Accelerating Genomic Data Analytics With Composable Hardware Acceleration Framework

Tae Jun Ham, *Seoul National University, Seoul, 130-743, South Korea*
David Bruns-Smith and Brendan Sweeney, *University of California Berkeley, Berkeley, CA, 94720-5800, USA*
Yejin Lee, Seong Hoon Seo, and U Gyeong Song, *Seoul National University, Seoul, 130-743, South Korea*
Young H. Oh, *Sungkyunkwan University, Seoul, 03063, South Korea*
Krste Asanovic, *University of California Berkeley, Berkeley, CA, 94720-5800, USA*
Jae W. Lee, *Seoul National University, Seoul, 130-743, South Korea*
Lisa Wu Wills, *Duke University, Durham, NC, 27708-0187, USA*

This article presents a framework, Genesis (genome analysis), to efficiently and flexibly accelerate generic data manipulation operations that have become performance bottlenecks in the genomic data processing pipeline utilizing FPGAs-as-a-service. Genesis conceptualizes genomic data as a very large relational database and uses extended SQL as a domain-specific language to construct data manipulation queries. To accelerate the queries, we designed a Genesis hardware library of efficient coarse-grained primitives that can be composed into a specialized dataflow architecture. This approach explores a systematic and scalable methodology to expedite domain-specific end-to-end accelerated system development and deployment.

As the democratization of wet lab sequencing technology drives down sequencing cost, the cost and runtime of data analysis are becoming more significant. 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). 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 parti`cular, identifying genomic variants in each individual genome, as it is one of the most computationally demanding pipelines.

0272-1732 © 2021 IEEE
Digital Object Identifier 10.1109/MM.2021.3072385
Date of publication 12 April 2021; date of current version 25 May 2021.

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 Genome Analysis ToolKit 4 (GATK4) Best Practices, 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 such as read alignment[1–3] or pair-HMM (hidden Markov model) in variant calling. Such specialized accelerators, targeting a specific implementation of a particular genome sequencing pipeline stage, have demonstrated orders of magnitude speedups and energy-efficiency improvements. With these specific algorithm accelerations in place, the remaining unaccelerated analysis stages that contain 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.[3] However, GATK4 does not utilize this stage with its updated variant calling algorithms rendering the proposal suitable largely 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. 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.

WE ARCHITECT AND EVALUATE A FLEXIBLE ACCELERATION FRAMEWORK THAT TARGETS GENERIC DATA MANIPULATION OPERATIONS COMMONLY USED IN GENOMIC DATA PROCESSING CALLED GENESIS.

Thus, we architect and evaluate a flexible acceleration framework that targets generic data manipulation operations commonly used in genomic data processing called *Genesis*. We view 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 transforms genomic data processing stages into simple extended SQL-style queries. Once the queries 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. As a proof of concept, we accelerate the data preprocessing phase in GATK4 Best Practices and deploy *Genesis*-generated accelerators on Amazon EC2 F1 instances. We demonstrate that the accelerated system targeting these queries provides a significant performance improvement and cost savings over a commodity CPU.

GENOMIC DATA ANALYTICS

A genome is an organism's complete set of DNAs. For a human genome, each chromosome is represented as a sequence of DNA base pairs expressed as a single character, `A`, `T`, `C`, `G`, representing a DNA nucleotide base. Genomic analysis uses a DNA sequence to identify variations from a biological sample 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 an 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 postmeasurement analysis is called the *primary analysis*, and the outcome of the primary analysis is an input to the *secondary analysis*. *Secondary 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).

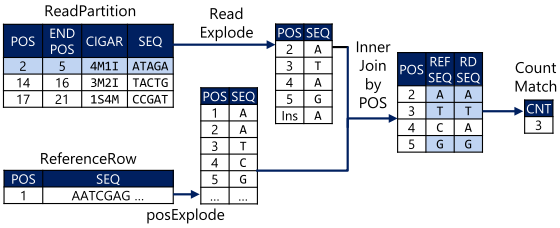
GENESIS ACCELERATION FRAMEWORK

Representing Genomic Data Analysis as Relational Database Queries

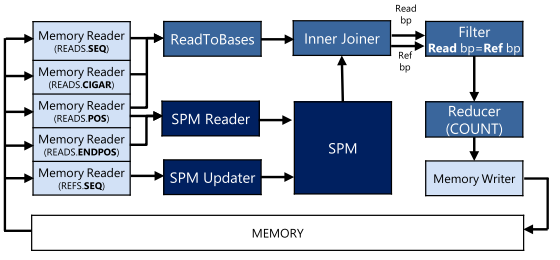
Genesis conceptualizes genomic data as relational data tables and uses SQL as a domain-specific language to represent the target genomic analysis operations and pipeline stages for acceleration. Genomic reads are represented as rows in a table, and attributes associated with each read are represented as columns. We use Illumina sequencer short reads (up to 151 base pairs per read) for a specific human in our evaluated dataset. 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 1 M base pairs. For the efficient management of those tables, we partition each table into multiple tables by chromosome identifiers, and then again by the mapped position of the reads or the reference data. For both tables, we assign a unique partition ID to the partition. *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

```
/* I1: Extract Reads and Reference Partition P */
  .. Omitted for the conciseness ..
/* I2: posExplode on ReferenceRow */
CREATE TABLE RelevantReference AS
PosExplode (ReferenceRow.SEQ, ReferenceRow.POS)
FROM ReferenceRow
/* Iterate over Rows */
FOR SingleRead IN ReadPartition:
  /* 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, ReadLen)
  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;
```

(a)



(b)



(c)

FIGURE 1. Example query, its execution flow, and the genesis-generated HW pipeline.

for every row that is exploded). `ReadExplode` converts a read, stored as a single row in the read tables, to multiple, separate rows where each row contains the base, the corresponding quality score, and its position. This operation converts individual base pairs and corresponding quality scores into separate rows utilizing its alignment information recorded in the metadata called CIGAR (CIGAR contains base pair alignment information such as substitution, matching, insertion, deletion, etc.). Finally, we support iteration over rows with the `FOR Row IN Table` clause, which is similar to that of Oracle PL/SQL.

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. In this example, the user wants 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 1(a), which essentially follows the execution flow depicted in Figure 1(b). **Step 1**: the set of reads and the relevant reference with the partition ID (`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 multirow 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`). **Genesis Hardware Library**: *Genesis* framework lets a user easily construct a dataflow pipeline that accelerates the desired target query [e.g., Figure 1(a)]. The key idea behind this framework is that a relational query can be decomposed into a series of relational operators. For example, it is well known that SQL queries can be easily parsed into a tree graph where each node represents a table (leaf node) or a relational/computational operator (nonleaf node).[4] In such a case, if there exists a set of configurable hardware modules where each of them can be directly mapped to each relational/computational operator, constructing a dataflow pipeline for the query becomes 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.

Each Genesis hardware library 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

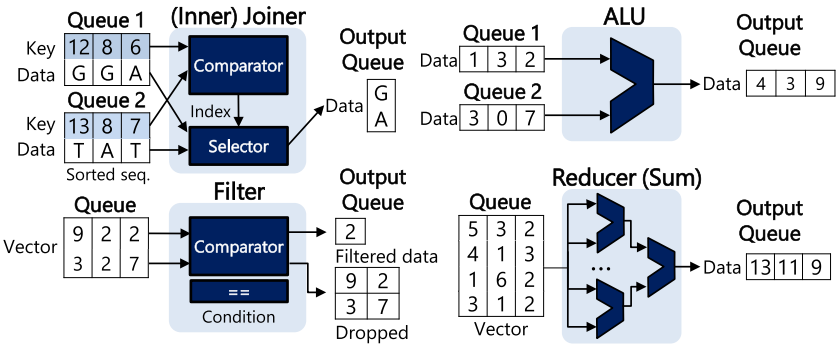


FIGURE 2. Block diagrams of genesis data manipulation and computation modules.

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 into the output queue, which will work as an input queue for the next module.

(1) Data Manipulation and Computation Modules

Figure 2 shows the block diagrams for four data manipulation and computation modules in Genesis hardware library. The figure visualizes the operations of each module with the example input/output values. Detailed explanations for each module are provided below.

Joiner merges flits from two input queues and produces a single output. For this module, a flit in an input queue should consist of a key field and a data field. 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.

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 takes a sequence of data and performs a reduction operation (e.g., Sum, Max, Min, Count) with a reduction tree. For this module, a reduction tree is utilized to obtain a 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).

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. When a single item contains multiple values, the unary/binary operation is performed in an elementwise manner.

(2) Memory Access Modules

Memory Reader reads contiguous data from memory and streams 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.

Memory Writer writes the data coming from an input queue to memory. It takes a single flit from the previous module per cycle and temporarily stores it in 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 preconfigured starting address.

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 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 configured to perform a random SPM write, which simply writes the value to the provided address. Finally, it can be configured to perform a read-modify-write update with the provided modify function (e.g., add/subtract a constant).

(3) Genomic Data Processing Modules

ReadToBases supports the `ReadExplode` operation. 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. The reference position field can be Ins if the base is an inserted base. Similarly, the base and quality score fields can be Del if the base is deleted.

Constructing Hardware Accelerator with Genesis Hardware Library

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 1(a) is translated to the hardware pipeline shown in Figure 1(c). The hardware pipeline 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 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. Unlike the reads data, the relevant reference data are mapped to an on-chip SPM to facilitate data reuse. A single memory reader 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. Finally, 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 one base pair per cycle. 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. *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. 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. A set of stock *Genesis* modules are often enough to design accelerators for data manipulation operations in the existing gene processing pipeline. However, different genomics data often need different treatment, and thus *Genesis* is designed to support user-defined modules. *Genesis* provides a standardized stream-based I/O interface for all of its modules; a user only has to specify the user-defined internal computation hardware in Chisel to utilize the framework at ease.

EVALUATION

Methodology

Figure 3(a) shows the runtime breakdown of the GATK4 Best Practices data preprocessing pipeline with (bottom) and without (top) recently developed hardware alignment accelerator.[2] To demonstrate *Genesis*'s capability to accelerate data manipulation operations in genomic data analysis, we architected and implemented hardware accelerators (we call each hardware pipeline(s) constructed for a particular algorithm an *accelerator* for the rest of this article to avoid confusion) for three key data manipulation operations,[5] namely Mark Duplicates, Metadata Update, and the table construction phase of BQSR, which together account for the majority of the runtime in the data preprocessing phase of GATK4 Best Practices. Figure 3(b) shows the block diagrams for these *Genesis*-generated accelerators.

We deployed *Genesis*-generated accelerators on the commercial cloud using the Amazon EC2 f1.2xlarge instances. Each F1 instance contains a Xilinx Virtex UltraScale+ VU9P FPGA card. We use a 250 MHz clock for all three accelerators. We configure the number of pipelines to 1) the resource limit we can fit on one FPGA card or 2) 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 on an Amazon EC2 r5.4xlarge instance that is memory-optimized. Large memory is crucial to obtain high performance for genomic data analysis workloads. For the reads input dataset, we use a well-characterized

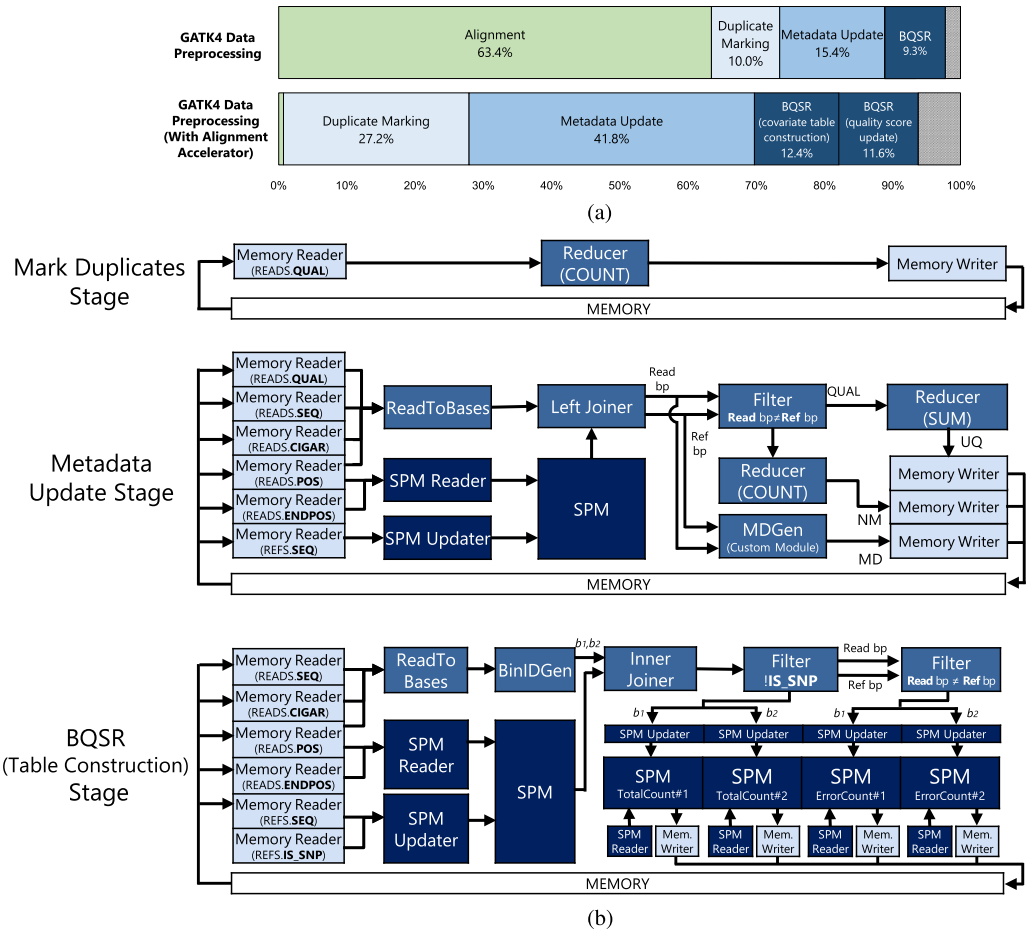


FIGURE 3. GATK4 best practices runtime breakdown and block diagrams for the genesis-generated accelerators.

Illumina sequencing result of patient NA12878 obtained from the Broad Institute Public Dataset, and we use GRCh38 as the reference genome.

Performance Results

Figure 4(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 memory-optimized CPU instance. *Genesis* achieves an overall speedup of 2× on *mark duplicates* stage, 19.3× on *metadata update*, and 12.6× on BQSR (covariate table construction). For *metadata update* and *BQSR* stages, per-chromosome speedups are also presented in Figure 4(c) and (d). 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 min. Figure 4(b) shows the breakdown of the *Genesis* framework runtime for the three stages in the GATK4 data preprocessing phase. The figure shows that the relatively low speedup of *mark duplicates* stage is due to the unaccelerated software portion of the stage, which is responsible for about 50% of the baseline runtime. Furthermore, the figure indicates the *metadata update* and *BQSR* speedups are partially limited by the host-FPGA communication (takes 53.4% and 29.5% of the runtime).

Cost Comparison

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. We compare the cost of running each accelerated stage in the AWS f1 instance (1.69$/hr) with the cost of running baseline software implementations on r5.4xlarge

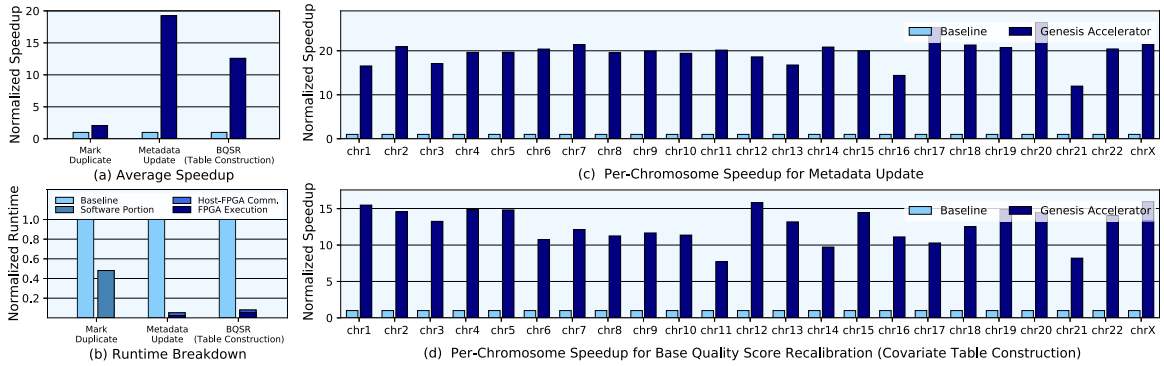


FIGURE 4. Performance comparison of the *Genesis* accelerators over baseline for three GATK4 data preprocessing stages.

instance (1.29$/hr). Compared to the baseline, *Genesis* reduces the cost of genomic data processing by 2.08×, 15.05×, and 9.84× for Mark Duplicates, Metadata Update, and BQSR (table construction) stages.

CONCLUSION AND IMPLICATIONS

Genesis acceleration framework explores a systematic and scalable methodology to democratize end-to-end accelerated system development and deployment using a software interface that is a standardized language as a domain-specific language and a hardware library that is composed of efficient coarse-grained primitives. While the work specifically showcases a set of de facto algorithms in genomic analytics, the concepts articulated and demonstrated in this work can be applied to many other domains and inspire accelerated systems research that offers a degree of generality without sacrificing the efficiency brought upon via specialization.

GENESIS HARDWARE LIBRARY DEMONSTRATES THAT CONSTRUCTING HARDWARE ACCELERATORS BY UTILIZING A SET OF COMPOSABLE HARDWARE MODULES ENABLES A FLEXIBLE HARDWARE ACCELERATOR DESIGN THAT CAN BE EASILY EXTENDED OR UPDATED.

In this article, we investigate commonalities between database and genomic analytics domains by conceptualizing genomic data as a very large relational database and mapping genomic data analytics algorithms into one or more relational database queries. In addition, we observe that genomic processing algorithms are composed of a mixture of specific algorithms as well as generic data manipulation operations and that the generic data manipulation operations are the common primitives that can be used across analytics domains. These commonalities allow the sharing, reusing, and composition of hardware modules across domains, lowering the development costs of highly efficient accelerated systems. These commonalities can be applied to many other big data analytics domains such as graph analytics.

Genesis hardware library demonstrates that constructing hardware accelerators by utilizing a set of composable hardware modules enables a flexible hardware accelerator design that can be easily extended or updated. We present a systematic way for accelerator researchers to decompose the algorithms into primitive operations and build hardware modules that directly map to those primitives as composable hardware blocks that form a hardware library. The algorithms are then composed using the hardware modules and the framework for ease of development. In domains where the algorithms are constantly changing, such as genomics, algorithm changes can be reflected quickly by simply updating or adding new hardware library components and recomposing the algorithm using the framework for rapid deployment. This development methodology can be adopted for various domains such as machine learning, having hardware library components to execute matrix–matrix multiply, matrix–vector multiply, etc.

In the world of accelerator research, efficiently mapping software onto custom hardware is a challenging problem. Researchers solve this problem by either using high-level synthesis to generate hardware or inventing new domain-specific languages that ease the mismatches between software and hardware. In this work, we advocate to leverage an already-standardized language as the domain-specific language and construct

primitive operators that directly map software primitives to hardware blocks to allow efficient mapping of software to hardware. The effect of this approach is realized in 1) the resultant accelerated systems achieving one order of magnitude better performance and cost-efficiency on FPGAs-in-the-cloud compared to multi-threaded software, and 2) the ease of adoption of the accelerated systems and the lowering of the barrier to entry for non-hardware-savvy scientists to use the accelerated systems, creating broader impacts.

ACKNOWLEDGMENTS

This work was supported in part by the Korean government grant (NRF-2016M3C4A7952587). This work was also funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy (Award Number DE-AR0000849), ADEPT Lab industrial sponsor Intel, RISE Lab and APEX Lab sponsor Amazon Web Services, and ADEPT Lab affiliates Google, Siemens, and SK Hynix.

REFERENCES

1. Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics coprocessor provides up to 15000x acceleration on long read assembly," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2018, pp. 199–213.
2. D. Fujiki *et al.*, "GenAX: A genome sequencing accelerator," in *Proc. Annu. Int. Symp. Comput. Archit.*, Jun. 2018, pp. 69–82.
3. L. Wu *et al.*, "FPGA accelerated INDEL realignment in the cloud," in *Proc. Int. Symp. High-Perform. Comput. Archit.*, 2019, pp. 277–290.
4. Oracle, "*Database SQL Tuning Guide—SQL Process*," [Online]. Avaialble: https://docs.oracle.com/database/121/TGSQL/tgsql_interp.htm#TGSQL94618 (URL)
5. T. J. Ham *et al.*, "Genesis: A. hardware acceleration framework for genomic data analysis," in *Proc. Int. Symp. Comput. Archit.*, Jun. 2020, pp. 254–267.

TAE JUN HAM is a Postdoctoral Researcher with Seoul National University, Seoul, South Korea. Contact him at taejunham@snu.ac.kr.

DAVID BRUNS-SMITH is currently working toward a Ph.D. degree with the Department of Electrical Engineering and Computer Sciences, University of California Berkeley, Berkeley, CA, USA. Contact him at bruns-smith@berkeley.edu.

BRENDAN SWEENEY is currently working toward a Ph.D degree with the Electrical and Computer Engineering Department, The University of Texas at Austin, Austin, TX, USA. This work was done when he was an undergraduate student at UC Berkeley. Contact him at brs@berkeley.edu.

YEJIN LEE is currently working toward a Ph.D. degree with the Computer Science and Engineering Department, Seoul National University, Seoul, South Korea. Contact her at yejinlee@snu.ac.kr.

SEONG HOON SEO is currently working toward a Ph.D degree with the Computer Science and Engineering Department, Seoul National University, Seoul, South Korea. Contact him at andyseo247@snu.ac.kr.

U GYEONG SONG is an undergraduate student with the Computer Science and Engineering Department, Seoul National University, Seoul, South Korea. Contact him at thddnrud2010@gmail.com.

YOUNG H. OH is currently working toward a Ph.D degree with the Semiconductor System Engineering Department, Sungkyunkwan University, Seoul, South Korea. Contact him at younghwan@skku.edu.

KRSTE ASANOVIC is a Professor with the Computer Science Division, Electrical Engineering and Computer Science Department, University of California Berkeley, Berkeley, CA, USA. Contact him at krste@berkeley.edu

JAE W. LEE is an Associate Professor of computer science and engineering (CSE) with Seoul National University (SNU), Seoul, South Korea. Contact him at jaewlee@snu.ac.kr.

LISA WU WILLS is the Clare Boothe Luce Assistant Professor of Computer Science and ECE with Duke University, Durham, NC, USA. Contact her at lisa@cs.duke.edu.