An FPGA Accelerator for Genome Variant Calling

Tiancheng Xu  
Rice University  
txu@rice.edu

Scott Rixner  
Rice University  
rixner@rice.edu

Alan L. Cox  
Rice University  
alc@rice.edu

Abstract—In genome analysis, it is often important to identify variants from a reference genome. However, identifying variants that occur with low frequency can be challenging, as it is computationally intensive to do so accurately. LoFreq is a widely used program that is adept at identifying low frequency variants. This paper presents an FPGA-based accelerator for LoFreq. In particular, this accelerator is targeted at virus analysis, which is particularly challenging, compared to human genome analysis, as the characteristics of the data to be analyzed are fundamentally different. This accelerator can achieve up to 120× speedups on the core computation of LoFreq and speedups of up to 32.4× across the entire program.

I. INTRODUCTION

Genome analysis has become an important computing workload as we work towards personalized medicine, better understanding diseases, and other basic scientific inquiry. One important aspect of genome analysis is variant calling. Variant calling is the process of identifying variants from a reference genome in genetic data. A typical pipeline consists of the following three stages. First, genomes are read by a sequencer to collect raw snippets of sequence data (called “reads”). Second, the reads are aligned and mapped to a reference genome (called “read mapping”). Finally, differences between the reads and reference genome are examined and variants are identified (called “variant calling”). Note that this is not as trivial as looking for differences, because it involves distinguishing between sequence read errors, read mapping errors, and true genome variations (“variants”).

LoFreq is an alignment-based variant caller that can accurately detect very rarely occurring variants [27], [31]. In particular, LoFreq accurately distinguishes between low frequency variants and errors in sequencing or mapping using rigorous statistical modeling. Unfortunately, LoFreq’s effectiveness at detecting low frequency variants comes at a performance cost. LoFreq is slower than other variant callers. Often, iVar [18] is used instead of LoFreq, as it is faster. However, it is less sensitive, so it may miss low frequency variants.

Despite its performance disadvantage, LoFreq’s sensitivity can be invaluable. For example, since the outbreak of COVID-19, LoFreq has been heavily used to track inter-host variants and the evolutionary dynamics of SARS-CoV-2 [22]. It is therefore important to improve the overall performance of LoFreq in order to enable detection of low frequency variants to further biological progress, understanding, and innovation.

The shape of the genomic dataset influences the available parallelism within the core LoFreq algorithm. There are three important parameters that characterize a dataset upon which variant calling is performed. The first is the length of the genome. All of the reads corresponding to a base (nucleotide) in the reference genome form a column. Each column can be processed independently, providing one source of parallelism. The second parameter is the depth of each column, which is the number of bases in that column. Each column may have a different number of bases in it, as the reads will not be mapped uniformly across the genome. The last parameter is the number of bases that are different from the reference base within a column. This parameter will also vary by column. The computational workload within a column is proportional to the product of the last two parameters. Unfortunately, compared to parallelization across columns, parallelization within a column is more challenging because of data dependencies that are inherent to the algorithm.

This paper presents an FPGA-based accelerator for the LoFreq variant caller. While the LoFreq algorithm is the same regardless of the parameters described above, the accelerator design is driven by the characteristics of virus data, which has relatively short genome lengths but large and varying depths. This is one important case in which the available parallelism is more difficult to exploit and is well suited to custom hardware acceleration. The accelerator performs the core probability calculations of LoFreq in order to identify variants. The accelerator design consists of several column units. Each column unit is designed to process a single column of data at a time. The column units make use of prefetching, pipelining, and parallelization to efficiently identify variants in that column. LoFreq processes every column independently, so once a column unit completes the computation for one column, it can begin processing another column. Furthermore, multiple column units can operate on different columns independently and in parallel.

Each column unit consists of multiple processing elements that operate on different portions of the computation within the column simultaneously. As LoFreq deals with very small probabilities and is trying to detect variants that occur with low frequency, these processing elements operate on very small numbers that need high precision. Therefore, all operations use double precision floating point arithmetic and all computations are done in log-space to avoid floating point underflow. This means that the key computations within a processing element are logarithms and exponentials. These computations
are of great significance but difficult to identify because
frequency from errors in such large columns. These SNVs
depths as high as 1,000,000, compared to that of human
SARS-CoV-2 genome data poses a unique challenge. Study
distinguishing between SNVs and errors. Variant calling on
probability that the base is erroneous can be computed directly
sequence quality and the mapping quality of that read. The
is associated with a quality score that is computed from the
a true genome variation, a Single Nucleotide Variant
from the previous stages (sequencing or read mapping), or
in the column. Such a varying base could either be an error
errors due to the nature of sequencing. The error rate of the
sequencer is generally well known.

The next step is to perform read mapping. Read mapping
is the process that maps these reads to a reference genome
in order to determine where the short read came from in
the longer DNA sequence. Note that again, reads may be
incorrectly mapped to the reference genome, as there are both
potential errors in the read from the sequencer and potential
mutations from the reference in the read fragment. Once all
of the reads are aligned and mapped to the reference genome,
every position in the reference genome will be covered by
many reads. At a given position, genome bases (nucleotides)
from all reads that cover this position form a column of
genome bases. As stated in the previous section, such a column
of bases is referred to as a column, and the total number of
bases in a column is known as the depth. The depth of a
column is denoted by \( N \).

Within a column, there can exist bases that differ from the
corresponding reference base and the majority of other bases
in the column. Such a varying base could either be an error
from the previous stages (sequencing or read mapping), or
a true genome variation, a Single Nucleotide Variant (SNV),
that is of significant interest. Therefore, each base in a column
is associated with a quality score that is computed from the
sequence quality and the mapping quality of that read. The
probability that the base is erroneous can be computed directly
from the quality score.

Variant callers take aligned sequences and their quality
scores as input and attempt to identify variants in the data,
distingushing between SNVs and errors. Variant calling on
SARS-CoV-2 genome data poses a unique challenge. Study
of the SARS-CoV-2 genome has much deeper columns, with
depths as high as 1,000,000, compared to that of human
genome data which typically have depths from 30 to 50. A
major challenge is to distinguish SNVs with extremely low
frequency from errors in such large columns. These SNVs
are of great significance but difficult to identify because
sequencing machines and read mappers produce errors at a
similarly low frequency.

LoFreq is a variant caller specialized in solving this chal-
lenge. LoFreq can accurately distinguish low frequency SNVs
from sequencing and mapping errors by virtue of its unique
and rigorous statistical modeling. It examines each column in
the alignment independently. For each column, LoFreq models
errors in that column using a Poisson-Binomial distribution. If
the number of varying bases is inconsistent with the computed
distribution, then SNVs most likely exist.

III. LOFREQ AND ITS COMPUTATION

As previously mentioned, LoFreq is able to identify SNVs
even when their frequency of occurrence is as low as that of
sequencing and mapping errors. However, this makes LoFreq
much slower than other variant callers, as it must do more
computation to identify low frequency variants.

As input, LoFreq takes a file of reads that have already been
mapped to a reference sequence along with quality scores,
which are a measure of confidence that both the read and the
mapping are correct. Before processing each column, LoFreq
initially must perform preprocessing on this data to do three
things. First, the entire column must be extracted from the
collection of mapped reads. This is non-trivial, as it involves
scanning a region of the file to find all of the bases that belong
in that column. Second, the error probability for each base
must be calculated from its associated quality score. Finally,
the number of observed varying bases from the reference
genome in the column must be counted.

After preparing the column, the core probability calculation
uses the depth of column \( N \), the number of observed varying
bases in the column \( K \), and the error probabilities to decide
whether there are SNVs among the observed varying bases.

A. Core Computation

The core of LoFreq operates on each column independently
of all other columns. It models sequencing and mapping errors
of an individual column in a Poisson-Binomial distribution.
The intuition is that if the number of observed varying bases
is not consistent with the error distribution, then chances are
high that SNVs exist; otherwise, these varying bases are most
likely sequencing or mapping errors.

In a column, each individual base is modeled as an inde-
pendent Bernoulli trial. The error probability of the \( n^{th} \) base,
\( p_n \), indicates the probability that the base is a sequencing
or mapping error. Using this model, the algorithm calculates
the p-value to confirm or refute the null hypothesis that all
observed varying bases in the column are errors. A predeter-
mined threshold, \( t \), is used to determine whether or not the null
hypothesis is true. If the calculated p-value is greater than \( t \),
then the null hypothesis is not wrong and all observed varying
bases are most likely errors. If the calculated p-value is less
than or equal to \( t \) then the null hypothesis is likely incorrect;
therefore there is strong evidence for the existence of SNVs.

Consider the case where \( K \) varying bases are observed in
a column which has \( N \) bases in total. \( Pr_n(X = k) \) is the
The core of LoFreq is shown in Algorithm 1. The algorithm follows directly from the previously explained mathematics. In particular, equations (3), (4), and (5) are used to ultimately compute the p-value.

For each base, $1 \leq n \leq N$, the algorithm iteratively computes part (0 $\leq k < K$) of the PMF based on equation (3) (lines 12 to 14). Before computing the PMF, it first reads the error probability, $p_n$, for the current iteration (line 5),
Algorithm 1: Probability Calculation Algorithm.

**Input:** Error Probability array \( \text{Err} \_\text{arr} \), column depth \( N \), number of varying bases \( K \).

**Result:** Probability mass function when \( n = N \).

1. Allocate \( \ PL[K] \); // \( PL_n(X=k) \) for \( k \) from 0 to \( K-1 \)
2. Allocate \( \ PL\_\text{prev}[K] \); // \( PL_{n-1}(X=k) \) for \( k \) from 0 to \( K-1 \)
3. \( PL\_\text{prev}[0] = 0; \) // \( \ln(1) \)
4. for \( n \) from 1 to \( N \) do
   5. \( p_n = \text{Err} \_\text{arr}[n]; \)
   6. \( \ln\_pm = \ln(p_n); \)
   7. \( \ln\_1\_pm = \ln(1.0 - p_n); \)
   8. if \( n < K \) then
      9. \( \quad \ PL\_\text{prev}[n] = -1e100; \) // approx. \( \ln(0) \)
   10. end
   11. bound = \( (n < (K-1)) \) ? \( n : (K-1); \)
   12. for \( k \) from 1 to \( \text{bound} \) do
      13. \( \quad PL[k] = \log\_\text{sum}\_\text{exp}(PL\_\text{prev}[k] + \ln\_1\_pm, \)
      14. \( \quad \quad PL\_\text{prev}[k] = \ln\_pm); \)
   15. end
   16. \( \quad PL[0] = \text{PL}\_\text{prev}[0] + \ln\_1\_pm; \)
   17. if \( n == K \) then
      18. \( \quad \ln\_\text{pval} = \text{PL}\_\text{prev}[K-1] + \ln\_pm; \)
   19. else if \( n > K \) then
      19. \( \quad \ln\_\text{pval} = \log\_\text{sum}\_\text{exp}(\ln\_\text{pval}\_\text{prev}, \)
      20. \( \quad \quad \text{PL}\_\text{prev}[K-1] + \ln\_pm); \)
   20. end
   21. \( \quad \text{PL}\_\text{prev} = \text{PL}; \)
   22. // moves data from \( PL \) to \( \text{PL}\_\text{prev} \);
   23. \( \ln\_\text{pval}\_\text{prev} = \ln\_\text{pval}; \)
24. return \( PL, \ln\_\text{pval}; \)

and computes the logarithm of both the error rate and the accuracy (lines 6 and 7). The p-value is then computed based on equation (5) (lines 16-20). At the end of each iteration, the current values are stored in their \( \text{prev} \) counterparts in preparation for the subsequent iteration.

B. Computation Characteristics

Table I shows the execution time breakdown of LoFreq on several SARS-CoV-2 datasets. This data is collected on an AMD Ryzen 7 5800X CPU using a single thread within a single process. This table illustrates two important characteristics of LoFreq on important, real-world datasets. First, processing can take quite a long time (over 41 hours on SRR12380204). Second, over 90% of the processing time takes place in the core probability calculations (Algorithm 1).

Fortunately, there are multiple sources of potential parallelism in the workload. The first is that different inner loop iterations (shown in lines 12 to 14 in Algorithm 1) can be fully parallelized (intra-column parallelism). The challenge here is that intra-column parallelism is irregular because the amount of parallelism depends on the parameter \( K \), which varies among columns.

Second, different columns within a dataset can be processed in parallel (inter-column parallelism). Different instances of Algorithm 1 can be launched to process columns in parallel. For each SARS-CoV-2 alignment dataset, for example, the total number of columns is fixed (29,903), while the computation workload of each column can vary drastically. The reason is that the parameters \( N \) and \( K \) can vary drastically among columns.

Moreover, the three different types of operations in an outer loop iteration, i.e., loading \( p_n \), calculating the logarithm of \( pn \), and the inner loop, can be pipelined.

Due to the nature of the problem, all input and intermediate data are represented in double-precision floating point. On an FPGA, computations on double-precision are less efficient if not carefully tuned. In LoFreq, the fundamental operations are logarithm and exponential. The key computation \( \log\_\text{sum}\_\text{exp} \) includes serialized exponential and logarithm operations, which has a long critical path latency.

The key input error rate array can take a non-trivial amount of memory to store. For example, it requires 7.63 MB of memory when \( N \) is 1,000,000. This makes it impractical to store all of the input data on-chip. The state-of-the-art implementation of LoFreq has used CPU multi-processing to accelerate the computation. However, LoFreq is still slow. As an example, when LoFreq is multi-processed using 16 CPU threads, the end-to-end running time on the SRR12380204 dataset is still 6.1 hours long.

IV. ACCELERATOR DESIGN AND IMPLEMENTATION

The FPGA accelerator performs the core probability computation, Algorithm 1, of LoFreq, while the rest of the application remains in software. The FPGA implementation’s output is identical to the software version. The accelerator is implemented using the high level synthesis tools in the Xilinx Vitis Development Platform 2020.2. This section introduces the design of the FPGA LoFreq accelerator.

A. Column Units

The FPGA accelerator is composed of multiple column units (CUs). Each column unit operates independently on a single column at a time. Multiple CUs can process multiple columns in parallel. Figure 2 shows the design of a CU. The general operation of the CU is as follows:

1) The software dispatches a column computation to the column unit.
2) The controller initiates a loop over the elements of the column (corresponding to the outer loop in Algorithm 1 on line 4) after prefetching the first two error probabilities ($p_1$ and $p_2$) and computing the initial logarithms ($\ln(p_1)$ and $\ln(1-p_1)$):
   a) The error probability, $p_{n+2}$, is prefetched.
   b) The logarithms of $p_{n+1}$ and $1-p_{n+1}$ are computed.
   c) The previously computed logarithms on $p_n$ ($\ln(p_n)$ and $\ln(1-p_n)$ in the algorithm) are used for this iteration.
   d) The main processing elements (PE) will compute $PL_n(X = k)$ for all $k < K$ (lines 12–14 in Algorithm 1). Multiple PEs operate on different data in both a parallel and pipelined manner. Each PE can initiate a new computation each cycle.

3) The column unit returns the results to the software.

This process is continually repeated until all of the columns have been processed.

As previously mentioned, the column data is large, so it cannot be stored on the FPGA chip. The error probabilities are stored in DRAM on the FPGA board and must be prefetched in order to keep up with the computation. By storing all of the column data in DRAM, the design is decoupled from the on-chip memory that is needed, as will be discussed in Section IV-B.

The most expensive operations within the computation are the logarithms and the exponentials. A dedicated logarithm unit is used to compute the logarithms of $p_{n+1}$ and $1-p_{n+1}$ during iteration $n$. In order for the error probability to be available for these logarithm calculations, the prefetcher loads $p_{n+2}$ during iteration $n$. This forms a three-stage pipeline at the macro level of the column unit in which the first stage fetches $p_n$, the second stage computes the logarithms of $p_n$ and $1-p_n$, and the third stage performs the column’s probability calculations to compute the resulting partial PMF and p-value.

<table>
<thead>
<tr>
<th>TABLE II. The latency and resource use of arithmetic units.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating Point Operators</td>
</tr>
<tr>
<td>--------------------------</td>
</tr>
<tr>
<td>add / sub</td>
</tr>
<tr>
<td>exp</td>
</tr>
<tr>
<td>log</td>
</tr>
</tbody>
</table>

B. Processing Elements

The core computation of each CU is the computation of $PL_n(X=k)$. Multiple processing elements (PEs) within the CU compute $PL_n(X=k)$ for different values of $k$ in parallel. Furthermore, each PE is pipelined, allowing it to initiate a new computation every cycle.

The column unit does not have dedicated hardware to compute $SL_n$. Instead, the PEs are used for that purpose. Once the PEs have computed all $K-1$ values of $PL_n$ in step 2d a final computation is issued to a PE to compute $SL_n$. Note the similarities between equations (3) and (5). $SL_n$ is actually stored in the $PL_n$ array in the $K^{th}$ location. This allows the initial mux to select whether to pass through just $PL_{prev}[k]$ (which is $SL_{n-1}$ when $k = K$) or $PL_{prev}[k] + \ln(1-p_n)$ (which is $PL_{n-1}(X=k)+\ln(1-p_n)$ when $k < K$). A simple adder (not shown) handles the computation of $SL_K$ during the $K^{th}$ iteration of the outer loop.

The primary computation of the PE is the log_sum_exp calculation of equation 4. Logarithms and exponentials on double precision floating point numbers are expensive both in terms of resources and latency. Therefore, the PEs dictate the overall performance of the accelerator. This requires careful tuning of the floating point operation units. It is critical to reduce the latency of the floating point operators while not negatively affecting timing. Also, because the floating point operators consume a non-trivial amount of LUTs and DSPs, the use of resources needs to be tuned as well. Over-utilizing either the LUTs or the DSPs will not only under-utilize the other, but also limit the number of PEs and CUs that can eventually be implemented. We use the floating-point operation implementations in Xilinx LogiCORE IPs (v7.1) [3]. The latency and resource use of floating-point operators are carefully tuned for each operation in log_sum_exp to optimize overall performance and efficiency, as shown in Table II.

The on-chip memory is partitioned based on the number of PEs so that all PEs can access data without competing for memory ports. Note that the amount of on-chip storage that is required is $2K$, as at any given time, the PEs are reading $PL_{n-1}$ and writing $PL_n$. Even when $K$ is 65536, which is unusually large, the memory required is only 1 MB. This memory is divided into two buffers. During each iteration of the outer loop, the PEs will read from one buffer and write to the other. For the next iteration, the input and output buffers are swapped.

C. Design Trade-offs

There is a trade-off between exploiting intra-column and inter-column parallelization to maximize performance across a variety of datasets. More column units can exploit higher levels of inter-column parallelism, whereas more PEs per column
unit can exploit higher levels of intra-column parallelism. The more PEs per column unit, the fewer column units will fit in the finite resources of an FPGA.

The amount of intra-column parallelism is limited by $K$ for each column. Therefore, there are diminishing returns to increasing the number of PEs in a column unit. However, the overhead of the column unit beyond the PEs makes it such that there is a benefit to exploiting intra-column parallelism to some degree. Therefore, it is critical to perform a design space analysis to determine the best design point.

V. HOST SYSTEM DESIGN AND IMPLEMENTATION

LoFreq is written in standard C as a single-threaded program. In order for LoFreq to utilize our Xilinx Alveo U250-based accelerator, we had to change the LoFreq source code that runs on the host.

Under the Xilinx Vitis Environment, the accelerator is presented to the host system as an OpenCL device. So, our changes to the LoFreq source code were to (1) use the OpenCL API to initialize this device so that our accelerator for the core probability computation on a column could be invoked as an OpenCL kernel and (2) replace calls to the function that performs the core probability computation on a column on the host processor with invocations of that OpenCL kernel. Since the ordinary function calls replaced are synchronous, i.e., they do not return until the function has completed, we invoke the OpenCL kernel synchronously. In other words, after enqueuing commands to the OpenCL runtime that transfer the inputs to the U250’s DRAM, execute the kernel, and transfer the results back to the host, the host waits for the results to be returned before starting any work on the next column.

Once the core probability computation is accelerated by the U250, the parts of the computation that remain on the host processor, such as preprocessing, will become a performance bottleneck if they are executed sequentially. To parallelize all parts of LoFreq’s execution, its developers used multiple processes. However, rather than modifying the LoFreq source code to implement multiprocess execution, they created a Python script that divides the dataset into chunks, runs the unmodified (single-threaded) LoFreq program on each chunk in parallel, and merges the results from each of the processes at the end. We directly use that Python script for multiprocessing in our system: each process works on one chunk and all processes run concurrently. This works fine because the Xilinx OpenCL runtime system allows multiple processes to concurrently execute multiple kernels on the FPGA device. Specifically, the OpenCL command queue allows the columns submitted by different processes to be processed in parallel on different column units. The OpenCL runtime automatically distributes the execution of the columns over the CUs as the CUs become available.

As discussed in Section III-B, the time that it takes to process different columns varies widely, depending on the values of $N$ and $K$ for a given column. Dividing the dataset into chunks, i.e., groups of consecutive columns, achieves a good balance between two competing factors: (1) minimizing operating system overheads, such as process creation and continual context switching between processes, and (2) minimizing workload imbalance between the processes. We find that dividing the datasets into many more chunks than the number of processor cores or hardware thread contexts yields the best performance results. However, at the operating system level, we observe a subtle difference between the multiprocess LoFreq and our accelerated implementation that affects the trade-off between these competing factors. Processes in our accelerated implementation are rarely preempted by the operating system because their scheduling quantum has expired, instead they are voluntarily relinquishing the processor when they wait for the completion of an OpenCL kernel on a column. Consequently, the number of context switches is primarily a function of the number of columns, and unrelated to the number of chunks, so increasing the number of chunks, and thus the number of processes, to achieve better load balance does not significantly increase the context switching overhead. Moreover, the OpenCL command queue decouples these processes from the hardware column units, allowing each process to enqueue columns that will be serviced as column units become available, making it practical to use more processes than there are hardware column units.

VI. EVALUATION

A. Experimental Setup

System Configuration. The system used for evaluating our accelerator design consists of an AMD Ryzen 7 5800X processor (with 8 cores and 16 hardware thread contexts), 128 GB of DDR4 3200 memory, a 1 TB Samsung 980 PRO SSD, and a Xilinx Alveo U250 card (Platform name: xilinx_u250_gen3x16_xdma_3_1) [9]. The system was running Ubuntu 18.04.4 LTS with Linux kernel 5.4.0 for compatibility with the Xilinx kernel module.

The host program was developed in OpenCL 1.2 with Xilinx XRT extensions, and the FPGA design was written as an OpenCL kernel in C. The high-level synthesis and hardware implementation were done using Xilinx Vitis 2020.02, which under the hood uses Xilinx Vivado 2020.02 for logic optimization, placement, and routing. All of the evaluated FPGA designs were implemented to run at 300 MHz. Aggressive optimization strategies were applied in the implementation: the ExtratimeOpt strategy is used for placement and the AggressiveExplo strategy is used for routing [8]. Furthermore, physical optimizations were enabled. Individual column units were implemented within one Super Logic Region [10] in order to minimize boundary crossings.

Datasets. Table III describes the real-world datasets used throughout this evaluation. These datasets come from the NCBI SARS-CoV-2 data repository [5] and recent clinical studies [14], [17].

Baseline. The stock multi-process implementation of LoFreq [7] is used as the baseline in our performance evaluation. It is configured to divide the dataset into 112 chunks,
which effectively utilizes the 8 cores and 16 hardware thread contexts provided by the AMD Ryzen 7 5800X processor.

### B. Probability Calculation Speedup Results

This section presents speedup results for just the probability calculation on a column, which is the core computation that is accelerated by the FPGA. The computational cost of this workload is dictated by the column’s values for \( N \) and \( K \). \( N \) determines the number of outer loop (starting at line 4 in Algorithm 1) iterations and \( K \) determines the number of inner loop (starting at line 12 in Algorithm 1) iterations.

We measure the speedup of one single column unit over one single CPU core, on processing columns from the datasets shown in Table III. One individual column unit from each of the four designs shown in Table IV is evaluated. Each column unit is implemented with a different number of PEs, so they vary in their resource utilization. All four column unit designs run at 300 MHz.

Figure 3 shows the speedup results. Each mark in the figure represents one column. The Y axis value is the speedup on the core probability calculation when processing that column. The X axis value of a mark is the column’s \( K \) value. Only \( K \) is shown, while \( N \) is not, since \( K \) largely determines the speedup. This is because \( K \) determines the number of iterations of the inner loop, and most of the speedup comes from the array of PEs parallelizing the computation.

To clearly show the speedup on columns with the most common values of \( K \), only the results for columns whose \( K \) ranges from 1 to 1500 are shown. In reality, the speedup is even larger on columns whose \( K \) is larger than 1500. The highest observed speedup is 121.7 \times when \( K \) is 55208, using one column unit with 32 PEs.

On dataset SRR12380204, using one column unit with 32 PEs, the time spent on the core probability calculation is reduced from 40.01 hours to 1.32 hours. And, the total execution time for the entire application is reduced to 2.96 hours, which is a 14.07 \times speedup compared to a sequential (single-process) execution of the baseline. Note that with this column unit the core probability calculation no longer constitutes the majority of the application’s execution time, showing why the rest of the application must still be parallelized on the host processor, as discussed in Section V.

### C. Complete Designs

While a single column unit with 32 PEs already delivers significant speedup on the core probability calculation, such a CU is not necessarily the optimal design. Furthermore, the FPGA has enough resources for multiple such CUs. As discussed in Section IV-C, a key tradeoff in the FPGA design space is between exploiting inter-column and intra-column parallelism. In the FPGA design, that question is transformed into what is the right number of PEs per column unit and the total number of column units in order to maximize the performance gain.

Table IV shows four representative designs, each exploiting both inter-column and intra-column parallelism at a different level. Each design has a different number of PEs in the column unit, and the maximum number of column units are implemented. In other words, there are insufficient resources on the FPGA to fit another CU in each design. Only CUs with a power-of-2 number of PEs are considered. The overhead of controlling the use of the PEs is prohibitive for other designs, so using a power-of-2 leads to the most efficient use of resources within a CU.

To unleash the full performance of each FPGA design, we evaluate these designs using the multi-process LoFreq described in Section V. A fixed number of chunks is chosen for each design: 72 (designs 1 and 2), 112 (design 3), 160 (design 4). It is unsurprising that more chunks are needed as the number of CUs increases, as it is important to keep all CUs busy throughout the entirety of the computation. As the columns within a chunk are processed sequentially, having more chunks makes it more likely that the simple chunk-based load balancing will be successful through the end of the computation.

Figure 4 shows the speedup of the four FPGA designs (described in Table IV) over the multi-process CPU baseline (described in VI-A). As the figure shows, designs 3 and 4 consistently outperform the other designs and speed up the entire application by over 20 \times for most computationally intense datasets. Furthermore, on dataset SRR12380204, for example, the execution time is reduced from 2.96 hours with a single column unit (with 32 PEs) to 0.22 hours with the 14 column units (each with 8 PEs) in design 3.

Designs 3 and 4 achieve the best performance because they both strike a balance between intra-column and inter-column parallelism. On the one hand, the results from Figure 3 show that it is beneficial to have a larger number of column units; as the number of PEs doubles in a column unit, the speedup increases by less than 2 \times. On the other hand, column units in both designs still deliver over 10 \times speedup for columns whose \( K \) is larger than 600.

For all datasets, we have verified that the end-to-end results from our CPU-FPGA system are identical to those produced by the original LoFreq, including the SNVs called and the quality scores that are associated with those SNVs.

The resource utilization of the different designs is shown in Table IV. The individual CU usage shows the percentage of resources used just for a column unit with the given number of...
Nvidia Clara Parabricks is a software suite of common Whole-Genome Sequencing analysis tools that are optimized for running on high-end Nvidia GPUs [6]. Recently, LoFreq was incorporated as part of the pipeline, but it is limited to variant calling on somatic human genome data only. Somatic human genome data has different characteristics from viral data. There are up to 5 orders of magnitude more columns, but each column is up to 4 orders of magnitude smaller. This makes it such that there is limited intra-column parallelism, so Parabricks focuses on mapping the inter-column parallelism to the GPU. Parabricks is able to achieve a 6× speedup over LoFreq on such data using 4 Nvidia V100 GPUs [4].

VIII. CONCLUSION

This paper has presented the design of an FPGA-based accelerator for the LoFreq variant caller that can achieve up to 32.4× speedup on the end-to-end execution of LoFreq on real SARS-CoV-2 datasets. This speedup is achieved over the state-of-the-art parallelized software version of LoFreq that efficiently utilizes 16 hardware threads. This paper has also presented a design space analysis of the accelerator that shows that a single column unit can speed up the core computation by up to 120×, but that there is a trade-off between inter- and intra-column parallelism. This accelerator has important applications to real world genomic analysis as LoFreq excels at identifying low frequency variants and is widely used, but suffers from long execution times.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their constructive feedback. We thank Dr. Todd Treangen, Bryce Kille, Yunxi Liu for their consistent help. This work is partially supported by the NSF under grant NSF-CNS2008857 and the Ken Kennedy Institute Computational Science & Engineering Recruiting Fellowship (funded by the Rice Oil & Gas HPC Conference).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Abbreviation</th>
<th>N (Depth)</th>
<th>K (Number of Varying Bases)</th>
<th>Exec Time (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Mean</td>
<td>Std. Dev.</td>
<td>Mean</td>
</tr>
<tr>
<td>SRR11177792 [5]</td>
<td>SRR111</td>
<td>11.35%</td>
<td>5.52%</td>
<td>3.81%</td>
</tr>
<tr>
<td>SRR12380204 [5]</td>
<td>SRR123</td>
<td>49.11%</td>
<td>24.47%</td>
<td>89.03%</td>
</tr>
<tr>
<td>COVHA-20200314-P8-F03-P [14]</td>
<td>P8-F03</td>
<td>11.35%</td>
<td>5.52%</td>
<td>3.81%</td>
</tr>
<tr>
<td>COVHA-20200316-P11-F06-P [14]</td>
<td>P11-F06</td>
<td>49.11%</td>
<td>24.47%</td>
<td>89.03%</td>
</tr>
<tr>
<td>COVHA-20200316-P12-F07-P [14]</td>
<td>P12-F07</td>
<td>11.35%</td>
<td>5.52%</td>
<td>3.81%</td>
</tr>
<tr>
<td>COVHA-20200403-P1-B03-P [14]</td>
<td>P1-B03</td>
<td>49.11%</td>
<td>24.47%</td>
<td>89.03%</td>
</tr>
<tr>
<td>H2HTFD5SV-2-IDUD6034</td>
<td>S20_L002</td>
<td>49.11%</td>
<td>24.47%</td>
<td>89.03%</td>
</tr>
</tbody>
</table>

TABLE IV. The hardware configurations and total resource usage for different FPGA designs.


References