost will drop to between \$100 and \$200 in the next two years [4]. This cost reduction has motivated investments in precision and genomic medicine, where the knowledge of an individual's genome is used to guide the prevention and the treatment of diseases [56], and has enabled genomic research projects that have collected data across tens of thousands of individuals [34], [65].

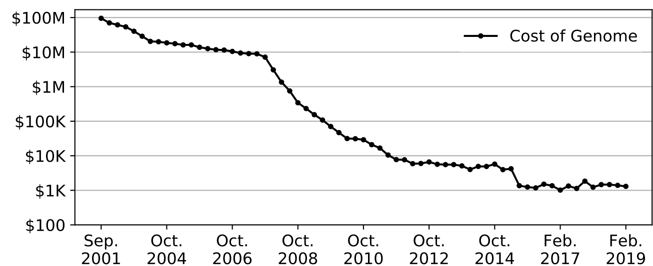

Fig. 1. The cost of sequencing a human genome has dropped by a hundred thousand fold, from 2001 to 2019. This data is replicated from the National Human Genome Research Institute's survey of genome sequencing costs [42].

An article published in *PLoS Biology* quantitatively claimed that genomics is projected to produce over 250 exabytes of sequence data per year by 2025, far surpassing the current major generators of big data such as YouTube (∼1-2 exabytes/year) and Twitter (∼1.36 petabytes/year) [54]. With the aforementioned big data generation comes challenges in genomic data acquisition, storage, distribution, and analysis. We focus our effort on addressing the *efficient analysis* of genomic data, in particular, identifying genomic variants in each individual genome, as it is one of the most computationally complex and demanding pipelines.

Genomic data processing algorithms are composed of a mixture of specific algorithms as well as generic data manipulation operations. For example, the most popular genome sequencing workflow, Broad Institute's [7] Genome Analysis

ToolKit 4 (GATK4) Best Practices [11], consists of stages implementing specific algorithms such as read alignment and variant calling as well as stages performing generic data manipulations such as mark duplicates and base quality score recalibration. Thus far, most prior work focused on the hardware acceleration of specific algorithms utilized in genome sequencing. Darwin [58], GenAx [21], and others [12], [13], [25], [36], [59], [62] accelerate the algorithms used in the alignment stage while several works [6], [26], [38] accelerate pair-HMM algorithms utilized in the variant calling stage. Such specialized accelerators, targeting a specific implementation of a particular genome sequencing pipeline stage, have demonstrated multiple orders of magnitude speedups and energy efficiency improvements. With these specific algorithm accelerations in place, the remaining un-accelerated analysis stages that contain generic data manipulation operations become the bottleneck and a large portion of the genomic analysis execution time, making them good targets for acceleration pursuant to Amdahl's law.

An important aspect of genomic data analysis is that the algorithms are still being refined and special care is needed when proposing hardware acceleration. For example, INDEL realignment was the major performance bottleneck in the now deprecated GATK3 and thus a hardware accelerator targeting the stage was proposed [62]. However, GATK4 does not utilize this stage with its updated variant calling algorithms rendering the proposal largely suitable for legacy pipelines. Similarly, accelerators targeting the pair-HMM algorithms used in the variant calling stages of GATK4 are likely being replaced by the DNN-based algorithm for the same stage [51]. Noting the rapid changes in specific algorithms, we argue that designing accelerators for the generic data manipulation portions of the pipelines is just as important, if not more, than designing accelerators for the specific algorithms.

Our work aims to address the computational challenges of genomic data analysis by introducing *Genesis*, a flexible acceleration framework that targets generic data manipulation operations commonly used in genomic data processing. We observe that there are ample similarities between the operations needed to perform genomic analytics and traditional big data analytics. We propose treating genomic data as traditional data tables and use extended SQL as a domain-specific language to process genomic analytics. Conceptualizing the genomic data as a very large relational database allows us to reason about the algorithms and transform genomic data processing stages into simple extended SQL-style queries. Once the queries (a.k.a. genomic analysis stages) are constructed, *Genesis* facilitates the translation of the queries into hardware accelerator pipelines using the *Genesis* hardware library that accelerates primitive operations in database and genomic data processing. We design and deploy *Genesis*-generated accelerators on Amazon EC2 F1 instances [3].

As a proof of concept for the *Genesis* framework, we accelerate the data preprocessing phase in GATK4 Best Practices. We show how multiple stages of the preprocessing phase can be represented in extended SQL-style queries and demonstrate that the accelerated system targeting these queries provides a significant performance improvement and cost saving over a commodity CPU hardware platform.

This paper makes the following contributions:

- We present *Genesis*, a hardware acceleration framework for generic data manipulation operations in genomic data processing pipelines. With our framework, users represent genomic data manipulation operations with standardized SQL and user-supplied custom operations. Then, *Genesis* aids the rapid translation of such representations to a performant, cloud-deployment-ready hardware accelerator. Specifically, *Genesis* provides a hardware library which contains configurable hardware modules that accelerate common operations in a relational database as well as genomic-data-specific operations. With this hardware library, users can easily stitch these hardware modules to construct a pipeline targeting the specific query. The resulting hardware is automatically augmented with parallelism, efficient memory accesses, and easy-to-use high-level user interfaces between the host and the accelerator.
- As a proof of concept, we implement and deploy hardware accelerators for the three stages of the preprocessing pipeline in GATK4, *mark duplicate*, *metadata update*, and *base quality score recalibration* using *Genesis*. We show that our accelerated system deployed on the Amazon Web Services (AWS) cloud FPGA achieves up to 19.3× speedup and 15× cost reduction compared to software running on a commodity multi-core CPU platform.

II. Background

We provide a brief background on genomics for the readers to better understand our proposal.

Genome. A genome is an organism's complete set of DNAs. For a human genome, it contains information for all 22 paired chromosomes and a sex chromosome pair. Each chromosome is represented as a sequence of DNA base pairs, where each base pair is expressed as a single character (i.e., `A, T, C, G`) representing a DNA nucleotide base. Typical sequence lengths for human chromosomes range from 50 million to 250 million base pairs, and a human genome contains roughly 3 billion base pairs in total.

Genomic Analysis. Genomic analysis uses genomic features such as a DNA sequence to identify variations from a biological sample containing a full copy of the DNA against a reference genome. Our work focuses on genomic analysis through the Next Generation Sequencing (NGS) technology, the *de-facto* technology for the whole genome analysis. In this process, fragmented DNA samples are read by a NGS wet lab instrument. Raw sensor data from the instrument are processed through an equipment-specific proprietary software (or hardware) and the instrument outputs processed data called *reads*. Reads contain multiple fragments from a sequence of base pairs and a sequence of quality scores where a single quality score represents the machine's confidence of the corresponding base pair measurement. This process of post-measurement analysis is called the *primary analysis* and the outcome of the primary analysis is an input to the *secondary analysis*. *Secondary*

Position	1234567 89012 (111 above 10–12)	CIGAR (alignment metadata)
Reference	ACGTAAC CAGTA	
Read 1	AGGTAACACGGTA	(7**M**, 1**I**, 5**M**)
Read 2	TTTTAAC CA_TA	(3**S**, 6**M**, 1**D**, 2**M**)

Fig. 2. Example Read Alignment.

analysis is a process of identifying genomic variants. Since it is very computationally demanding, this is what most computer software/hardware research (including ours) focuses on. Once these genomic variants are identified, they can be used to analyze the specific characteristics of this DNA (e.g., disease risk).

Genomic Read Data. Aligned read data is the most important data type in the preprocessing phase of genomic analysis. Aligned read data contains the chromosome identifier (that this read is aligned to), the position within the chromosome that the mapped read starts from, the sequence of base pairs, and the sequence of quality scores. In addition, the aligned read contains metadata about the alignment called CIGAR (Concise Idiosyncratic Gapped Alignment Report). CIGAR summarizes the alignment information about the read and is represented with a list of (integer, **operation type**) pairs with the integer indicating the number of base pairs and the operation indicating aligned (**M**), inserted (**I**), deleted (**D**), or soft-clipped (**S**).

For example, Read 1 in Figure 2 has a CIGAR of (7**M**, 1**I**, 5**M**). This indicates that the read's first seven base pairs are aligned (7**M**), the next single base (i.e., A) is inserted and not present in the reference (1**I**), and the next five bases are again aligned (5**M**). Note that aligned (**M**) can either mean a match or a mismatch to the reference in the actual base pair. Read 2 in Figure 2 has a CIGAR of (3**S**, 6**M**, 1**D**, 2**M**). Here, **S** indicates that the aligner ended up not considering this soft-clipped portion to determine the alignment. **D** represents a deletion where the base pair present in the reference sequence is not present in the read. In addition to CIGAR, there are several other fields in an aligned read such as flags, mapping quality, information about the paired read, etc. While our hardware handles those fields appropriately, we omit the detailed explanations and implementation descriptions for conciseness.

III. Genesis Framework

A. Overview

Genesis is a framework which enables users to rapidly deploy the hardware accelerators for the data manipulation operations in genomic data analysis workloads. Based on the observation that genomic data can be conceptualized as a very large relational database and most data manipulation operations in genome sequencing can be expressed in database operations, *Genesis* adopts an extended SQL-style interface that allows users to express the target data manipulation operations. One important aspect here is that such SQL representations (i.e., queries) can also be represented as a series of relational operators (often called the logical query plan). *Genesis* framework facilitates the process of hardware accelerator designs for such queries by providing a set of configurable hardware modules that directly map to many common relational operators (e.g., join, aggregate, etc.). With the *Genesis* hardware library, a user can easily configure a simple dataflow architecture by stitching them using high-level hardware description language (e.g., Chisel [14]). The resulting design is augmented with our framework, which provides support for parallelism, performant memory system accesses, and high-level APIs which enable the user to easily manage the data and control communications between the host (a commodity `x86_64` Xeon server) and the hardware accelerator. For deployment and evaluation, this design is compiled into Verilog and synthesized, placed, and routed into an FPGA image. This image then can be deployed with a local or a cloud FPGA such as an AWS F1 instance.

In this section, we describe how a user can represent a genomic analysis pipeline as an extended SQL query (Section III-B), how a set of modules in the *Genesis* hardware library (Section III-C) can be assembled to construct a hardware accelerator pipeline targeting the query (Section III-D), how a user can utilize our simple high-level APIs to manage the accelerators from the host (Section III-E), and how a user can extend our framework by adding custom computation modules (Section III-F).

B. Genesis SQL Interface

TABLE I
EXAMPLE GENOMICS DATA TABLE REPRESENTATIONS.

Field	Data Type	Remarks
Read Table (READS)		
CHR	`uint8_t`	Chromosome Identifier (1,..., 22, X, Y)
POS	`uint32_t`	Leftmost position of this aligned read
ENDPOS	`uint32_t`	Rightmost position of this aligned read
CIGAR	`uint16_t[CLEN]`	An array of cigar operations
SEQ	`uint8_t[LEN]`	Sequence of base pairs (e.g., A,C,G,T)
QUAL	`uint8_t[LEN]`	Sequence of quality scores
Reference Table (REF)		
CHR	`uint8_t`	Chromosome Identifier (1,..., 22, X, Y)
REFPOS	`uint32_t`	Starting position of reference segment
SEQ	`uint8_t[PSIZE+LEN]`	Sequence of base pairs
IS_SNP	`bool[PSIZE+LEN]`	A bit indicating whether the corresponding position is a known site of variation

Representation. *Genesis* utilizes SQL as a domain-specific language to represent the target genomic analysis operation for acceleration. The genomic read and reference data are represented as tables with schemas shown in Table I. Reads are represented as rows in the table and attributes associated with each read are represented as columns. In our evaluated data set (i.e., Illumina [27] sequencer reads for a specific human), there are more than 700 million reads and each read has up to 151 base pairs (i.e., `LEN` = 151). A reference sequence is fragmented into many segments and each segment is represented as a row in the reference table. We configure a single row in the reference table to have about 1M base pairs (i.e., `PSIZE` = 1M). The reference table also has a column named `IS_SNP` (Single-Nucleotide Polymorphism), which is a bitmap representation of the known sites of base pair variations (i.e., this position's

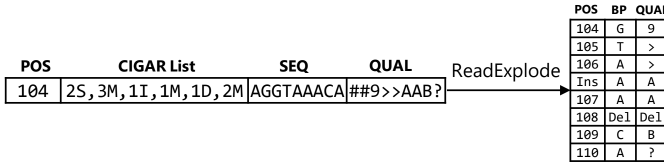

Fig. 3. ReadExplode Operation Example.

base pair is known to vary across individuals) for the reference fragment. When a locus (or location, position) is identified as an SNP known site, a base pair mismatch at that locus is expected and therefore does not count as an error. This column is used in the base quality score recalibration algorithm described in Section IV-D.

Partitioning. A common data manipulation operation in genomic data processing is to compare a read's base pair sequence to the corresponding reference sequence. For such operations, it is often helpful to pre-partition both the read table and the reference table to multiple smaller tables based on their chromosome identifier (`CHR`), and positions (i.e., `READS.POS` or `REF.REFPOS`) to make finding reads' corresponding reference fragments easier. We partition the read table first by `CHR` and then again by `POS` so that the nth partition for a chromosome would have reads whose positions fall in the interval $[(n-1)\times$ `PSIZE`, $n\times$ `PSIZE`]. We also partition the reference table so that the nth partition for a chromosome would have reference sequence whose positions fall in the interval $[(n-1)\times$ `PSIZE`, $n\times$ `PSIZE+LEN`]. For both tables, we assign a unique partition ID (`PID`) to each partition. With this partitioning scheme, a read can simply obtain a relevant reference sequence fragment by inspecting the reference table with the same `PID`.

Supported SQL-style Operations. *Genesis* supports common SQL operations such as `Select`, `Where`, `GroupBy`, `Join`, `Limit` (used to select a subset of rows), `Count`, and `Sum`. In addition, we support two additional operations `PosExplode` and `ReadExplode`. `PosExplode(COL,INITPOS)` converts an array in a single row of a single column (`COL`) to multiple rows with an extra `POS` column that starts from the position `INITPOS` (`POS` is incremented by one for every row that is exploded). This is similar to the `PosExplode` operation that already exists in Hive QL [57] and Spark SQL [5]. `ReadExplode` is a genomics specific operation that is explained in the next paragraph. Lastly, we support iteration over rows with the `FOR Row IN Table` clause, which is similar to that of Oracle PL/SQL [46].

ReadExplode. This operation converts a read, stored as a single row in the READS table, to multiple, separate rows where each row contains the base, the corresponding quality score, and its position. Figure 3 shows the example operation of `ReadExplode`. `ReadExplode` requires `POS`, `CIGAR`, `SEQ`, and `QUAL` (optional) columns of a READS table as inputs. This operation converts individual base pairs and corresponding quality scores into separate rows utilizing its alignment information recorded in CIGAR (explained in Section II). In this example, two leftmost base pairs corresponding to 2**S** part of the CIGAR are clipped and thus not included in the output. The next three matching base pairs and corresponding quality

```
/* I1: Extract Reads and Reference Partition P */
CREATE TABLE ReadPartition AS
SELECT POS, ENDPOS, CIGAR, SEQ
FROM READS PARTITION (P)
CREATE TABLE ReferenceRow AS
SELECT POS, SEQ
FROM REF PARTITION (P)
/* I2: posExplode on ReferenceRow */
CREATE TABLE RelevantReference AS
PosExplode (ReferenceRow.SEQ, ReferenceRow.POS)
FROM ReferenceRow
DECLARE @rlen int
/* Iterate over Rows */
FOR SingleRead IN ReadPartition:
  SET @rlen = SingleRead.ENDPOS - SingleRead.POS)
  /* Q1: ReadExplode to convert a read into multi-row
       table where each row represents a base pair */
  CREATE TABLE #AlignedRead AS
  ReadExplode (SingleRead.POS, SingleRead.CIGAR,
       SingleRead.SEQ)
  FROM SingleRead
  /* Q2: Inner-Join two tables with the base pair's
       corresponding position as a key */
  CREATE TABLE #ReadAndRef AS
  SELECT #AlignedRead.SEQ, RelevantReference.SEQ
  FROM #AlignedRead
  INNER JOIN (SELECT * FROM RelevantReference LIMIT
       SingleRead.POS, rlen)
  ON #AlignedRead.POS = RelevantReference.POS
  /* Q3: Find the sum of matching base pairs */
  INSERT INTO Output
  SELECT SUM(#AlignedRead.SEQ == RelevantReference.SEQ)
  FROM #ReadAndRef
END LOOP;
```

Fig. 4. Example queries to find the number of bases in the read within the partition P that matches with the reference. This sequence of SQL queries are organized as two initialization steps (`I1` and `I2`) and three query steps (`Q1`, `Q2`, and `Q3`).

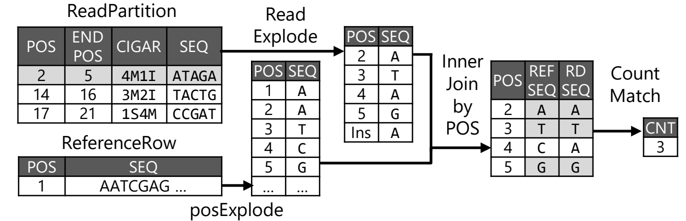

Fig. 5. Execution Flow of the Example Query.

scores are assigned the appropriate position values and included in the output as three separate rows. The following base pair `A` is inserted (**I**), and thus its reference position is marked as `Ins`, a special bit indicating that it is not in the reference. In the case of a deleted (**D**) base pair (the base pair with position 108 in the example), the reference position is included in the output, but the base pair column and the quality score column are marked as deleted (`Del`).

Example Query. We use an example to illustrate how to construct queries for a genomic data analysis operation and walk through the execution of the query using a high level block diagram. We want to find the number of bases that matches the reference for all reads whose partition ID is equal to the constant `P`. In this case, the user can represent this operation as a sequence of SQL queries as shown in Figure 4, which essentially follows the execution flow depicted in Figure 5. Step 1: the set of reads and the relevant reference

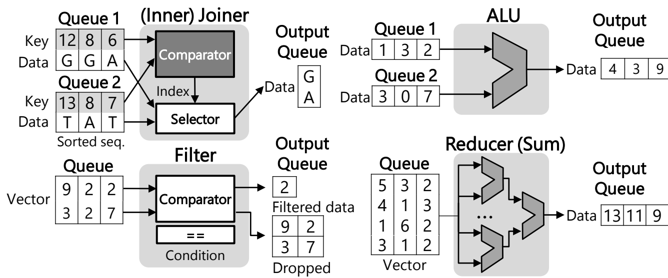

Fig. 6. Genesis Data Manipulation and Computation Modules.

with the partition ID `PID = P` are first extracted (`I1`). Step 2: the relevant reference row's base pair sequence is expanded into multiple rows with `PosExplode` (`I2`). Step 3: for each read in the `ReadPartition`, its base pairs are converted to a multi-row table with `ReadExplode` (`Q1`). Step 4: inner-join the `ReadExplode`'ed table and the subset of the `PosExplode`'ed reference row table (the subset is obtained with the `LIMIT base offset` clause) to obtain a joined table that allows us to extract base matching information (`Q2`). Step 5: the number of matching base pairs (i.e., a read's base pair is identical to the reference's base pair) are computed and inserted into the output table (`Q3`).

C. Genesis Hardware Modules

Overview. In *Genesis* framework, re-configurable hardware modules are assembled to accelerate genomic data manipulation operations. *Genesis* adopts a dataflow execution model where multiple independent modules are connected to each other via hardware queues. Each hardware module operates with a sequence of data called *streams*. A stream consists of many *data items*, each of which can contain multiple different types of fields. Each data item is also divided into multiple *flits*, where a single flit represents the atomic unit of data communication and operation. For example, when a sequence of reads forms a single stream, each read is a data item, and each base pair (or multiple base pairs), which is part of a base pair sequence in a read, is a flit. In general, each module consumes (or inspects) a single flit from its input queue(s) and generates a single output flit. The output flit is then inserted to the output queue, which will work as an input queue for the next module. Figure 6 shows some of the key *Genesis* hardware modules and we discuss each module in detail below.

Data Manipulation & Computation Modules

Joiner. Joiner merges flits from two input queues and produces a single output. For this module, a flit in an input queue should consist of the key field and the data field. In addition, the flits from the input queue should be supplied in ascending order of the key. Every cycle, this module compares keys of the flits from two input queues and either outputs or discards a single flit with the smaller key while leaving the other one intact. If both flits from two input queues have the same key, their data fields are merged (through concatenation). Specifically, this module can be configured to either perform an inner-join (i.e., discard flits without matching key), a left-join (i.e., discard flits from the second queue if it does not have a matching key in the first queue), or an outer-join (i.e., never discard flits).

Filter. Filter takes input data from a single queue, checks whether it matches the specified comparison condition (across fields or for a field and a constant), and outputs the item if and only if the item satisfies the specified condition.

Reducer. Reducer takes a sequence of data and performs a reduction operation (e.g., Sum, Max, Min, Count) with a reduction tree. For this module, a single flit can contain multiple values, and a reduction tree is utilized to obtain a single reduction result at a throughput of a single flit per cycle. Note that this module can also support reduction across multiple flits (i.e., reduction at an item granularity) and masked reduction, which means that a bit-mask (single bit per value) can be supplied to apply reduction on a subset of the data.

ALU. Stream ALU takes input data from a single or two input queues (or a single input queue and a constant item) and performs a relatively simple unary/binary ALU operation (e.g., NOT, ADD, SUB, CMP, AND, OR, etc.) with data from those queues. This module takes a single item from each queue, performs a binary operation, and outputs a single item. When a single item contains multiple values, the unary/binary operation is performed in an element-wise manner. Similar to a reducer, this module can also take a bit-mask sequence and conditionally perform unary/binary operations.

Memory & Scratchpad Memory (SPM) Access Modules

Memory Reader. Memory reader is in charge of reading contiguous data from memory and streaming the read data to the next module. Given a starting address and the total amount of data to read from memory, it continuously sends memory requests to memory at a memory access granularity (e.g., 64B) as long as its internal prefetch buffer is not full. At the same time, this module supplies the returned data from memory to the next module at a throughput of a single flit per cycle. Note that the flit granularity can be different from the memory access granularity.

Memory Writer. Memory writer is in charge of writing the data coming from an input queue to memory. It takes a single flit from the previous module per cycle and temporarily stores it to its internal buffer. Once its internal buffer size reaches the size of the memory access granularity (or a specific termination condition), it sends a write request to memory starting from the pre-configured starting address.

SPM Reader. SPM (Scratchpad Memory) reader simply takes an address from the input queue and outputs the scratchpad read result to the output queue. It can also be configured to read all elements in the interval when the starting address and the finishing address are provided. This module is also used to drain all of its content to the output queue when a drain signal is provided.

SPM Updater. SPM updater takes an address and the value from an input queue and updates the scratchpad memory. This module supports three operating modes. First, it can work like a memory writer which performs sequential writes to the SPM buffer when provided a starting address. It can also be

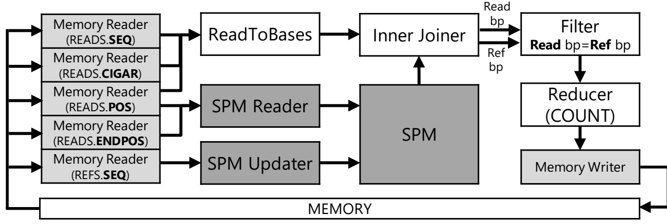

Fig. 7. Constructed Hardware Pipeline for the Example Query in Figure 4.

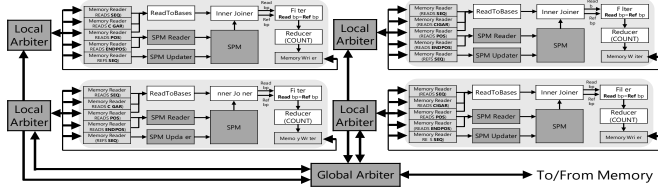

Fig. 8. Parallel Configurations of *Genesis* Hardware Pipelines. The pipeline in Figure 7 is replicated for four times.

configured to perform a random SPM write, which simply writes the value to the provided address. Finally, it can also be configured to perform a read-modify-write update with the provided modify function (e.g., add/subtract a constant). When configured to perform a read-modify-write update, the hardware needs to ensure that two flits with the same address will not be processed in these pipeline stages (i.e., read, modify, write) at the same time to avoid the RAW (read-after-write) hazard. For this purpose, our hardware buffers addresses that are being processed in these pipeline stages and checks if the incoming flit's address matches with any of these three buffered addresses. If so, the hardware prevents this incoming flit from starting the read stage.

Genomic Data Processing Modules

ReadToBases. This is the module that supports the `ReadExplode` operation explained in Section III-B. This module takes a sequence of `CIGAR`, `POS`, `SEQ`, and optionally `QUAL` values from the input queues and produces a `ReadExplode`'ed table. Each cycle, this module outputs the tuple of the reference position, the corresponding base, and the quality score. Here, as shown in Figure 3, the reference position may be `Ins` if the base is an inserted base. Similarly, the base and quality score fields can be `Del` if the base is deleted.

D. Construction of Hardware Pipeline with Genesis

Example Pipeline. *Genesis* accelerates the user-provided query by constructing a hardware pipeline using multiple *Genesis* hardware modules written in Chisel. For example, the SQL query in Figure 4 is translated to the hardware pipeline shown in Figure 7. The hardware pipeline in Figure 7 has five memory readers and each reader reads the data streams from `READS.POS`, `READS.ENDPOS`, `READS.CIGAR`, `READS.SEQ`, and `REFS.SEQ`. Three of these memory readers (the ones reading `READS.POS`, `READS.SEQ`, and `READS.CIGAR`) are connected to the ReadToBases module which generates a sequence of flits where each flit is a pair of a base and the corresponding reference position. This generated sequence is then provided as an input to the Joiner (configured to perform an inner-join). Unlike the reads data, the relevant reference data is mapped to an on-chip SPM to facilitate data reuse. A single memory reader (the one reading `REFS.SEQ`) is connected to the SPM Updater module so that it can initialize the SPM with data from memory. The contents from this SPM is retrieved with the SPM Reader which takes two inputs from the memory readers (the ones reading `READS.POS` and `READS.ENDPOS`), reads the SPM contents for the corresponding interval, and supplies the read data (i.e., reference base pairs) to the Joiner. The Joiner takes these two input sequences (i.e., one from the read, another from the reference), performs an inner-join, and passes the joined sequence to the Filter which compares two data fields (i.e., the base pair from the read and the base pair from the reference), and only outputs the matching items. Lastly, the Reducer module accumulates the number of matched base pairs and passes the outcome to the memory writer which stores the outcome to memory. The constructed pipeline is fully-pipelined and can process a single base pair per cycle.

Pipeline Construction. For now, our framework assumes that the process of translating SQL-style queries to the hardware pipeline is manual. However, we envision it to be automated in the near future. SQL queries can be easily parsed into a tree graph where each node represents a table (leaf node) or a relational/computational operator (non-leaf node) [47]. Since the *Genesis* hardware library (Section III-C) provides various modules for each relational operator and each genomics-specific operator, designing a hardware pipeline corresponding to such a query plan (or a tree graph) is rather simple. Specifically, each node in the graph can be mapped to a *Genesis* hardware module, and each edge in the graph is mapped to a hardware queue connecting these modules. For better resource allocation, the user can provide a hint to the translator so that frequently re-used tables are allocated to on-chip SPMs instead of off-chip memory (as in Figure 7).

Parallelism. A single pipeline is often insufficient to fully utilize the available memory bandwidth provided to the system. In order to fully utilize the available memory bandwidth and achieve high throughput, it is necessary to exploit abundant parallelism in genomic data processing operations through the use of multiple pipelines. Figure 8 shows how *Genesis* exploits parallelism through the use of multiple pipelines. *Genesis* treats each pipeline to be independent of each other except that they share memory interfaces and the command interfaces. This separation allows the utilization of different hardware pipelines targeting different operations to work together. As shown in Figure 8, input/output ports of all hardware pipelines' memory modules are first arbitrated by a local arbiter and then arbitrated again by one of the global arbiters, each of which is connected to one out of four memory channels in the system. The same structure applies to the command interface as well.

E. Genesis Application-Programmer Interface

To run *Genesis*-generated hardware pipeline, the user needs to configure the hardware memory readers and writers using the following C++ function.

```
void configure_mem (void* addr, int elemsize, int
len, string colname, int pipelineID)
```

Configuration. This blocking function needs to be invoked once for each memory reader and writer, which is in charge of reading/writing specific columns of the table. The function argument `addr` represents the address where the data for a column is located in the host address space, `elemsize` represents the size of an element for the column, `len` represents the number of rows that the column has, and `colname` represents the name of the column in the query. Lastly, `pipelineID` is necessary to specify the pipeline when there are multiple pipelines executing in parallel. Once invoked, this function copies the column data to the accelerator memory (if it is a memory reader) and configures the corresponding memory reader (or writer).

Execution and Completion. Once all memory readers and writers (for a specific pipeline) are configured, the user can execute the non-blocking call `void run_genesis(int pipelineID)` to start the execution. At this point, the user can run another non-blocking call `bool check_genesis(pipelineID)` to see if the accelerator execution is completed or use the blocking call `void wait_genesis(pipelineID)` to wait until the accelerator execution is finished. Once the accelerator execution completes, the user will then use `void genesis_flush(pipelineID)` to get the data back from the accelerator memory to the host memory address configured with `configure_mem(..)`. Note that the existence of these non-blocking calls is to allow the host CPU to perform useful work while the accelerator is running.

F. *Extending Genesis with Custom Operations*

While the standard SQL clauses, supported by the *Genesis* hardware library modules, are expressive enough to support many data operations in genomic analytics workloads, some data manipulation operations can benefit more from the capability to add a custom operation. *Genesis* allows a user to add a custom module that performs the desired computation. To add a custom module, a user needs to write a Chisel module with the provided interface which takes inputs from one or multiple streams and outputs data to a single output stream. Then the user can invoke this module using our SQL-style interface as follows: `EXEC ModuleName InputStream1 = _ InputStream2 = _ ··· InputStreamN = _`. This is similar to the user-defined function/procedure call semantics in many SQL implementations.

IV. Accelerating Genome Sequencing with Genesis

A. *GATK4 Best Practices Data Preprocessing Pipeline*

GATK4 Best Practices presents a step-by-step recommendation for the state-of-the-art genome variant discovery analysis (i.e., secondary analysis). In GATK4, the genome variant discovery process is divided into two phases. The first phase is the data preprocessing phase, which takes base pair sequences and the associated quality score sequences as inputs and refines

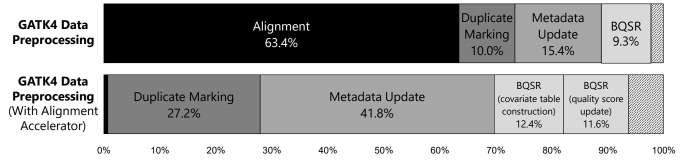

Fig. 9. Runtime breakdown of the GATK4 Best Practices data preprocessing pipeline on an AWS system with eight cores.

their alignment accuracy. The second phase is the variant discovery phase, which identifies and filters the variants or the differences of the measured genome from the reference genome. Here, the first phase is a single pipeline that precedes all types of variant discovery phases. For the second phase, there are many variant discovery pipelines, each targeting a different type of variant that can occur in a genome (e.g., germline variants, somatic variants, copy-number-variants, etc.). In this section, we focus on accelerating the data preprocessing phase since this phase often takes a substantially larger amount of time than the variant discovery phase (regardless of the type of the variant discovery). However, our proposed design can be utilized in several different data manipulation operations present in many variant discovery pipelines as well.

The data preprocessing phase consists of four major stages: alignment, mark duplicates, metadata update, and base quality score recalibration. The alignment stage takes a set of the reads (i.e., short base pair sequence and the associated quality score sequence) and maps each read to the appropriate position of the reference sequence. After the alignment, the mark duplicates step happens. In this step, all reads mapped to the exact same starting position are considered as duplicates, which resulted from the same DNA fragment, and thus all but a single read is removed from each duplicate set. In addition, this step also sorts all reads based on their starting positions. Following this step, a few metadata for each read is generated. Such metadata include information about the differences between a read and the corresponding reference sequence. Lastly, the base quality score recalibration (BQSR) stage refines the quality score (for each base pair) provided by the instrument. This stage is divided into two sub-stages: covariate table construction and quality score update. During the covariate table construction stage, the algorithm first goes through all base pairs and counts the error rates (i.e., mismatches with the reference sequence) across different potential sources of biases (e.g., the lane of the sequencing instrument used for measurement). Then, during the quality score update stage, the algorithm adjusts each base quality score based on the empirical error rates across different conditions and makes each quality score better match the empirical quality score.

Figure 9 shows the runtime breakdown of the GATK4 Best Practices data preprocessing pipeline (with the evaluation setting shown in Table II). As depicted, most of the runtime is spent on three major stages on a system with eight cores. Among the three major stages, the alignment takes the most time (i.e., 63.4 %). However, recent hardware accelerators such as GenAx [21] achieves over 4058K/reads throughput in the alignment stage. If we assume such throughput on the alignment

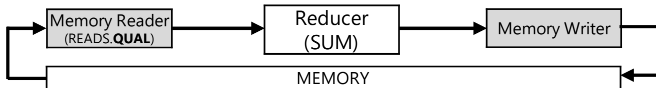

Fig. 10. Hardware for the Mark Duplicates Stage.

stage, the portion of time spent on the alignment stage shrinks to merely 0.7% on an eight core system. Consequently, the mark duplicates, metadata update, and the base quality score recalibration stages now account for the majority (i.e., 93%) of the runtime. As a proof of concept for our *Genesis* library, we design accelerators targeting various operations within the three stages: mark duplicates, metadata update, and base quality score recalibration.

B. Accelerating Mark Duplicates

Algorithm. The goal of the *mark duplicates* stage is to identify a set of reads originating from a single fragment of the DNA. Duplicate reads result from the PCR (polymerase chain reaction) amplification, which is part of the DNA sample preparation process. To identify a set of duplicate reads, the algorithm first generates a key value for each read. Specifically, the unclipped 5' prime positions of a read are used as a key for the read. To obtain this value, the `CIGAR` value is inspected and the number of soft clipped (**S**) bases at the front are subtracted from the `POS` value[1]. Once these keys are computed, the algorithm identifies sets of duplicate reads that share the same key. Among those reads sharing the same key, all but one with the highest sum of the quality scores are marked as duplicates. In addition to marking the duplicates, this stage also sorts all the reads by their aligned read start positions.
Acceleration. For this stage, our work focuses on the acceleration of the sum-of-quality score computation. Our hardware takes the `QUAL` column of each read as an input and computes the sum of quality scores for each row. The host core simply utilizes these sums of quality scores to determine duplicate reads among the ones sharing the same key.
Hardware Composition. For this step, the hardware performs a straightforward task of computing the sum of all quality scores. Figure 10 shows the hardware pipeline for this task. This is the most basic example of the *Genesis* pipeline which simply takes a stream of data from the input with a Memory Reader, performs a computation (i.e., sum reduction) with a Reducer, and stores the result back to memory with a Memory Writer. Here, the main benefit of hardware acceleration comes from the use of large parallelism (across quality-scores and across reads).

C. Accelerating Metadata Update

Algorithm. The *metadata update* stage in GATK4 (also called SetNmMdAndUqTags) calculates three specific types of metadata for each read: NM metadata, MD metadata, and UQ metadata. NM metadata represents the number of mismatches compared with the reference base pairs in this read. MD metadata is a specifically formatted string that enables the recovery of the reference base pair sequence by inspecting this metadata and the read base pair sequence. MD metadata represents the sequence of contiguous *matching* bases as a number and outputs the reference base pairs for the mismatched base pairs or the deleted base pairs. Since the inserted base pairs are not present in the reference, MD tag does not include any information about insertions. As an example, Read 1 in Figure 2 has a MD of `1C6A3` because it has a mismatch at the second base pair and the ninth base pair. Lastly, UQ metadata sums up the quality scores of the base pairs that mismatch the reference base pairs. This metadata essentially represents the likelihood that the read is erroneous. These updated metadata are utilized in the latter steps of the genomic data processing pipeline.

[1]In the paired-end sequencing technology, a key is generated per pair. The key for each read of a pair is concatenated to construct the key for a pair. One of the paired read is a *reverse* read and the number of soft-clipped (**S**) bases at the end is added to the `ENDPOS` value to obtain the unclipped 5' prime positions for such a read.

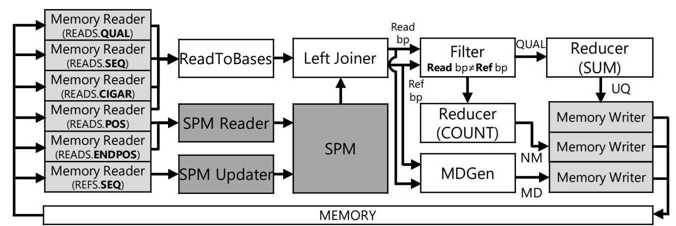

Fig. 11. Hardware for the Metadata Update Stage.

Acceleration. *Genesis* hardware takes `POS, ENDPOS, CIGAR, SEQ, QUAL` columns from reads data and relevant `REFPOS, SEQ` from the reference data as inputs and produces the computed NM, MD, UQ metadata as outputs. These metadata are attached to the original reads file in the software. For this task, the reference and reads are pre-partitioned in software (as explained in Section III-B) and a single invocation of the pipeline handles a single partition. The accelerator is invoked multiple times to finish processing the entire data set.

Hardware Composition. Figure 11 shows the *Genesis* hardware pipeline for this stage. In a way, this pipeline is similar to the example case covered in Section III. Specifically, the NM metadata computation is almost identical to the example query (Figure 4) except that it counts the number of mismatches, which includes the insertions and the deletions. The Joiner is configured to perform a left-join instead of an inner-join to preserve the insertion/deletion information. UQ metadata computation shares most of the front-end pipeline stages with NM metadata computation but instead of counting the number of mismatches, it performs a sum-reduction on filtered quality scores. MD metadata computation is performed by passing the outcome of the left-joiner to a custom module (MDGen) to generate the MD tag. This custom module simply takes the output of the left-joiner as an input stream and performs one of the following: i) if the read base pair and the reference base pair matches, increment the match counter or ii) if the base pairs do not match, output the match counter and print the reference base pair[2].

[2]In the case where a deleted base pair is present in the read, print ^ along with the reference base pair to indicate a deletion.

D. Accelerating Base Quality Score Recalibration

Algorithm. A genome sequencing instrument produces a quality score for each base pair it identifies (or measures, calls). This quality score represents the probability of the case where the sequencing machine correctly calls this base pair. However, it is known that these scores often do not match well with the empirical error rate. For example, when the average quality score for a particular read with 200 base pairs translates to 1% error rate, this read is expected to have about two base pair errors. However, due to various sources of systematic biases (e.g., the lane of the sequencing machine used to process this data), this expected error rate often does not match with the empirical results obtained by manually counting the number of mismatches between this read's base-pair sequences and the reference sequence.

To reduce the deviation of empirical quality scores from the machine-generated base quality scores, the *base quality score recalibration* (BQSR) stage first categorizes each base pair from each read to different bins. Then, the BQSR algorithm counts the number of entries and the number of errors for each bin. Errors are counted when a reported base pair and the corresponding reference base pair mismatches and the particular base pair is not an SNP (i.e., the known sites of base pair variations; see Section III-B). Specifically, two binning policies are used in GATK4 BQSR. The first is to bin each read base pair by its read group (i.e., the lane of the machine that is used to process this read), the reported quality score, and the relative position of the base pair within the read (called *cycle*). The second policy is to bin each read base pair by its read group, the reported quality score, and the type of the base pair preceding this base pair, and the current base pair (called *context*). Once the number of base pairs and the number of errors in each bin are computed, the algorithm adjusts the corresponding quality score of each base pair in each read based on this statistic (i.e., a covariate table). The resulting adjusted quality scores from BQSR are known to match very well with the empirical quality scores [18].

Acceleration. For this stage, our work accelerates the binning process of BQSR (i.e., the covariate table construction stage). Our hardware accelerator takes reads and inspects each read base pair. Then, it checks if a particular base pair mismatches with the corresponding reference base pair as well as if this read base pair maps to a non-SNP site. If so, the hardware increments both the number of observations counter and the number of errors counter for the corresponding bin. If not, it only increments the number of observations counter. Once the binning finishes, the GATK4 software tool reads the constructed covariate table and adjusts the quality scores accordingly (i.e., the quality score update stage). For this stage, the reference table is pre-partitioned in software as in the *metadata update* stage. The reads table is first partitioned by its `POS` (as usual), and partitioned again by its read group. A single pipeline invocation handles a single read partition and hence multiple invocations are necessary to finish the processing of the entire data set.

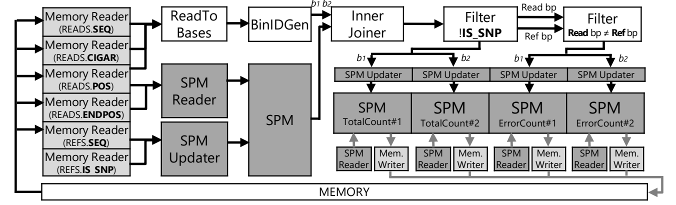

Fig. 12. BQSR (Covariate Table Construction) Hardware.

Hardware Composition. Figure 12 shows the *Genesis* hardware pipeline for the BQSR stage. This pipeline shares part of the front-end with the example hardware pipeline presented in Figure 7. However, a different set of *Genesis* hardware library modules are utilized to implement the new functionality. First, a custom module (BinIDGen) which calculates two BQSR bin IDs is added between the ReadToBases module and the Joiner module. This module takes a sequence of base pairs and their quality scores as inputs. For each base pair with quality score q, it outputs the first bin ID $b_1 = q \times$ # of *cycle* values $+$ *cycle* value [3]. It also outputs another bin ID, $b_2 = q \times$ # of *context* types $+$ *context* ID. Here, the number of context types is 16 and the context ID is assigned as follows: `AA` = 0, `AC` = 1, `AG` = 2, `AT` = 3, `CA` = 4, ..., `TT` = 15.

A more significant functionality addition compared to the pipeline presented in Figure 7 is the use of the `IS_SNP` column. For this stage, this column is stored in the SPM similar to how we store the `REF.SEQ` column. Outputs of the BinIDGen module and the reference data outputs from the SPM (columns `POS`, `SEQ`, and `IS_SNP`) are inner-joined with the Joiner which uses the `POS` from both inputs as the key. The outcome of this join is then passed to a Filter, which filters out all the data whose `IS_SNP` column is true. The resulting output of the Filter is first passed to two SPM Updaters (configured to perform read-increment-write) associated with two SPMs (TotalCountBuffer #1 and #2 in Figure 12) using input addresses b_1 and b_2 respectively. The outcome of the Filter is cascaded to another Filter, which outputs only the items whose read base pair and the reference base pair mismatches (i.e., the base pairs that are counted as empirical errors). The results are finally passed to the two SPM Updaters to update the ErrorCountBuffers #1 and #2, similar to the way the TotalCountBuffers are updated. This hardware pipeline counts the number of errors and the number of total base pairs belonging to each bin (across two binning addressing schemes) and store the results on four SPMs. Once all reads within the partition are handled, the contents of the SPMs are drained with the associated SPM Readers and then stored in memory through the Memory Writer modules.

E. Applying Genesis for other Genomic Data Processing Operations or Genome Sequencing Pipelines

The previous sections (Section IV-B, IV-C, and IV-D) explored how *Genesis* can be used to accelerate three data manipulation stages of the data preprocessing phase in the

[3] For our target data set, the # of cycle values is 302. The read length is 151 and hence the relative position in the read ranges from 0 to 150, but additional cycle values are assigned for its *reverse* read as well.

GATK4 pipeline. However, this is not the only use of *Genesis*. *Genesis* can also be used to accelerate genomic data manipulation operations in other phases or pipelines. Our preliminary analysis indicates that *Genesis* can be used for other portions of the GATK4 genome sequencing pipeline such as FM-index based seeding in the BWA-MEM aligner, active region determination in the HaplotypeCaller, joint genotyping, and intersection of training/truth resource sets and callsets in Variant Quality Score Recalibration (VQSR), which essentially consist of data consolidation, filtering, and matching operations. Furthermore, *Genesis* framework can be used to accelerate the post-sequencing analysis (e.g., checking whether a genome sequence is susceptible to specific diseases), which are often performed using relational database queries (e.g., GenAp [32], Gemini [49]).

Genesis-generated accelerators are also applicable to variants of the GATK4 Best Practices pipelines or other independent genome sequencing pipelines. These data manipulation operations are independent of the choice of alignment algorithms (e.g., Minimap2 [35], DRAGEN aligner [20]) or variant calling algorithms (e.g., Strelka2 [30], FreeBayes [22], DeepVariant [51]), and thus applicable to many variants of the GATK4 Best Practices pipelines. Moreover, many of the data manipulation operations are common across different genome sequencing pipelines. For example, pipelines such as DRAGEN [20], Sentieon [29], or Berkeley ADAM [43], perform similar tasks with slightly different algorithms for specific stages (e.g., alignment, variant calling). Thus all or most of *Genesis*-generated accelerators presented in this work are applicable to such pipelines as well.

V. Evaluation

A. Methodology

TABLE II
HARDWARE CONFIGURATIONS FOR THE AWS EC2 F1 AND R5 INSTANCE

Machine Configurations		
AWS Instance	**System Components**	
f1.2xlarge (for *Genesis* HW)	Host Processors	Intel Xeon E5-2686 v4 (Broadwell) 4C/8T, 2.3 GHz (Turbo 3 GHz)
	Host Memory	122 GiB
	Host Storage	500 GB SSD
	FPGA	1x Xilinx Virtex UltraScale+ VU9P 2.5 M logic elements, 6,800 DSPs
	FPGA Memory	64 GB
	Cost (2019.11)	$1.65/hr
r5.4xlarge (for GATK4 SW)	Processors	Intel Xeon Platinum 8175M (Skylake-SP) 8C/16T, 2.5GHz (3.5 GHz Turbo Boost)
	Memory	128 GiB
	Storage	2TB SSD
	Cost (2019.11)	$1.01/hr (Compute), $0.28/hr (Storage)

To demonstrate *Genesis*'s capability to accelerate data manipulation operations in genomic data analysis, we implement hardware accelerators (we call each hardware pipeline(s) constructed for a particular algorithm an *accelerator* in the rest of this paper to avoid confusion) for three key data manipulation operations in the data preprocessing phase of GATK4 using the *Genesis* framework, and deploy them on the commercial cloud using the Amazon EC2 f1.2xlarge instances (Table II). Each F1 instance contains a Xilinx Virtex UltraScale+ VU9P FPGA card [3]. We use a 250MHz clock for all three accelerators. We configure the number of pipelines to i) the resource limit we can fit on one FPGA card or ii) the performance limit where an accelerator can no longer get more speedup from parallelism due to memory or communication bottlenecks. We used 16× pipelines for *mark duplicates*, 16× pipelines for *metadata update*, and 8× pipelines for *base quality score recalibration*.

To compare our design with the software-only implementation, we run GATK version 4.1.3 [9] on an AWS r5.4xlarge instance (Table II) that is memory-optimized. A large memory is crucial to obtain high performance for genomic data analysis workloads. In addition, we add a 2TB SSD volume to this machine so that the software can run with fast SSDs. For the reads input data set, we use a well-characterized Illumina sequencing result of patient NA12878 obtained from the Broad Institute Public Dataset [8]. We use `GRCh38` reference genome [55] and `dbSNP138` [41] SNP sites data set to construct a reference table.

B. Evaluation Results

Performance. Figure 13(a) shows the speedup of three *Genesis* accelerators designed to accelerate various stages of the GATK4 data preprocessing phase, over the GATK4 software implementations run on a carefully configured 8-core AWS EC2 R5 instance. For the *mark duplicates* stage, a single speedup number is shown since the GATK4 pipeline does not divide the data by chromosome until the sorting is completed, which happens at the end of the *mark duplicates* stage. For *metadata update*[4] and *base quality score recalibration* stages, both per-chromosome speedups and overall speedups are presented (Figure 13(c) and (d)). *Genesis* achieves an overall speedup of 2× on *mark duplicates* stage, 19.3× on *metadata update*, and 12.6× on BQSR (covariate table construction). Considering that these three stages take about three and a half hours for a single genome to execute (assuming that *metadata update* perfectly scales), *Genesis* reduces the computation time to process a single person's gene by roughly 140 minutes.

Runtime Breakdown. Figure 13(b) shows the breakdown of the *Genesis* framework runtime for the three stages in the GATK4 data preprocessing phase. Figure 13(a) shows that *mark duplicates* achieves a lower speedup (approximately 2×) than others since the un-accelerated software portion of the stage (takes 99.35% of the runtime) works as a bottleneck once the hardware achieves significant speedup. *Metadata update* and *BQSR* speedups are partially limited by the host-FPGA communication (takes 53.4% and 29.5% of the runtime repectively as shown in Figure 13(b)) interface bandwidth. On an AWS F1 instance, the host communicates to and from the FPGA via a PCIe DMA interface, which is measured at approximately 7 GB/s on our custom microbenchmark. Considering that the next generation communication interfaces

[4]For *metadata update* stage, since the baseline implementation is single-threaded, we conservatively assume that the baseline software implementation perfectly scales for 8 cores.

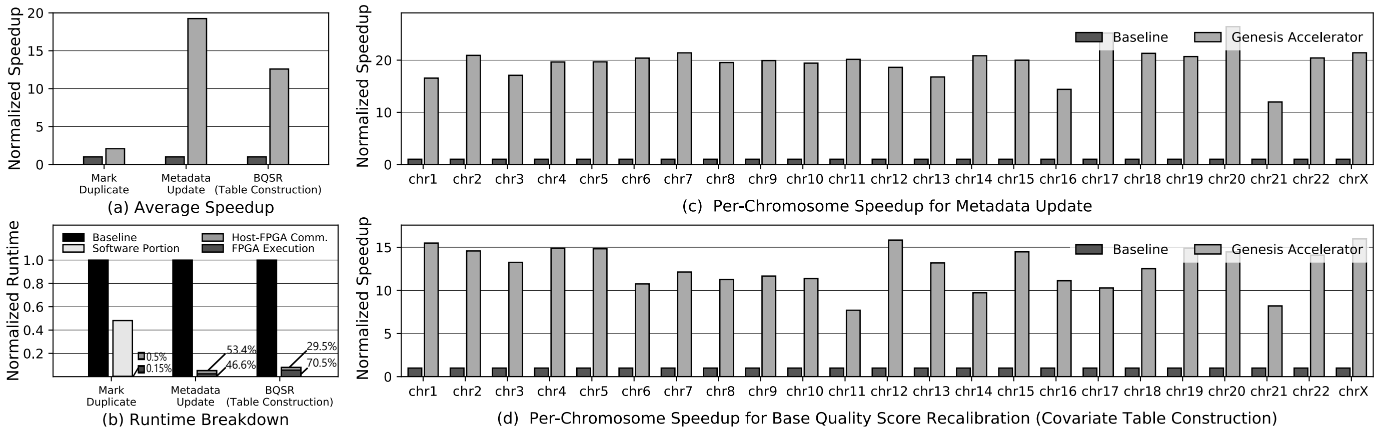

Fig. 13. Performance/Cost comparison of the *Genesis* accelerators over baseline for three GATK4 data preprocessing stages.

such as PCIe 4.0 [50] or CXL [17] will provide much higher bandwidths and the DMA controller performance will improve accordingly, the presented speedups for *Metadata update* and *BQSR* can improve significantly (e.g., 33× and 16.4× respectively when 32 GB/s PCIe 4.0 interface is assumed) with such technologies.

The speedups from *Genesis* may seem lower than that of other ASIC domain-specific accelerators targeting a very specific, mostly compute-intensive algorithm and achieving several orders of magnitude speedups. Unlike such accelerators, our accelerators target data manipulation operations, whose performance is often bottlenecked by limitations in the host and FPGA communication interface, memory, or storage systems. Furthermore, the relatively lower speedup on data manipulation operation does not mean that this is less important. In fact, it is quite the opposite since the importance of accelerating data manipulation operations becomes greater when other compute-intensive stages of the pipeline are significantly accelerated, as shown in Section IV-A. More importantly, *Genesis* provides this speedup with deployability and flexibility. While an FPGA has significant drawbacks over ASICs in terms of performance, energy consumption, and area, it has a notable advantage in that it is much easier to deploy and adapt. *Genesis* adds further programmability and productivity with its SQL-oriented software interface and a composable accelerator construction approach utilizing the *Genesis* hardware library.

Cost. Many genomic data processing workloads exhibit a plethora of parallelism and thus often scale relatively well with the increased amount of resources. In such a scenario, the cost can be a more meaningful metric than the raw speedup itself since it considers the amount of resources the system utilizes. Table III compares the cost of running each accelerated stage in the AWS cloud with the configurations in Table II. *Genesis* reduces the cost of genomic data processing by up to 15× and achieves up to 290× better efficiency measured in performance per dollar compared to the software-only baseline.

FPGA Resource Usage. Table IV shows the FPGA resource consumption of the *Genesis* accelerators. *Genesis* accelerators tend to under-utilize the FPGA resources since most of the algorithm is communication/memory-bound once the number of pipelines in a system exceeds a certain threshold. This implies that further improvement on the FPGA memory interface, such as the adoption of HBM2 in recent Xilinx UltraScale+ FPGAs [64], can lead to further speedup. It is also possible to exploit the under-utilized configuration and place multiple *Genesis* accelerators targeting different operations in a single FPGA so that users can time-multiplex the accelerators and avoid reprogramming.

TABLE III
COST COMPARISON OF *Genesis* AND BASELINE SYSTEMS.

Stage	*Genesis* Cost Reduction	*Genesis* Speedup	Normalized Performance/$
Mark Duplicates	2.08×	2.08×	4.31×
Metadata Update	15.05×	19.25×	289.59×
BQSR (Table Construction)	9.84×	12.59×	123.92×

TABLE IV
FPGA RESOURCE USAGE OF *Genesis*.

Type	Used	Available	Utilization(%)
Mark Duplicates			
CLB Lookup Tables	228K	895K	25.4%
CLB Registers	272K	1790K	15.2%
BRAMs	0.34MB	7.56MB	4.55%
Metadata Update			
CLB Lookup Tables	333K	895K	37.19%
CLB Registers	424K	1790K	23.7%
BRAMs	4.95MB	7.56MB	65.5%
Base Quality Score Recalibration			
CLB Lookup Tables	502K	895K	56.1%
CLB Registers	257K	1790K	14.4%
BRAMs	1.69MB	7.56MB	22.4%

VI. RELATED WORK

Hardware Accelerators for Genomics. There is substantial prior work focused on accelerating genome sequencing, signifying its importance. However, most prior work attempts to accelerate a very specific algorithm, rather than common generic operations. For example, many accelerators target different types of alignment algorithms [12], [13], [21], [23], [25], [26], [36], [58], pre-alignment filtering [2], variant calling algorithms [6], [26], [38], or the INDEL alignment algorithm [62]. These works achieve exceptional efficiency with an architecture highly specialized to a specific algorithm,

which contrasts with our approach. We leverage higher-level primitives (SQL and SQL extensions) to map a wider swath of data manipulation operations in genomic data processing into hardware. Further, once specific algorithms in the genomics domain are accelerated, the time spent on the generic data manipulation operations becomes much larger, which *Genesis* can successfully accelerate.

Storing or Processing Genomics using DBMSs. Many existing frameworks, such as Gemini [49], GenAP [32], and many others [31], [39], [52], [53], [61], conceptualize genomic data as a database and use a DBMS and SQL-style query languages to process data in a fast and efficient way in software. Our work adopts the SQL query as a software interface and utilizes specialized hardware to accelerate such database operations. The popularity of database usage in genomic data processing shows that *Genesis*'s front-end interface can be effectively utilized to bridge the gap between the bioinformaticians and the hardware designers.

Hardware Accelerators for Data-intensive Applications. Our work is most closely related to LINQits [15], Q100 [63], and SDA [48] in that those three works attempt to accelerate database operations with the use of ASICs or FPGAs. For this reason, our hardware library has partial overlap with some of the hardware modules proposed in those frameworks. However, in practice, it is often challenging, if not impossible, to utilize such prior works to accelerate genomics data manipulation operations efficiently. For example, LINQits or SDA programming model assumes a specific processing pattern, which is too naïve to efficiently support the complicated dataflow patterns shown in Figure 11 and 12. To support a complex processing pattern, such approaches need to decompose a complex operation into multiple smaller operations and use the main memory for communication between operations, which is extremely inefficient. Similarly, Q100 only utilizes scratchpad memory as a stream buffer and thus cannot implement the dataflow pipeline exploiting data reuse. In addition, such accelerators specialize on a relatively small subset of the SQL operations (i.e., select, sort, join, groupby), which are not sufficient to represent genomics data manipulation operations that require other SQL operations (such as PosExplode) or genomic data-specific operation (such as ReadExplode). Mondrian data engine [19] also aims to accelerate database operations exploiting the specific characteristics of a near-memory processing architecture. Finally, *Genesis* draws inspirations from accelerators focusing on efficient streaming data movements such as DaNa [37], Imagine [1], [28], RSVP [16], Softbrain [44], Graphicionado [24], and CoRAM++ [60], but exploits different abstractions and targets completely different domains.

Proprietary Commercial Solutions. Illumina DRAGEN [20] is an FPGA-based proprietary solution that accelerates the genomic secondary analysis. Its pipeline is different from GATK4; however, they recently started a collaboration with Broad Institute and planned to release the co-developed GATK-compatible pipeline in the second half of 2020 [10]. In addition, NVIDIA now provides the proprietary GPU-acceleration solution for the GATK4-compatible pipeline [45] after its acquisition of Parabricks in December 2019. DRAGEN claims a 30× end-to-end speedup over traditional CPU solution [20] and NVIDIA Parabricks [45] claims a 48× end-to-end speedup over a single 32-core CPU with 8 V100 GPUs. A direct comparison with our proposal is difficult since DRAGEN utilizes a slightly different pipeline and does not report the exact experimental setup. Furthermore, most of DRAGEN's and Parabricks's end-to-end speedup comes from large, computation-oriented stages such as alignment or variant calling stages while *Genesis* focuses on data manipulation stages. These industrial solutions can provide high performance for the existing stages in the pipeline, but their micro-architectures (DRAGEN) or GPU optimization strategies (Parabricks) are not open to the public. In addition, *Genesis*'s approach allows highly productive accelerator development of new or modified algorithms thanks to its composability.

VII. Conclusion

We propose *Genesis*, a flexible acceleration framework for genomic data analysis. *Genesis* consists of a software interface using SQL-style queries for data manipulation operations, a genomic data processing hardware library, as well as an accelerator management API. We demonstrate the deployability, flexibility, and composability of *Genesis* by implementing multiple GATK4 preprocessing stages. *Genesis* achieves notably higher performance and cost efficiency than GATK4 software running on a commodity Xeon CPU server. In addition, *Genesis* allows bioinformaticians to expedite their research analysis by leveraging domain-specific hardware platforms, that are generally difficult to use, by providing an easy-to-use programming interface based on an already widely-adopted data analysis language, SQL.

Acknowledgment

We thank the reviewers for their insightful comments. This work was supported in part by a National Research Foundation of Korea research grant funded by the Ministry of Science and ICT (PE Class Heterogeneous High Performance Computer Development, NRF-2016M3C4A7952587). This work was also funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy (under Award Number DE-AR0000849), ADEPT Lab industrial sponsor Intel, RISE Lab sponsor Amazon Web Services, and ADEPT Lab affiliates Google, Siemens, and SK Hynix. The views and opinions expressed here are solely those of the authors and do not necessarily state or reflect those of the government or any of the sponsors.

References

[1] J. H. Ahn, W. J. Dally, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das, "Evaluating the imagine stream architecture," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[2] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping," *Bioinformatics*, vol. 33, no. 21, pp. 3355–3363, 2017.

[3] "Amazon EC2 F1 Instances," https://aws.amazon.com/ec2/instance-types/f1, Amazon.

[4] J. Andrews, “23andMe competitor Veritas Genetics slashes price of whole genome sequencing 40% to $600,” https://www.cnbc.com/.

[5] “Spark SQL,” https://spark.apache.org/sql/, Apache Software Foundation.

[6] S. S. Banerjee, M. el-Hadedy, C. Y. Tan, Z. T. Kalbarczyk, S. Lumetta, and R. K. Iyer, “On accelerating pair-HMM computations in programmable hardware,” in *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[7] “Broad Institute About Us: This is Broad,” https://www.broadinstitute.org/about-us, Broad Institute.

[8] “Broad Institute Public Datasets Google Cloud Repository,” https://console.cloud.google.com/storage/browser/broad-public-datasets?project=broad-public-datasets&organizationId=548622027621, Broad Institute.

[9] “GATK4 data preprocessing,” https://github.com/gatk-workflows/gatk4-data-processing, Broad Institute.

[10] “Illumina and broad institute announce agreement to co-develop genomic secondary analysis tools,” https://www.broadinstitute.org/news/illumina-and-broad-institute-announce-agreement-co-develop-genomic-secondary-analysis-tools, Broad Institute.

[11] Broad Institute GATK Dev Team, “Introduction to the GATK best practices,” https://software.broadinstitute.org/gatk/best-practices/.

[12] M. F. Chang, Y. Chen, J. Cong, P. Huang, C. Kuo, and C. H. Yu, “The smem seeding acceleration for dna sequence alignment,” in *Proceedings of the 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[13] Y.-T. Chen, J. Cong, J. Lei, and P. Wei, “A novel high-throughput acceleration engine for read alignment,” in *Proceedings of the 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2015.

[14] “Chisel hardware construction language,” https://chisel.eecs.berkeley.edu/.

[15] E. S. Chung, J. D. Davis, and J. Lee, “Linqits: Big data on little clients,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[16] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, “The reconfigurable streaming vector processor (RSVP),” in *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO)*, 2003.

[17] “Compute Express Link,” https://www.computeexpresslink.org/, CXL Consortium.

[18] M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. del Angel, M. A. Rivas, M. Hanna, A. McKenna, T. J. Fennell, A. M. Kernytsky, A. Y. Sivachenko, K. Cibulskis, S. B. Gabriel, D. Altshuler, and M. J. Daly, “A framework for variation discovery and genotyping using next-generation dna sequencing data,” *Nature Genetics*, vol. 43, 2011.

[19] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Penvmatikatos, “The mondrian data engine,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[20] “Dragen bio-it processor,” http://www.edicogenome.com/dragen_bioit_platform/, Edico Genome.

[21] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, “GenAX: a genome sequencing accelerator,” in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018.

[22] E. Garrison and G. T. Marth, “Haplotype-based variant detection from short-read sequencing,” *ArXiv e-print arXiv:1207.3907*, 2012.

[23] L. Guo, J. Lau, Z. Ruan, P. Wei, and J. Cong, “Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu,” in *Proceedings of the 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[24] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, “Graphicionado: A high-performance and energy-efficient accelerator for graph analytics,” in *Proceedings of the 49th Annual International Symposium on Microarchitecture (MICRO)*, 2016.

[25] B. Harris, A. C. Jacob, J. M. Lancaster, J. Buhler, and R. D. Chamberlain, “A banded Smith-Waterman FPGA accelerator for Mercury BLASTP,” in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2007.

[26] S. Huang, G. J. Manikandan, A. Ramachandran, K. Rupnow, W. H. Wenmei, and D. Chen, “Hardware acceleration of the Pair-HMM algorithm for DNA variant calling,” in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[27] “Illumina,” https://www.illumina.com/, Illumina.

[28] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, “The imagine stream processor,” in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, 2002.

[29] K. I. Kendig, S. Baheti, M. A. Bockol, T. M. Drucker, S. N. Hart, J. R. Heldenbrand, M. Hernaez, M. E. Hudson, M. T. Kalmbach, E. W. Klee, N. R. Mattson, C. A. Ross, M. Taschuk, E. D. Wieben, M. Wiepert, D. E. Wildman, and L. S. Mainzer, “Sentieon dnaseq variant calling workflow demonstrates strong computational performance and accuracy,” *Frontiers in Genetics*, vol. 10, p. 736, 2019.

[30] S. Kim, K. Scheffler, A. L. Halpern, M. A. Bekritsky, E. Noh, M. Källberg, X. Chen, Y. Kim, D. Beyter, P. Krusche, and C. T. Saunders, “Strelka2: fast and accurate calling of germline and somatic variants,” *Nature Methods*, vol. 15, no. 8, pp. 591–594, 2018.

[31] C. Kozanitis, A. Heiberg, G. Varghese, and V. Bafna, “Using Genome Query Language to uncover genetic variation,” *Bioinformatics*, vol. 30, no. 1, pp. 1–8, 2013.

[32] C. Kozanitis and D. A. Patterson, “Genap: a distributed sql interface for genomic data,” *BMC bioinformatics*, vol. 17, pp. 63–63, 2016.

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

[34] M. Lek, K. J. Karczewski, E. V. Minikel, K. E. Samocha, E. Banks, T. Fennell, A. H. O’Donnell-Luria, J. S. Ware, A. J. Hill, B. B. Cummings *et al.*, “Analysis of protein-coding genetic variation in 60,706 humans,” *Nature*, vol. 536, no. 7616, pp. 285–291, 2016.

[35] H. Li, “Minimap2: pairwise alignment for nucleotide sequences,” *Bioinformatics*, vol. 34, no. 18, 2018.

[36] I. T. Li, W. Shum, and K. Truong, “160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA),” *BMC bioinformatics*, vol. 8, no. 1, p. 185, 2007.

[37] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh, “In-rdbms hardware acceleration of advanced analytics,” *Proceedings of the VLDB Endowment*, vol. 11, no. 11, p. 1317–1331, 2018.

[38] G. J. Manikandan, S. Huang, K. Rupnow, W. W. Hwu, and D. Chen, “Acceleration of the Pair-HMM algorithm for DNA variant calling,” in *Proceedings of the 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[39] M. Masseroli, P. Pinoli, F. Venco, A. Kaitoua, V. Jalili, F. Palluzzi, H. Muller, and S. Ceri, “GenoMetric Query Language: a novel approach to large-scale genomic data management,” *Bioinformatics*, vol. 31, no. 12, pp. 1881–1888, 2015.

[40] P. Muir, S. Li, S. Lou, D. Wang, D. J. Spakowicz, L. Salichos, J. Zhang, G. M. Weinstock, F. Isaacs, J. Rozowsky *et al.*, “The real cost of sequencing: Scaling computation to keep pace with data generation,” *Genome biology*, vol. 17, no. 1, p. 53, 2016.

[41] “Human genome resources at NCBI,” https://www.ncbi.nlm.nih.gov/genome/guide/human/, National Center for Biotechnology Information.

[42] “DNA sequencing costs,” http://www.genome.gov/sequencingcosts, National Human Genome Research Institute.

[43] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, M. Franklin, A. D. Joseph, and D. A. Patterson, “Rethinking data-intensive science using scalable analytics systems,” in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2015.

[44] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, “Stream-dataflow acceleration,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[45] “NVIDIA PARABRICKS,” https://developer.nvidia.com/nvidia-parabricks, NVIDIA Corporation.

[46] “PL/SQL for developers,” https://www.oracle.com/database/technologies/appdev/plsql.html, Oracle.

[47] “Database SQL Tuning Guide - SQL Processing,” https://docs.oracle.com/database/121/TGSQL/tgsql_interp.htm#TGSQL94618, Oracle, 2020.

[48] J. Ouyang, W. Qu, Y. Wang, Y. Tu, J. Wang, and B. Jia, “SDA: Software-defined accelerator for general-purpose big data analysis system,” in *Hot Chips: A Symposium on High Performance Chips*, 2016.

[49] U. Paila, B. A. Chapman, R. Kirchner, and A. R. Quinlan, “Gemini: Integrative exploration of genetic variation and genome annotations,” *PLOS Computational Biology*, vol. 9, 2013.

[50] “PCI Express Specifications,” https://pcisig.com/specifications, PCI SIG.

[51] R. Poplin, P.-C. Chang, D. Alexander, S. Schwartz, T. Colthurst, A. Ku, D. Newburger, J. Dijamco, N. Nguyen, P. T. Afshar, S. S. Gross, L. Dorfman, C. Y. McLean, and M. A. DePristo, "A universal snp and small-indel variant caller using deep neural networks," *Nature Biotechnology*, vol. 36, 2018.

[52] U. Röhm and J. A. Barkeley, "Data management for high-throughput genomics," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2009.

[53] M.-P. Schapranow and H. Plattner, "HIG – an in-memory database platform enabling real-time analyses of genome data," in *Proceedings of the IEEE International Conference on Big Data (IEEE Big Data)*, 2013.

[54] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: Astronomical or genomical?" *Public Library of Science (PLoS) Biology*, vol. 13, no. 7, p. e1002195, 2015.

[55] "Genome reference consortium human build 38," https://www.ncbi.nlm.nih.gov/grc, The Genome Reference Consortium.

[56] The White House Office of the Press Secretary, "Fact sheet: President Obama's Precision Medicine Initiative," https://obamawhitehouse.archives.gov/the-press-office/2015/01/30/fact-sheet-president-obama-s-precision-medicine\\-initiative.

[57] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[58] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[59] Y. Turakhia, S. D. Goenka, G. Bejerano, and W. J. Dally, "Darwin-wga: A co-processor provides increased sensitivity in whole genome alignments with high speedup," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2019.

[60] G. Weisz and J. C. Hoe, "CoRAM++: Supporting data-structure-specific memory interfaces for FPGA computing," in *Proceedings of the 25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015.

[61] M. Wiewiórka, A. Leśniewska, A. Szmurło, K. Stępień, M. Borowiak, M. Okoniewski, and T. Gambin, "SeQuiLa: an elastic, fast and scalable SQL-oriented solution for processing and querying genomic intervals," *Bioinformatics*, vol. 35, no. 12, pp. 2156–2158, 2018.

[62] 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, and A. D. Joseph, "FPGA accelerated INDEL realignment in the cloud," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2019.

[63] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: the architecture and design of a database processing unit," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[64] "Supercharge Your AI and Database Applications with Xilinx's HBM-Enabled UltraScale+ Devices Featuring Samsung HBM2," https://www.xilinx.com/support/documentation/white_papers/wp508-hbm2.pdf, Xilinx, 2019.

[65] C. K. Yung, G. Bourque, P. C. Boutros, K. El Emam, V. Ferretti, B. M. Knoppers, B. O'Connor, B. F. Ouellette, C. Sahinalp, S. P. Shah *et al.*, "ICGC in the cloud," 2016.