# Algorithm-Hardware Co-design for BQSR Acceleration in Genome Analysis ToolKit

Michael Lo*, Zhenman Fang†, Jie Wang*, Peipei Zhou*, Mau-Chung Frank Chang* and Jason Cong*
*University of California, Los Angeles, USA
milo168@ucla.edu, {jiewang, memoryzpp}@cs.ucla.edu, mfchang@ee.ucla.edu, cong@cs.ucla.edu
†Simon Fraser University, Burnaby, BC, Canada; zhenman@sfu.ca

***Abstract*—Genome sequencing is one of the key applications in healthcare and has a great potential to realize precision medicine and personalized healthcare. However, its computing process is very time consuming. Even pre-processing the raw sequence data of a whole genome for a single person to the analysis ready data can take several days on a single-core CPU.**

**In this paper, we propose to accelerate the performance of the widely used Genome Analysis ToolKit (GATK) using FPGAs. More specifically, we focus on the algorithm and hardware co-design for the Base Quality Score Re-calibration (BQSR) step in GATK, which is an important and time-consuming step to correct systematic errors made by a sequencing machine. Prior studies did not consider hardware acceleration for BQSR because it requires a large amount of memory with random access and has a lot of control flow. To address these challenges, we first adapt the algorithm to resolve the random memory access conflicts to achieve a fully pipelined accelerator design and reduce its dataset size. Second, we leverage the newly introduced large-capacity UltraRAM (URAM) in Xilinx UltraScale+ FPGAs to buffer BQSR's large dataset on chip, and further optimize its operating frequency. Finally, we also explore the coarse-grained pipeline and parallelism to improve the overall performance of the BQSR accelerator. Compared to the latest software implementation of BQSR on GATK 4.1, running on single-thread and 56-thread CPUs (14nm Xeon E5-2680 v4), our FPGA accelerator running on Xilinx 16nm UltraScale+ VCU1525 board achieves up to 40.7x and 8.5x speedups, respectively.**

## I. Introduction

Genome sequencing is one of the most important applications and has a great potential to reshape future healthcare systems. By sequencing a patient's genome and analyzing this genome against a reference genome, clinical professionals may precisely identify the patient's health issue and accurately prescribe corresponding medicine for treatment [1]. In the past decade, the cost to sequence a whole human genome has dramatically decreased, much faster than Moore's law. Two decades ago, it cost about $1,000,000 to sequence a human genome; now the cost is at the level of $1,000 per genome [2], affordable for health insurance providers and patients.

However, the high computing cost to analyze the genomes has become one of the major obstacles in the clinical adoption of genome sequencing. As shown in Figure 1, in genome sequencing, the first computing stage is the *data pre-processing* that transforms the raw sequence data, called *short reads*, to analysis-ready genome reads for the downstream *variant discovery* stage. Those short reads coming from a single run of the sequencer are considered as a read group. The

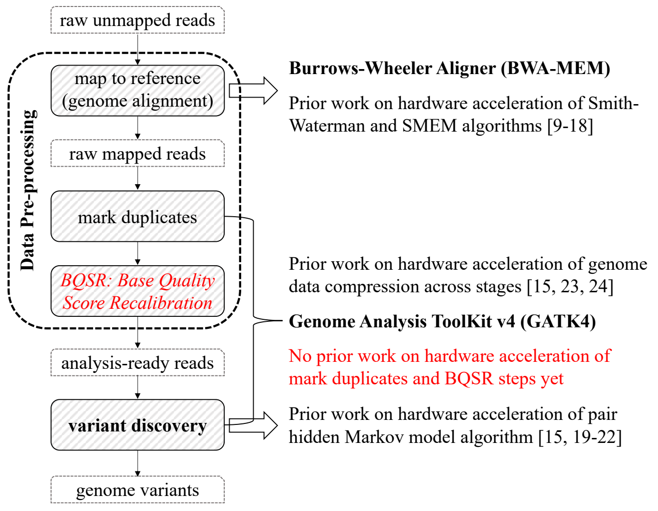
raw unmapped reads
map to reference (genome alignment)
raw mapped reads
mark duplicates
BQSR: Base Quality Score Recalibration
Data Pre-processing
analysis-ready reads
variant discovery
genome variants
Burrows-Wheeler Aligner (BWA-MEM)
Prior work on hardware acceleration of Smith-Waterman and SMEM algorithms [9-18]
Prior work on hardware acceleration of genome data compression across stages [15, 23, 24]
Genome Analysis ToolKit v4 (GATK4)
No prior work on hardware acceleration of mark duplicates and BQSR steps yet
Prior work on hardware acceleration of pair hidden Markov model algorithm [15, 19-22]

Fig. 1. Overview of the genome sequencing computation pipeline.

variant discovery stage is where the genome variants are discovered, allowing for individually specific suggestions of potential medicine and treatment [3]. In next-generation sequencing technology (NGS) [4], each short read is a small piece of *nucleobases* (usually around 50 to 400 *base pairs*, i.e., *nucleobases*), and a whole sequenced genome for a single person usually includes 3 billion base pairs and hundreds of millions of such short reads with high redundancy.

The data pre-processing stage further includes three major steps [3]. The first step is *map to reference (i.e., genome alignment)* that aligns short reads to a reference genome, which is usually done using the widely used open-source software package called BWA-MEM [5]. The second step is *mark duplicates* that mark short reads that are likely to come from the same genome fragments due to the artifactual process in the sequencing machine. The third step is *Base Quality Score Recalibration (BQSR)* that builds a statistical model to adjust quality scores for each read base by correcting systematic errors from the sequencing machine. The mark duplicates and BQSR steps, together with the variant discovery stage, are implemented in Broad Institute's latest open-source software package called Genome Analysis ToolKit version 4 (GATK4) [6]. GATK4 is implemented based on the in-memory map-reduce computing framework Apache Spark [7] and is widely used in the genome sequencing community.

According to Intel's white paper on deploying GATK best practices on Xeon CPUs [8], it took about 9.5 days to run the data pre-processing and variant discovery stages for a 30x coverage whole human genome on a single-core CPU, and 1.5 days on a 36-core CPU (6.3x speedup compared to single-core CPU). This significantly limits the potential clinical adoption of genome sequencing, especially for time sensitive cases such as cancer treatment. To further improve the computation performance of genome sequencing, prior studies have developed hardware accelerators for the genome alignment step [9]–[18], variant discovery step in GATK4 [15], [19]–[22], and genome data compression [15], [23], [24] in GATK4, by exploring abundant parallelism and customizable computation pipeline in these steps. These studies have shown great potential of hardware acceleration for genome sequencing. For example, the FPGA acceleration of the Smith-Waterman algorithm in the genome alignment step achieved 343.8x speedup over the single-core CPU implementation [9], and the FPGA acceleration of the pair hidden Markov model algorithm in the variant discovery step achieved 53x speedup over the well-optimized single-core CPU implementation [20].

In this paper, we focus on the algorithm-hardware co-design to accelerate the BQSR (Base Quality Score Recalibration) step, which is an important and time-consuming step in the latest GATK4 [6] software package. To the best of our knowledge, there is no prior work on hardware acceleration for the mark duplicates and BQSR steps yet, and we are the first to accelerate the BQSR algorithm on FPGAs. We choose to accelerate BQSR for two reasons: 1) BQSR is about 4x more time consuming than the mark duplicate step [25]; and 2) compared to other steps, BQSR has some unique challenges for hardware acceleration.

First, BQSR uses large covariate tables to characterize the quality score of each read base, which have a total size of around 8.7 MB. It is impractical to buffer such a large dataset on conventional Block RAM (BRAM) of an FPGA. Second, BQSR accesses these covariate tables in a random order, which makes it very difficult to efficiently customize the computation pipeline to achieve full pipeline with initiation interval (II) of one. Third, BQSR has a lot of control flows in its algorithm. As a result, this makes it more irregular and harder to accelerate on GPUs. More details of the BQSR algorithm and acceleration challenges are presented in Section II.

To tackle the above challenges, we propose algorithm and hardware co-optimization for the BQSR accelerator design. At the algorithm level, we first reduce the large size of covariate tables by half through changing the data precision of covariate table values from 64-bit to 32-bit; the average accuracy loss is less than 0.008% and is negligible. Moreover, we adapt the algorithm by buffering and merging potential conflict accesses in a small cyclic queue before reading and writing the covariate tables to resolve the conflicts of random memory accesses during the accelerator pipeline design. At the hardware design level, we leverage the newly introduced large-capacity UltraRAM (URAM) in Xilinx UltraScale+ FPGAs, which has about a 4x larger capacity than conventional BRAM, to buffer BQSR's large covariate tables on chip. To address the low frequency issue caused by the long critical path that the URAM blocks span, we pipeline off-chip data at multiple checkpoints along the critical path and improve the operating frequency of our accelerator design by 25%. Moreover, to optimize the overall performance of the BQSR accelerator, we fully pipeline the design of each computing engine and explore the coarse-grained pipeline and parallelism between computing engines. Finally, we reorganize the input data layout to maximize the off-chip memory bandwidth utilization.

Our entire BQSR accelerator design is implemented in high-level synthesis (HLS) C++ and built using Vivado HLS 2018.3. It runs at 122MHz and utilizes 74% of the URAM resource, which is the bottleneck resource. For the software baseline, we run the latest GATK 4.1 package on a 28-core server with dual-socket 14nm Xeon E5-2680 v4 CPUs. Our FPGA accelerator running on Xilinx 16nm UltraScale+ VCU1525 board achieves up to 40.7x and 8.5x speedups over the single-thread and 56-thread versions, respectively.

## II. BQSR Algorithm and Challenges

### A. Base Quality Score Re-calibration (BQSR) Algorithm

Since today's sequencing machines are error prone, they also report a Phred quality score for each base in the sequenced reads to characterize the confidence of the base accuracy. Such quality scores affect the accuracy of downstream analysis, such as the genome wide association study and precision medicine. BQSR [26] aims to detect and correct patterns of systematic biases of the short reads from the sequencing machine by generating a model using these reported quality scores.

*1) Covariates:* To generate the quality score model, the following four features (i.e., covariates) of each base in the short reads are explored in BQSR. First, *read group covariate* describes which group the read strand belongs to. Second, *quality score covariate* characterizes the association of mismatches with each individual quality score. Third, *sequence context covariate* characterizes the association of mismatches with the sequencing context; for example, dinucleotide 'AC' often has much lower quality than 'TG'. Fourth, *cycle sequence covariate* characterizes the association of mismatches with machine cycles on which the sequencer is on.

*2) Covariate Tables:* Each covariate has its own table, where each table entry includes a value pair {*#occur, error*} with 128 bits: one is the number of occurrences the base has appeared (*#occur*), which is a 64-bit integer in GATK4; and the other is the accumulation of mismatch error values for the base (*error*), which is a 64-bit double-precision floating point in GATK4. The read group and quality score covariate tables are 2 and 3 dimensional, respectively; while both the sequence context and cycle sequence covariate tables are 4 dimensional. The first dimension has 3 types of error events: 1) mismatch, if the given base and reference base do not match; 2) insertion, if an extra base was read when it should not be read, and 3) deletion, if a base was not read when it should have been. The second dimension is the number of read groups. The third dimension is the number of possible quality scores that range from 0 to 93. The last dimension is a covariate's specific size: it is 1012 for context covariate and 1002 for cycle covariate.

*3) Algorithm 1:* This algorithm describes the core algorithm of the BQSR step in GATK4 [6]. This takes up to 98% of the execution time of BQSR and is a good candidate for hardware acceleration. BQSR uses a histogram-like algorithm and has two major functions for each read: `COMPUTE_COVARIATE_INDICES` and `UPDATE_COVARIATE_TABLES`.

Function `COMPUTE_COVARIATE_INDICES` uses each read's bases, quality scores, and read group (RG) to generate four groups of indices (*idxRG, idxQual, idxCtx,* and *idxCyc*) that will be employed to update the four covariate histograms, i.e., read group, quality score, sequence context, and cycle sequence covariate tables. Each group of indices is a 2 dimension array: the first dimension is the type of error event, and the second dimension is the base index inside the read.

1. Lines 2-4 in Algorithm 1 are used to compute the read group covariate index *idxRG*. We only need to get the read group number (*RG*) from the input genome to index the second dimension of the table.
2. Lines 5-7 are used to compute the quality score covariate index *idxQual*. We only need to get the mismatch/insertion/deletion quality scores (*BaseQ, InQ, DelQ*) from the input genome to index the third dimension of the table.
3. Lines 8-22 are used to compute the sequence context covariate index *idxCtx*. We only need to compute the mismatch and insertion/deletion context values *ctxM* and *ctxIndel* to index the fourth dimension of the table. As shown in lines 9-12 and lines 18-22, we only compute the indices for a valid range of bases inside a read. We also complement the base value for a negative read strand (lines 13-15), e.g., changing 'A' to 'T' and 'G' to 'C', etc. The *CtxM* and *CtxIndel* values are calculated through the *Context* function (lines 16-17), which mainly involves bit shift and mask operations for the bases and a constant valued key mask (*mismatchMask* or *indelMask*).
4. Lines 23-35 are used to compute the cycle sequence covariate index *idxCyc*. We only need to compute the mismatch and insertion/deletion key values *subKey, indelKey* to index the fourth dimension of the table. As shown in lines 24-26, it calculates the machine *cycle* and *inc* step from the *read order factor (ROF)*. It then reverses them if it is a negative strand (lines 27-29). By using the `keyFromCycle` function to generate a value through negation, addition, and shift operations based on the *cycle* input, the code calculates the *subKey* for each base in line 31. The *indelKey* is the same as *subKey* for the valid range of bases (lines 32-33).

**Algorithm 1** Pseudo code for software BQSR for 1 read

```
 1: function COMPUTE_COVARIATE_INDICES
 2:     #1. Generate Indices for Read Group (RG) Covariates:
 3:     for each base do //base index: b; 3 types of errors
 4:         idxRG[3][b] $\leftarrow$ {RG,RG,RG}
 5:     #2. Generate Indices for Quality Covariates:
 6:     for each base do
 7:         idxQual[3][b] $\leftarrow$ {BaseQ,InQ,DelQ}
 8:     #3. Generate Indices for Context Covariates:
 9:     {leftClipIdx, rightClipIdx} $\leftarrow$ GetClipIdx(bases)
10:     for each base do
11:         if b $<$ leftClipIdx || $>$ rightClipIdx then
12:             invalidate base[b]
13:     for each base do
14:         if read is negative strand then //len: # of bases
15:             base[b] $\leftarrow$ complement(base[len-b-1])
16:     CtxM $\leftarrow$ Context(mismatchMask,MaskSize,bases)
17:     CtxIndel $\leftarrow$ Context(indelMask,MaskSize,bases)
18:     for each base do
19:         if !(leftClipIdx $>$ rightClipIdx) then
20:             idxCtx[3][b] $\leftarrow$ {ctxM,CtxIndel,CtxIndel}
21:         else
22:             idxCtx[3][b] $\leftarrow$ {-1,-1,-1}
23:     #4. Generate Indices for Cycle Covariates:
24:     ROF $\leftarrow$ (isReadPairs && isSecondOfPair) ? -1 : 1
25:     cycle $\leftarrow$ ROF //ROF: Read Order Factor
26:     inc $\leftarrow$ ROF
27:     if read is negative strand then
28:         cycle $\leftarrow$ len * ROF
29:         inc $\leftarrow$ -1 * ROF
30:     for each base do
31:         subKey $\leftarrow$ keyFromCycle(cycle)
32:         //T1 and T2 are two constant thresholds
33:         indelKey $\leftarrow$ (b $<$ T1 || b $>$ T2) ? -1 : subKey
34:         idxCyc[3][b] $\leftarrow$ {subKey,indelKey,indelKey}
35:         cycle += inc
36: function UPDATE_COVARIATE_TABLES
37:     for each event type do
38:         for each valid base do
39:             for each covariate do
40:                 if idx[] is valid then
41:                     updateTable(idx[],{#occur,error})
```

Function `UPDATE_COVARIATE_TABLES` updates the values of the four covariate tables. For each covariate table, based on the valid base index and event type (lines 37-39 in Algorithm 1), it reads the corresponding multi-dimensional indices *idx[]* to the table from the results generated by function `COMPUTE_COVARIATE_INDICES`. For a valid idx[], the function will update the value pair {*#occur, error*} of the corresponding table entry by accumulating (reading-adding-writing) the number of occurrences (*#occur*) and mismatch (*error*) value of that base (lines 40-41 in Algorithm 1).

For more details about the BQSR algorithm, please refer to Broad Institute's documents on GATK4 [6] and BQSR [26].

## B. BQSR Acceleration Challenges

*1) Challenge 1: Large Dataset:* Table I summarizes the storage requirements of the four major covariate tables (in function `UPDATE_COVARIATE_TABLES`) and their corresponding table indices used in BQSR if it is implemented on hardware. Their element size, number of elements, and total size is included. This is assuming that we accelerate one read group at a time (accelerating K read groups at a time will increase the covariate table sizes by K times) and there is at most 1,000 bases per read (long enough for next-generation sequencing). To efficiently accelerate BQSR, a minimum on-chip storage of 8.7MB is required, but this amount is very difficult to fit onto conventional BRAMs of an FPGA. For

TABLE I. Major storage requirements for hardware acceleration

| Name | Element Size | Num of Elements | Total Size |
|---|---|---|---|
| 4*2-D Table Indices | 32-bits | 4x3x1000 | 46.88KB |
| 2-D Read Group Table | 128-bits | 3x1 | 48B |
| 3-D Quality Table | 128 bits | 3x1x94 | 4.41KB |
| 4-D Context Table | 128 bits | 3x1x94x1012 | 4.35MB |
| 4-D Cycle Table | 128 bits | 3x1x94x1002 | 4.31MB |

example, the Xilinx UltraScale+ VCU1525 board we use in this paper—which has the same FPGA chip as the one used in Amazon F1 instance [27]—has a total of 8.9MB BRAMs. Typically, one cannot use more than 80% of the BRAM resource in order to build the FPGA design. Moreover, it is also very difficult to apply data tiling for the BQSR algorithm due to the random memory access behavior explained below.

*2) Challenge 2: Random Memory Access Conflict:* When updating (it is actually *read-update-write*, for simplicity, we use *update* throughout the paper) the covariate tables, the indices that come in have no particular order or pattern. This random memory access behavior presents a considerable challenge to accelerating the performance of the covariate table updating stage. First, it leads to the synchronization issue when two table updates index the same position: the second update has to happen after the first one, as illustrated in Figure 2a). Second, due to the large table sizes and limited on-chip cache or buffer sizes, it is hard to avoid the synchronization by simply replicating the covariate tables for parallel access. Moreover, unless we can ensure two random updates use different indices, as illustrated in Figure 2b), when we try to pipeline the table update on an FPGA—assuming we can buffer tables on chip—this random memory access conflict issue prevents the high-throughput design with a pipeline initiation interval (II) of one.

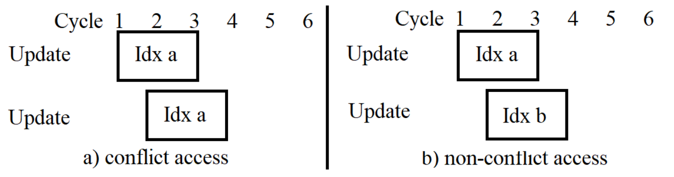

Fig. 2. An example of random memory access conflict: a) both updates use index a to the table, the second update has to happen after the first one; b) two updates using different indices can be fully pipelined with II=1.

*3) Challenge 3: Control Flow Divergence:* When computing the sequence context and cycle sequence covariate indices, many computations depend on input properties of the short read, leading to control flow divergence. This divergence also exists when updating the covariate tables. Therefore, it is difficult for GPU acceleration. But it is natural for an FPGA accelerator to fully pipeline the design with control flow.

## III. BQSR Accelerator Design

### A. Hardware-Friendly Algorithm Optimization

In order to efficiently accelerate BQSR on FPGA, we propose several algorithm-level optimizations. The pseudo code of the hardware-friendly algorithm is presented in Algorithm 2, with the changes from Algorithm 1 highlighted in *red italic*.

**Algorithm 2** Pseudo code for hardware BQSR for 1 read

```
 1: function LOAD_INPUT
 2:     for each base do //pipeline II=1; base index: b
 3:         rawInput[b] ← off-chip DRAM
 4: function COMPUTE_COVARIATE_INDICES
 5:     for each base do //pipeline II=1
 6:         parseInput(rawInput[b])
 7:     {leftClipIdx, rightClipIdx} ← GetClipIdx(bases)
 8:     //Merge two loops (lines 10-15) in Algorithm 1
 9:     for each base do //pipeline II=1
10:         if read is negative strand then //len: # of bases
11:             if len-b < leftClipIdx || > rightClipIdx then
12:                 invalidate base[b]
13:             else
14:                 base[b] ← complement(base[len-b-1])
15:         else
16:             if b < leftClipIdx || > rightClipIdx then
17:                 invalidate base[b]
18:     CtxM ← Context(mismatchMask,MaskSize,bases)
19:     CtxIndel ← Context(indelMask,MaskSize,bases)
20:     ROF ← (isReadPairs && isSecondOfPair) ? -1 : 1
21:     cycle ← ROF //ROF: Read Order Factor
22:     inc ← ROF
23:     if read is negative strand then
24:         cycle ← len * ROF
25:         inc ← -1 * ROF
26:     //Merge the four indices computing in Algorithm 1
27:     for each base do //pipeline II=1;
28:         subKey ← keyFromCycle(cycle)
29:         //T1 and T2 are two constant thresholds
30:         indelKey ← (b < T1 || b > T2) ? -1 : subKey
31:         for each covariate do //unrolled
32:             for each event do //unrolled
33:                 if valid base then
34:                     idx[] ← {RG,BaseQ,InQ,DelQ,CtxM,
35:                              CtxIndel,subKey,indelKey}
36:         cycle += inc
37: function UPDATE_COVARIATE_TABLES
38:     //Swap the loops in Algorithm 1
39:     for each covariate do //unrolled for parallelism
40:         for each event do //unrolled for parallelism
41:             for each valid base do //pipeline II=1;
42:                 //Buffer and partially merge table updates
43:                 //in queue[Q] to avoid memory conflicts
44:                 found ← Find&Merge(queue[Q],idx[])
45:                 if queue[b%Q] is valid then
46:                     updateTable(queue[b%Q])
47:                     invalidate queue[b%Q]
48:                 if !found then
49:                     queue[b%Q] ← {idx[],#occur,error}
```

1. Function LOAD_INPUT is added in Algorithm 2 lines 1-3, which is used to load raw input genome data from off-chip DRAM to on-chip URAM.
2. We merge multiple loops into one whenever possible so that they can be accelerated in a single pipeline and executed concurrently on hardware. For example, we merge two loops in Algorithm 1 (lines 10-15) into one in Algorithm 2 (lines 8-17). Then we combine the four covariate indices computing loops into one in Algorithm 2 (lines 27-37).
3. To reduce the covariate table sizes, we change the type of the table value #occur (number of occurrences) from 64-bit

```
 1 Datum queue[Q]; //queue to buffer and merge table updates
      Datum struct holds table entry {idx[], #occur, error}
 2
 3 //Iterate each valid base: from lines 42-50 in Algorithm 2
 4 for(int b = 0; b < READ_LENGTH+Q; b++){
 5    #pragma HLS pipeline II=1
 6    #pragma HLS dependence variable=table inter true
          distance=Q
 7
 8    //Get info of current table entry to update
 9    Datum current = getTableEntryInfo(covariate, event, b);
10
11    //Find if current entry is in queue, if yes, merge the
          partial results to the found entry in queue
12    bool found = false;
13    MERGE: for(int q = 0; q < Q; q++){ //Unrolled by HLS
14       if(current.index == queue[q].index){
15          queue[q].numOccurance += 1;
16          //type of partial mismatchError in queue is int
17          queue[q].mismatchError += current.mismatchError;
18          found = true;
19       }
20    }
21
22    //Update actual covariate tables using the queue entry
          indexed by b%Q (if valid), then free this entry.
          This queue delays the conflict access (index b+Q
          vs b) to covariate table by Q iterations
23    if(queue[b%Q].index >=0) {
24       updateTable(covariateTables,queue[b%Q]);
25       queue[b%Q].index = -1;
26    }
27
28    //If current entry is not in queue, insert it into
          queue at the position indexed by b%Q
29    if(found == false){
30       queue[b%Q].index = current.index;
31       queue[b%Q].numOccurance = 1;
32       queue[b%Q].mismatchError = current.mismatchError;
33    }
34 }
```

Listing 1. Major code to fully pipeline the covariate table updating (lines 42-50 in Algorithm 2) by buffering and partially merging potential conflict accesses in a small cyclic queue.

integer to 32-bit integer as there is at most 3 billion bases in a human genome. We also alter the type of the table value error (accumulated mismatch error) from a 64-bit double to a 32-bit single floating point, with an average accuracy loss less than 0.008%. As a result, the table sizes are reduced by half to about 4.3MB.

4. The nested loops in the covariate table update have been swapped, as shown in Algorithm 2 (lines 39-42), enabling it to efficiently pipeline the base loop (line 42) and parallelize the outer loops (lines 40-41).
5. Last and most importantly, to resolve the random memory access conflict issue, we add a small cyclic queue of size Q to buffer and partially merge potential conflicts before updating the covariate tables, as shown in Algorithm 2 lines 43-50. A more thorough code is shown in Listing 1. As shown in lines 11-20 (*MERGE loop*), for the current table entry associated with base b, the algorithm first checks if the table index is already in the merge queue. If yes, it partially accumulates the results to the found entry in the queue. Otherwise, it inserts this table entry into the merge queue using the cyclic index b%Q (lines 28-33). The essential part is in lines 22-26: Instead of updating the covariate table directly using the current table entry which might lead to access conflict with a prior update (due to the random access behavior), now we use the queue entry indexed by b%Q to update the covariate table. Since every entry in the queue is different, it is guaranteed that there will be no conflict access within Q iterations of the base loop (line 4).

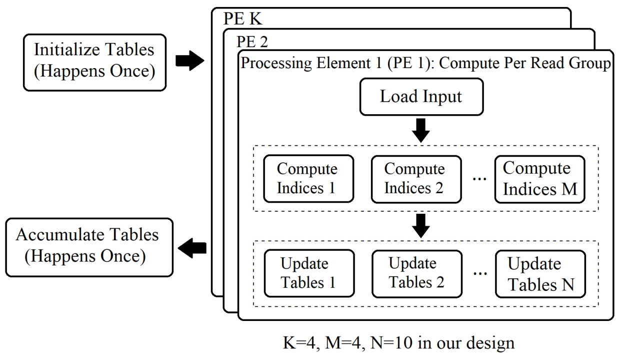
PE K
PE 2
Processing Element 1 (PE 1): Compute Per Read Group
Initialize Tables (Happens Once)
Load Input
Compute Indices 1
Compute Indices 2
...
Compute Indices M
Update Tables 1
Update Tables 2
...
Update Tables N
Accumulate Tables (Happens Once)
K=4, M=4, N=10 in our design

Fig. 3. Overview architecture of the BQSR accelerator.

## B. BQSR Accelerator Design and Optimization

Based on the optimized algorithm, we design and implement a parallel and fully pipelined BQSR accelerator on FPGA in Vivado HLS C++. We leverage the large-capacity URAM resource to buffer the large covariate tables on-chip and further optimize its operating frequency. Moreover, we optimize the merge queue design in hardware to resolve URAM access conflict and thus achieve full pipeline with initiation interval (II) of one. Figure 3 gives an overview architecture of our BQSR accelerator.

1. *Initialize Tables.* This beginning stage initializes covariate tables on URAM to 0, as URAMs (unlike BRAMs) do not automatically initialize themselves. It only executes once.
2. *Load-Compute-Update Processing Element (PE).* This is the main component to implement Algorithm 2, and each PE is designed to execute one read group, which essentially removes the read group dimension needed for the covariate tables and thus minimizes the on-chip storage requirement. While the number of read groups can dynamically range from 1 to any number in the software version, now they can always be distributed to our PEs without the hardware accelerators underutilized. Since we reduce the covariate table sizes and have large-capacity URAMs, we create K copies of covariate tables and K parallel PEs. Each PE implements three computing engines—*Load Input*, *Compute Indices*, and *Update Tables*—corresponding to the three functions in Algorithm 2. We further explore coarse-grained pipeline between these three stages by using a ping-pong buffer and balance them by parallelizing *Compute Indices* with M duplications and *Update Tables* with N duplications. We present more details of these stages and corresponding hardware optimizations below.
3. *Accumulate Tables.* This stage merges the K copies covariate tables into a single copy and writes it back to off-chip memory. It only executes once at the end of the program.

*1) Load Input and Data Layout Re-organization:* This stage loads the raw input genome data from off-chip memory to on-chip URAM and is fully pipelined with II=1. To maximize the

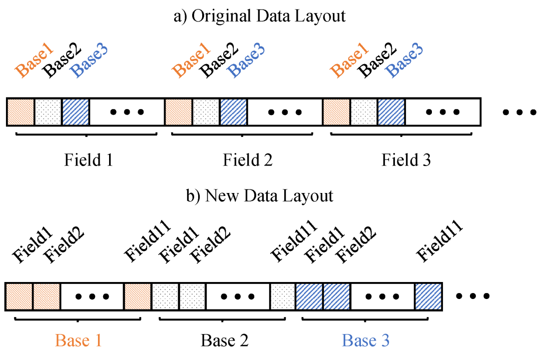

Fig. 4. Input read data layout re-organization to pack all data fields of one base into a single consecutive chunk.

off-chip memory bandwidth, we pack off-chip data transfer in 512-bit chunks. There are 11 data fields for each base. Originally, for each field, data fields from multiple bases are packed together into a single 512-bit chunk, shown in Figure 4a). As a result, it needs multiple memory accesses to grab all data fields of one base. To address this limitation, we change the input data layout so that one 512-bit chunk packs all the fields for a single base, as shown in Figure 4b).

*2) Compute Indices:* This stage computes all covariate table indices. Shown in Algorithm 2, we have pipelined all loops in function `COMPUTE_COVARIATE_INDICES` with II=1, and we parallelize this engine for reads inside a read group.

*3) Update Tables:* As shown in Algorithm 2, we parallelize the two outer loops (covariate and event loops in lines 40-41) and fully pipeline the innermost base loop (line 42) with II=1. Eliminating the random memory access conflicts when updating the table is the biggest challenge to achieving II=1. Based on the revised algorithm using a small cyclic merge queue as presented in Listing 1, our hardware implementation needs to provide 1) a shorter latency of the merge step in the queue than that of a covariate table update, and 2) a large enough queue size Q, so that before one covariate table update using *queue[b%Q]* is completed, it will not cycle back to the update using the queue element *queue[(b+Q)%Q]*, which is the same as *queue[b%Q]*.

1. *Shorter queue merge latency*. In the *MERGE loop* shown in lines 13-20 of Listing 1, it only accumulates a very small number of potential conflict entries (<= the queue size Q) in the queue. Therefore, it does not need all bits in a single floating point to store the partially accumulated mismatch error value. This gives us an opportunity to downgrade the *float* type of the mismatch error value to an *int* type in the *MERGE loop*, which has much faster latency for the arithmetic operations (1 cycle in our design) compared to the float operations [28]. To avoid the value overflow and minimize the accuracy loss, we convert the 32-bit float to a 32-bit integer by multiplying $2^{27}$ in the *MERGE loop*, then we convert it back to the regular 32-bit float when updating the covariate table in URAM (line 24 in Listing 1). The average accuracy loss is less than 0.008%.
2. *Queue size Q selection.* To make sure the second covariate table update will not cycle back to use the queue element *queue[(b+Q)%Q]* before the first covariate table update using *queue[b%Q]* is completed, we need a queue size: $Q >= CovariateUpdateLatency/QueueMergeLatency$ In our design, it takes 1 cycle for the *MERGE loop* so $QueueMergeLatency = 1$. The latency of updating the covariate table $CovariateUpdateLatency$ (dominated by the mismatch error value update) includes two parts: 1) the integer to floating point conversion that requires 2 cycles, and 2) the floating point accumulation in HLS that requires 16 cycles. Therefore $CovariateUpdateLatency = 18$ and $Q >= 18$. On the other hand, the larger Q is, the longer the critical path of the pipeline is (we fully pipeline the base loop in line 4 of Listing 1). Therefore, we choose the smallest queue size $Q = 18$ in our design.

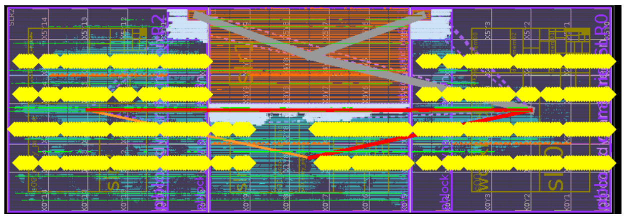

Fig. 5. BQSR accelerator layout on Xilinx VCU1525 FPGA board: Yellow diamonds represent URAMs used in the design, the thin limegreen lines indicate BRAMs used, and the red line indicates the critical path.

*4) URAM Optimization for Higher Frequency:* With data reading and writing, timing issues usually arise in the critical paths between the DRAM interface and the on-chip storage, especially URAMs blocks. As shown in Figure 5, URAM blocks span the whole FPGA chip (in fact, 3 dies in the Xilinx UltraScale+ VCU1525 FPGA board in Figure 5). However, for each iteration of the pipeline, updates on URAM covariate tables need to be completed in 1 cycle to achieve II=1. As a result, it lowers the overall accelerator frequency. To address the frequency issue, we pipeline the data to multiple checkpoints along the critical path using the open-source tool Latte [29]. By increasing some latency to the data loading, we improve the accelerator frequency by 25%. As presented in Section IV-D, the increased latency in the *Load Input* stage can be hidden in our coarse-grained pipeline design.

## IV. Results and Analysis

### A. Experimental Setup

We run the latest version of the widely used GATK4.1 [6] for the software baseline. This is developed on top of the in-memory map-reduce computing framework Apache Spark. For the CPU, we use a dual socket 14nm Xeon E5-2680 v4 CPU server that has 28 cores and 72GB DRAM; each CPU core also has 2 hyper-threads. For the software version, we increase the number of threads from 1 to 2 to 4, and up to 56.

Our FPGA accelerator is designed using Xilinx Vivado HLS 2018.3. It supports up to 1000 bases per read, which is generally enough for short reads in next-generation sequencing; it can be increased if necessary since the processing of each base is fully pipelined and it does not add too much on-chip storage. The FPGA platform we run on is Xilinx's 16nm

TABLE II. GENOME DESCRIPTION AND ITS SINGLE-CORE RUN-TIME

| Name | Source | # of Reads | File Size | Time (s) |
|---|---|---|---|---|
| SRR2114965 | [30] | 42,888,871 | 57 GB | 481.66 |
| HCC1954 | [31] | 18,373,093 | 30 GB | 245.15 |
| NA12878 | [32] | 57,962,777 | 134 GB | 985.18 |

Virtex UltraScale+ VCU1525 FPGA board, which has the same FPGA chip as that in Amazon F1 instance [27]. On this board, we can put 4 processing elements (PEs) and, inside each PE, there is 1 *Load Input* engine, 4 *Compute Indices* engines, and 10 *Update Tables* engines, as shown in Figure 3. The accelerator runs at a frequency of 122MHz.

For input genomes, we select three random human genomes from the 1,000 genome project database [33]. We also randomly chop them into smaller segments with a different number of short reads (18 million to 58 million short reads), so that the single-thread software version of the BQSR step can finish within 20 minutes. Table II summarizes the genome (segment) name, its source, the number of short reads it includes, its file size, and the execution time (in seconds) on a single-core CPU.

## B. Resource Utilization and Performance Upper Bound

We summarize the resource utilization of our current BQSR accelerator design on the Xilinx VCU1525 FPGA board in Table III. Note that Vivado HLS usually only allows designers to use up to 70-80% of any FPGA resources; otherwise, the build will fail. As shown in Table III, the performance of our accelerator is limited by the number of URAM resource, where it already occupies 74%. We cannot add any more processing elements that require extra URAM blocks.

TABLE III. BQSR RESOURCE UTILIZATION ON FPGA

| BRAM | DSP | FF | LUT | URAM |
|---|---|---|---|---|
| 50% | 4% | 10% | 50% | 74% |

Between the URAM and BRAM usage, we buffer the large covariate tables (dominating resource) and raw input genome data onto URAM. And we buffer the covariate table indices and the parsed covariate table value pairs {#occur,error} for the current K (K=4) read groups that the accelerator is processing onto BRAM.

## C. Overall Speedup

Figure 6 presents the speedups of multi-thread CPU versions and our FPGA accelerator version over the single-thread CPU version. To make a fair comparison, for the CPU version, we only measure the time it takes to do the index computations and table updates, which are the same ones we measure for the FPGA version. The execution time for the FPGA version includes CPU to FPGA data transfer but does not include the input data layout reorganization. Due to the random memory access conflicts and large dataset, the CPU performance does not scale linearly with the number of threads. Depending on the input genome, the 56-thread CPU version only achieves 4.8x to 7x speedup. While our FPGA accelerator achieves 35x to 40.7x speedup over the single-thead CPU version, and 5.5x to 8.5x speedup over the 56-thread CPU version.

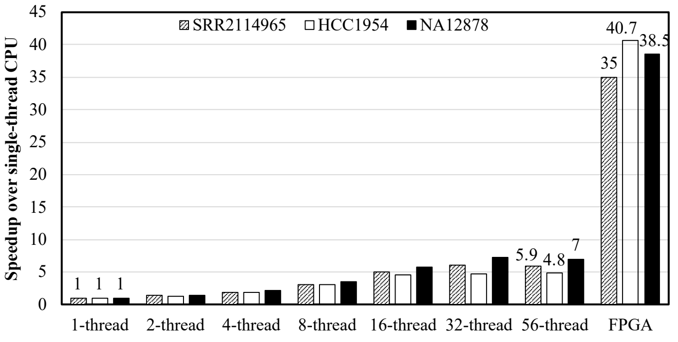

Fig. 6. Overall speedup comparison of FPGA and multi-thread CPU versions

We applied the double to float conversion to the software version of BQSR as well, but found that there was little impact on the CPU performance since the random memory access behavior dominates its performance. The same algorithm optimization using a small cyclic queue to eliminate memory access conflicts does not work well in the CPU version since it requires the corresponding hardware customization as well.

## D. Accelerator Efficiency Analysis

We further analyze the accelerator efficiency to demonstrate that our accelerator design has been well optimized. First, as presented in Algorithm 2, all the major loops in our HLS-based accelerator have been fully pipelined with II=1. Second, we analyze the workload balance between the *Load Input*, *Compute Indices*, and *Update Tables* stages inside each PE, where coarse-grained pipeline optimization has been applied using ping-pong buffer. The execution latencies of these three stages are summarized in Table IV under different read lengths (number of bases). The dominating stages are the *Compute Indices* and *Update Tables* stages (columns 'Compute' and 'Update'). These are balanced between each other with 4 parallel copies of *Compute Indices* and 10 parallel copies of *Update Tables*. The original *Load Input* stage (column 'Load-Original') was short in execution latency, but lowered the clock frequency. After the frequency optimization with Latte [29], the latency of the new *Load Input* stage (column 'Load-Latte') becomes 70%-85% of that of the *Compute Indices* and *Update Tables* stages, which is still hidden by those two stages.

TABLE IV. LATENCY FOR *Load Input*, *Compute Indices (4 copies)*, AND *Update Tables (10 copies)* STAGES UNDER DIFFERENT READ LENGTHS

| Read Length | Load-Original | Load-Latte | Compute | Update |
|---|---|---|---|---|
| 50 | 121 | 207 | 251 | 270 |
| 100 | 209 | 328 | 451 | 470 |
| 150 | 302 | 534 | 651 | 670 |
| 250 | 482 | 902 | 1051 | 1070 |
| 500 | 937 | 1713 | 2051 | 2070 |
| 1000 | 1842 | 3338 | 4051 | 4070 |

## E. Accuracy Analysis

Since we reduce the precision of the accumulated mismatch error value in the covariate tables from the 64-bit double to a 32-bit float and further convert the 32-bit float to a 32-bit integer for the partial accumulation results in the small cyclic merge queue, there is some accuracy loss for this data field.

TABLE V. HIGHEST AND AVERAGE ERROR RATE FOR THE ACCUMULATED MISMATCH ERROR VALUE IN THE COVARIATE TABLES

| | Single-Precision | | Half-Precision | |
|---|---|---|---|---|
| Genome | Highest | Average | Highest | Average |
| SRR2114965 | 0.0131% | 0.0078% | 97.16% | 37.18% |
| HCC1954 | 0.2247% | 0.0008% | 92.56% | 16.29% |
| NA12878 | 0.0231% | 0.0017% | 98.9% | 42.32% |

We profile the average and highest error rates for each input genome, as shown in Table V. Depending on the input genome, the highest error rate for a single table value is from 0.0131% to 0.2247%, while the average error rate for all table values is from 0.0008% to 0.0078%, which can be well tolerated.

To check whether we can save more on-chip storage resource, we also profile the average and highest error rates for each input genome using 16-bit half-precision floating point numbers. As shown in Table V, when using half-precision, the average error rates are high and the highest error rates are well over 90%. This is because there are not enough bits in the exponent field of half-precision float and there are frequent value overflows. Therefore, we choose to utilize the 32-bit floating point in our design.

## V. RELATED WORK

In addition to the genome sequencing acceleration work mentioned in Section I, we further discuss two major categories of related work: 1) acceleration for parallel histogram computation and 2) hardware acceleration leveraging URAM.

### A. Acceleration for Parallel Histogram Computation

There are many prior studies that accelerate the histogram-based applications on FPGA [34]–[38]. To solve the memory conflict in the updates of histogram tables, these studies resort to duplicating either the computation or memory resources.

Gautam [34] proposed an FPGA accelerator for calculating the histogram using a map-reduce fashion. The frequency of each input element is calculated in parallel and then reduced by a shuffle network. However, the computation resource of this architecture is proportional to the input and output size and therefore it is not scalable for handling large-scale histogram problems like the BQSR algorithm.

Maggiani et al. [35] exploited parallelism by duplicating the histogram tables. The number of duplicated tables equals the latency of the update operation. This will also cause scalability issues as the storage requirements for the duplicated tables could easily go beyond the on-chip limit when handling large tables and complicated update operations with long latency. A similar idea is explored in [36]–[38].

When accelerating the BQSR algorithm in GATK4.1, the large size of histogram tables and long latency of floating point table update make it impractical to simply duplicate the computation or memory resources. Instead, in this work, we introduce a novel small cyclic queue to buffer and quickly merge potential conflicts to hide the histogram update latency. The queue size is the same as the histogram update latency, which is much smaller compared to the histogram table size. Our design fully pipelines the processing element (PE) without the need to replicate a large amount of computation or memory resources. Only if there is enough computing and memory resource will our fully-piplined PE duplicate (e.g., 4 copies on the Xilinx VCU1525 FPGA board). Our architecture is highly scalable in handling complicated and large-scale histogram-based applications like BQSR.

### B. Acceleration Leveraging UltraRAM

UltraRAM (URAM) is first introduced on Xilinx UltraScale+ devices which adds about 4x more on-chip memory [39]. The large volume of URAM enables designers to buffer more data on-chip, saving off-chip communication and improving the overall performance. It has been used by many previous accelerator designs [40]–[43]. In this work, we use URAM to buffer the histogram tables on-chip to enable fast random access. The use of URAM often brings frequency issues due to the long data wires. This issue can be alleviated by either scattering the data and localizing them to each PE [41], [43], or pipelining the data transfer onto multiple checkpoints [29]. Since tiling is prohibited in BQSR due to its random data access, we have chosen the second approach that helps us increase the design frequency.

## VI. CONCLUSION

In this paper, we presented the first algorithm and hardware co-design to accelerate the Base Quality Score Re-calibration (BQSR) algorithm in GATK4. This algorithm is an important and time-consuming step to correct systematic errors of a genome from a sequencing machine. To address BQSR's unique challenges of large covariate table size and random meory access conflicts when updating the tables, we optimized the algorithm by reducing its table size with a lower data precision and resolving the memory access conflict with a small cyclic queue that buffers and quickly merges potential conflict accesses before updating the tables. At the hardware design, we buffered the large tables on chip by leveraging newly introduced URAMs, optimized its operating frequency, and fully pipelined the accelerator design by implementing a fast merge queue. Moreover, we also explored coarse-grained pipeline and coarse-grained parallelism to achieve the optimal performance of our BQSR accelerator. Compared to the single-thread and 56-thread implementations running on the 14nm dual socket Xeon E5-2680 v4 CPU server, our HLS-based FPGA accelerator achieved up to 40.7x and 8.5x speedups on Xilinx 16nm UltraScale+ VCU1525 board. We believe these optimizations are not limited to the BQSR algorithm and can be applied to other histogram-like algorithms.

## ACKNOWLEDGEMENTS

We acknowledge the support from Center for Domain-Specific Computing and its industrial partners, including Huawei, Samsung, and VMWare; Natural Sciences and Engineering Research Council of Canada (NSERC Discovery Grant RGPIN-2019-04613 and DGECR-2019-00120); Simon Fraser University New Faculty Start-up Grant; and Xilinx. We also thank Brian Hill for earlier discussions, Marci Baun for editing the paper, and Amazon for AWS credit donation.

## References

[1] G. S. Ginsburg and K. A. Phillips, "Precision medicine: From science to value," in *Health Aff (Millwood)*, vol. 37, 2018, p. 694–701.

[2] National Human Genome Research Institute, "The cost of sequencing a human genome," 2019. [Online]. Available: https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost

[3] Broad Institute, "Genome Analysis Toolkit (GATK) Best Practices," 2019. [Online]. Available: https://software.broadinstitute.org/gatk/best-practices/workflow?id=11145

[4] J. M. Besser, H. A. Carleton, P. Gerner-Smidt, R. L. Lindsey, and E. Trees, "Next-generation sequencing technologies and their application to the study and control of bacterial infections," in *Clinical Microbiology and Infection: Official Publication of the European Society of Clinical Microbiology and Infectious Diseases*, vol. 24, 2017, pp. 335–341.

[5] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *arXiv preprint arXiv:1303.3997*, 2013.

[6] Broad Institute, "Genome Analysis Toolkit (GATK) 4," 2019. [Online]. Available: https://github.com/broadinstitute/gatk

[7] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012, pp. 2–2.

[8] Intel, "Infrastructure for deploying GATK best practices pipeline," 2016. [Online]. Available: https://www.intel.com/content/www/us/en/healthcare-it/solutions/documents/deploying-gatk-best-practices-paper.html

[9] Y.-T. Chen, J. Cong, J. Lei, and P. Wei, "A novel high-throughput acceleration engine for read alignment," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2015, pp. 199–202.

[10] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 199–213.

[11] J. Cong, L. Guo, P.-T. Huang, P. Wei, and T. Yu, "Smem++: A pipelined and time-multiplexed smem seeding accelerator for genome sequencing," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 210–2104.

[12] 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 *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 127–135.

[13] W. Tang, W. Wang, B. Duan, C. Zhang, G. Tan, P. Zhang, and N. Sun, "Accelerating millions of short reads mapping on a heterogeneous architecture with FPGA accelerator," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2012, pp. 184–187.

[14] J. Arram, K. H. Tsoi, W. Luk, and P. Jiang, "Reconfigurable acceleration of short read mapping," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2013, pp. 210–217.

[15] Intel, "Intel genomics kernel library," 2019. [Online]. Available: https://github.com/Intel-HLS/GKL

[16] N. Ahmed, V.-M. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars, "Heterogeneous hardware/software acceleration of the BWA-MEM DNA alignment algorithm," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2015, pp. 240–246.

[17] Y. Yamaguchi, H. K. Tsoi, and W. Luk, "FPGA-based Smith-Waterman algorithm: analysis and novel design," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2011, pp. 181–192.

[18] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, and P. Wei, "When Apache Spark meets FPGAs: A case study for next-generation DNA sequencing acceleration," in *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 64–70.

[19] M. Ito and M. Ohara, "A power-efficient FPGA accelerator: Systolic array with cache-coherent interface for Pair-HMM algorithm," in *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)*. IEEE, 2016, pp. 1–3.

[20] S. Huang, G. J. Manikandan, A. Ramachandran, K. Rupnow, W.-m. W. Hwu, and D. Chen, "Hardware acceleration of the Pair-HMM algorithm for DNA variant calling," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2017, pp. 275–284.

[21] J. Wang, X. Xie, and J. Cong, "Communication optimization on GPU: A case study of sequence alignment algorithms," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 72–81.

[22] Intel, "Accelerating genomics research with opencl and fpgas," 2017. [Online]. Available: https://www.intel.com/content/www/us/en/healthcare-it/solutions/documents/genomics-research-with-opencl-and-fpgas-paper.html

[23] W. Qiao, J. Du, Z. Fang, M. Lo, M. F. Chang, and J. Cong, "High-throughput lossless compression on tightly coupled CPU-FPGA platforms," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2018, pp. 37–44.

[24] J. Cong, Z. Fang, M. Huang, L. Wang, and D. Wu, "CPU-FPGA coscheduling for big data applications," *IEEE Design & Test*, vol. 35, no. 1, pp. 16–22, 2018.

[25] P. Zhou, Z. Ruan, Z. Fang, M. Shand, D. Roazen, and J. Cong, "Doppio: I/O-aware performance analysis, modeling and optimization for in-memory computing framework," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2018, pp. 22–32.

[26] Broad Institute, "Base quality score recalibration methods and algorithms," 2018. [Online]. Available: https://software.broadinstitute.org/gatk/documentation/article?id=11081

[27] Amazon, "Amazon EC2 F1 instance," 2019. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[28] Xilinx, "Vivado design user guide," p. 548, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf

[29] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Latte: Locality aware transformation for High-Level Synthesis," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 125–128.

[30] N. Cai, T. B. Bigdeli, W. Kretzschmar, Y. Li, J. Liang, L. Song, J. Hu, Q. Li, W. Jin, Z. Hu *et al.*, "Sparse whole-genome sequencing identifies two loci for major depressive disorder," vol. 523, no. 7562. Nature Publishing Group, 2015, p. 588.

[31] A. F. Gazdar, V. Kurvari, A. Virmani, L. Gollahon, M. Sakaguchi, M. Westerfield, D. Kodagoda, V. Stasny, H. T. Cunningham, I. I. Wistuba *et al.*, "Characterization of paired tumor and non-tumor cell lines established from patients with breast cancer." JOHN WILEY & SONS LTD, 1998.

[32] Illumina, 2019. [Online]. Available: https://support.illumina.com/downloads.html

[33] IGSR: The International Genome Sample Resource, "1000 genomes project data," 2019. [Online]. Available: https://www.internationalgenome.org/data

[34] K. S. Gautam, "Parallel histogram calculation for FPGA: Histogram calculation," in *2016 IEEE 6th International Conference on Advanced Computing (IACC)*. IEEE, 2016, pp. 774–777.

[35] L. Maggiani, C. Salvadori, M. Petracca, P. Pagano, and R. Saletti, "Reconfigurable architecture for computing histograms in real-time tailored to FPGA-based smart camera," in *2014 IEEE 23rd International Symposium on Industrial Electronics (ISIE)*. IEEE, 2014, pp. 1042–1046.

[36] A. Shahbahrami, J. Y. Hur, B. Juurlink, and S. Wong, "FPGA implementation of parallel histogram computation," in *2nd HiPEAC Workshop on Reconfigurable Computing*. Published, 2008, pp. 63–72.

[37] J. H. Ahn, M. Erez, and W. J. Dally, "Scatter-add in data parallel architectures," in *11th International Symposium on High-Performance Computer Architecture*. IEEE, 2005, pp. 132–142.

[38] M. Hosseinabady and J. L. Núñez-Yáñez, "Pipelined streaming computation of histogram in FPGA OpenCL," in *PARCO*, 2017, pp. 632–641.

[39] Xilinx, "UltraRAM: Breakthrough embedded memory integration on UltraScale+ devices," 2019. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf

[40] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2019, pp. 73–82.

[41] Xilinx, “Xilinx ML suite,” 2019. [Online]. Available: https://github.com/Xilinx/ml-suite

[42] M. Zhang, L. Li, H. Wang, Y. Liu, H. Qin, and W. Zhao, “Optimized compression for implementing convolutional neural networks on FPGA,” *Electronics*, vol. 8, no. 3, p. 295, 2019.

[43] X. Wei, Y. Liang, X. Li, C. H. Yu, P. Zhang, and J. Cong, “TGPA: tile-grained pipeline architecture for low latency CNN inference,” in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.