An FPGA Accelerator for Genome Variant Calling

TIANCHENG XU, SCOTT RIXNER, and ALAN L. COX, Rice University, USA

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 article presents a design framework for 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. Across the design space, this accelerator can achieve up to 120× speedups on the core computation of LoFreq and speedups of up to 51.7× across the entire program.

CCS Concepts: • **Computer systems organization → Reconfigurable computing**; **High-level language architectures;**

Additional Key Words and Phrases: HLS, FPGA, Variant Calling

ACM Reference format:
Tiancheng Xu, Scott Rixner, and Alan L. Cox. 2023. An FPGA Accelerator for Genome Variant Calling. *ACM Trans. Reconfig. Technol. Syst.* 16, 4, Article 53 (September 2023), 29 pages.
https://doi.org/10.1145/3595297

1 INTRODUCTION

Genome analysis has become an important computational 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 [30, 35]. In particular, LoFreq accurately distinguishes between low-frequency variants and errors in sequencing or mapping using rigorous statistical modeling. Unfortunately, LoFreq's

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).
Authors' address: T. Xu, S. Rixner, and A. L. Cox, Rice University, 6100 Main St, Houston, TX, 77005; emails: {txu, rixner, alc}@rice.edu.

1936-7406/2023/09-ART53 $15.00
https://doi.org/10.1145/3595297

effectiveness at detecting low-frequency variants comes at a performance cost. LoFreq is slower than other variant callers. Often, iVar [20] 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 [25]. It is therefore important to improve the overall performance of LoFreq 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 article 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 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 expensive in terms of both latency and resource use. The processing element design is optimized to take into account these long latency operations.

Using high-level synthesis, this article performs a design space analysis of the accelerator architecture along multiple dimensions, including parallelism across columns and processing elements, pipelining within the column units, and storage of intermediate results. This highlights the various tradeoffs across the design space and shows how different designs behave on different datasets. A column unit with 32 processing elements can speed up the core computation of LoFreq by up to 120× over the software version. Furthermore, the best overall accelerator design is able to speed up the entire application by 10.0–51.7× compared to a parallelized software version of LoFreq that utilizes 16 hardware CPU threads.

2 GENOMICS ANALYSIS

The first step in a genomics analysis pipeline is sequencing. Short genome fragments are read using a sequencer. As previously stated, these fragments are known as *reads*. They can be from anywhere within the genome and may contain 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 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, distinguishing 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 challenge. 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.

3 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.

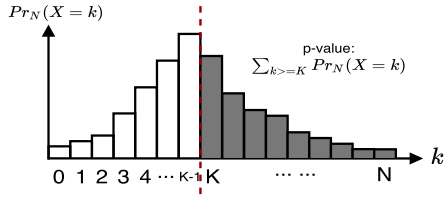


Fig. 1. Probability mass function.

3.1 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 independent Bernoulli trial. The error probability of the nth 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 predetermined 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 that has N bases in total. $Pr_n(X = k)$ is the probability of observing **exactly** k varying bases in the **first** n bases of this column. Figure 1 shows the **probability mass function (PMF)** of $Pr_N(X = k)$ with respect to k. $Pr_N(X = k)$ is the probability of observing exactly k varying bases in the entire column. Each bar in the figure is the probability that there are exactly k sequencing or mapping errors in the column. The p-value from this PMF is its tail sum, which is the sum of the shaded bars in the figure. In this case, $\sum_{k>=K} Pr_N(X = k)$. This p-value is the probability that there are K or more errors in the column, which is used to decide whether there are SNVs in the column.

Therefore, to compute the p-value, $Pr_N(X = k)$ must be calculated for all k. $Pr_n(X = k)$ can be computed from Pr_{n-1}, based on the following cases:

(1) k varying bases are observed in the first $(n-1)$ bases, and the nth base is not a varying base;
(2) $(k-1)$ varying bases are observed in the first $(n-1)$ bases, and the nth base is a varying base.

$Pr_n(X = k)$ is the sum of the probability of both cases. Given the probability that base n is erroneous, p_n, this can be expressed mathematically as follows:

$$\begin{aligned} o_1(n,k) &= Pr_{n-1}(X = k) \times (1 - p_n) \\ o_2(n,k) &= Pr_{n-1}(X = k-1) \times p_n \\ Pr_n(X = k) &= \begin{cases} o_1(n,k) & \text{if } k = 0 \\ o_1(n,k) + o_2(n,k) & \text{if } k > 0. \end{cases} \end{aligned} \tag{1}$$

The function o_1 computes the probability of the first case above being true and the function o_2 computes the probability of the second case above being true. Further note that $Pr_0(X = 0)$ is initialized to 1, as it is always true that there are exactly 0 varying bases in the first 0 bases of

the column. With these initializations, this equation can naively be iteratively used to calculate $Pr_n(X = k)$ for all k and for all n up to N. However, the original LoFreq paper proposed an optimization that is mathematically equivalent. The p-value can be computed based on the formula below:

$$S_n = \begin{cases} Pr_{n-1}(X = K-1) \times p_n & \text{if } n = K \\ S_{n-1} + Pr_{n-1}(X = K-1) \times p_n & \text{if } n > K. \end{cases} \tag{2}$$

Here, S_N is the p-value used to decide whether there are SNVs. Intuitively, S_n is the probability that there are K or more errors in the first n bases. This is calculated as the sum of the probability that there were K or more errors in the first $n-1$ bases plus the probability that there were *exactly* $K-1$ errors in the first $n-1$ bases and the nth base is an error. Therefore, S_N is the probability that there are K or more errors in the column, which is equivalent to the tail sum of the PMF. Note that this optimization means that Equation (1) would only need to be used to compute Pr_{n-1} for k values between 0 and $K-1$ to compute S_n.

An important property of the problem is that the error rate and the probabilities can be extremely small numbers. Directly applying Equations (1) and (6) could lead to floating-point underflow, even when using double precision floating-point numbers. Therefore, to prevent underflow and maintain numerical precision, the log of all of the probabilities are used in all calculations.

Thus, Equation (1) can be transformed to the form below (note that $PL_n(X = k)$ is the natural logarithm of $Pr_n(X = k)$):

$$\begin{aligned} ol_1(n,k) &= PL_{n-1}(X = k) + ln(1 - p_n) \\ ol_2(n,k) &= PL_{n-1}(X = k-1) + ln(p_n) \\ PL_n(X = k) &= \begin{cases} ol_1(n,k) & \text{if } k = 0 \\ log\_sum\_exp(ol_1(n,k), ol_2(n,k)) & \text{if } k > 0. \end{cases} \end{aligned} \tag{3}$$

Again, $PL_0(X = 0)$ is initialized to $ln(1)$. Recall that multiplication is simply addition in log space. Addition is performed by `log_sum_exp` in log space. This could be performed as follows: $ln(exp(a) + exp(b))$. However, the exponential calculations could overflow and this requires two exponential and one logarithm operation [13]. Therefore, the following definition of `log_sum_exp` is used instead:

$$log\_sum\_exp(a,b) = \begin{cases} a + ln(1.0 + exp(b-a)) & \text{if } a > b \\ b + ln(1.0 + exp(a-b)) & \text{if } a \le b. \end{cases} \tag{4}$$

This eliminates one of the exponential operations and only exponentiates a number that is smaller than the maximum of the inputs.

The optimized p-value computations can be computed in log space as follows:

$$SL_n = \begin{cases} PL_{n-1}(X = K-1) + ln(p_n) & \text{if } n = K \\ log\_sum\_exp(SL_{n-1}, PL_{n-1}(X = K-1) + ln(p_n)) & \text{if } n > K. \end{cases} \tag{5}$$

The core of LoFreq is shown in Algorithm 1. The algorithm follows directly from the previously explained mathematics. In particular, Equations (3)–(5) are used to ultimately compute the p-value.

For each base, $1 \le n \le N$, the algorithm iteratively computes part ($0 \le k < K$) of the PMF based on Equation (3) (lines 12–14). Before computing the PMF, it first reads the error probability, p_n, for the current iteration (line 5), 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 *prev* counterparts in preparation for the subsequent iteration.

ALGORITHM 1: Probability Calculation Algorithm.

Input: Error Probability array *Err_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_prev[K];                              // PL_{n−1}(X=k) for k from 0 to K−1
3  PL_prev[0] = 0;                                   // ln(1)
4  for n from 1 to N do
5      p_n = Err_arr[n];
6      ln_pn = ln(p_n);
7      ln_1_pn = ln(1.0 − p_n);
8      if n < K then
9          PL_prev[n] = −1e100;                      // approx. ln(0)
10     end
11     bound = (n < (K−1)) ? n : (K−1);
12     for k from 1 to bound do
13         PL[k] = log_sum_exp(PL_prev[k] + ln_1_pn, PL_prev[k − 1] + ln_pn);
14     end
15     PL[0] = PL_prev[0] + ln_1_pn;
16     if n == K then
17         ln_pval = PL_prev[K − 1] + ln_pn;
18     else if n > K then
19         ln_pval = log_sum_exp(ln_pval_prev, PL_prev[K − 1] + ln_pn);
20     end
21     PL_prev = PL;
                                   // moves data from PL to PL_prev;
22     ln_pval_prev = ln_pval;
23 end
24 return PL, ln_pval;
```

3.2 Computation Characteristics

Table 1 characterizes 46 real SARS-CoV-2 datasets obtained from multiple sources. LoFreq takes these datasets as input. Each of these SARS-CoV-2 datasets contains 29,903 columns. As previously mentioned, every column has two key parameters, N and K. For each dataset, Table 1 shows the characteristics of N and K for all columns within that dataset. The mean value of N is shown. N ranges from 200,000 to 400,000 in common cases, but can be as large as nearly a million (dataset *L21*). The distribution of K is shown in more detail. Besides mean and median, 75th percentile, 90th percentile, 99th percentile, and max values are also shown. These additional values show that while K is small in common cases, it can occasionally be quite large. Taking dataset *L2* as an example, among all columns, the average value of K is 645, the 99th percentile is 3,219, but the largest value is 200,863. As will be shown later, the value of K is critical to the accelerator design. For better visualization, the datasets are presented in ascending order of K and divided into two groups based on the value of K.

The table also shows the execution time using the state-of-the-art parallel implementation of LoFreq. This implementation uses multiple CPU processes to parallelize the computation. First, it divides the dataset into multiple regions, each consisting of many columns, and processes each region using one single-threaded process. Then, it simply gathers the results from these processes. We measured the end-to-end execution time of running LoFreq using 16 processes

Table 1. SARS-CoV-2 Dataset Characteristics

Dataset	Abbr.	N (Depth)		K (Number of Varying Bases)							Exec Time (hours)
		Mean	Std. Dev.	Mean	Std. Dev.	Median	p75	p90	p99	Max	of 16-process CPU
SRR20774184 [3]	S1	161,801	119,857	135	220	83	151	272	912	10,009	0.70
SRR19253115 [3]	S2	186,646	148,768	141	403	75	141	262	1,106	26,501	1.01
SRR20774449 [3]	S3	92,503	75,159	147	205	97	173	296	938	6,425	0.57
SRR21181448 [3]	S4	201,463	183,481	153	305	82	169	333	1,097	13,617	1.12
SRR18781852 [3]	S4	231,935	193,258	223	338	137	269	480	1,344	17,470	1.61
SRR21343573 [3]	S6	221,099	193,148	268	1,439	141	304	555	1,682	168,303	1.80
SRR11177792 [3]	S7	93,959	34,896	277	230	221	365	541	964	8,156	0.47
SRR21333713 [3]	S8	315,505	263,217	289	500	156	333	654	2,133	18,654	2.67
SRR21343113 [3]	S9	246,390	254,002	337	588	165	384	810	2,394	25,771	2.89
SRR16686249 [3]	S10	198,982	142,163	343	632	164	379	819	2,605	30,772	1.77
SRR21345205 [3]	S11	242,332	193,033	343	449	228	425	730	1,955	26,048	2.49
COVHA-P11-B04 [14]	S12	253,004	75,507	347	430	293	404	556	1,214	61,107	1.44
SRR21182863 [3]	S13	258,627	171,854	360	506	226	431	771	2,152	24,169	2.40
SRR21347132 [3]	S14	283,287	284,738	392	653	169	498	994	2,776	21,472	3.60
SRR21347716 [3]	S15	388,310	288,530	418	598	253	504	899	2,651	30,222	3.77
SRR20769207 [3]	S16	434,298	301,942	455	896	263	538	962	2,940	70,311	4.57
SRR20769204 [3]	S17	422,216	274,446	462	651	293	568	973	2,661	28,308	4.71
SRR20494172 [3]	S18	322,330	214,143	468	674	285	553	998	2,878	23,842	3.65
SRR19302558 [3]	S19	247,325	244,608	510	868	258	602	1,235	3,467	31,741	4.13
COVHA-P6-E05 [14]	S20	349,853	132,891	547	382	470	654	897	1,827	13,612	3.42
SRR12380204 [3]	S21	430,569	271,733	606	844	433	756	1,206	2,800	55,208	5.26
SRR21331921 [3]	L1	294,881	245,603	615	873	330	741	1,473	4,000	26,182	4.91
SRR21347735 [3]	L2	383,622	288,941	645	1,696	433	805	1,368	3,219	200,863	5.38
SRR20882805 [3]	L3	318,529	257,317	652	975	353	792	1,497	4,331	22,972	5.39
COVHA-P8-F03 [14]	L4	320,880	112,528	655	486	567	816	1,113	2,117	31,547	3.40
SRR21332795 [3]	L5	361,633	312,553	683	990	368	867	1,674	4,506	50,698	6.49
SRR21178800 [3]	L6	257,202	190,754	721	1,140	413	824	1,542	5,019	54,577	5.53
COVHA-P12-F07 [14]	L7	238,276	62,935	724	383	658	894	1,175	1,953	8,542	2.75
SRR20928666 [3]	L8	369,455	252,515	731	987	433	861	1,600	4,766	24,001	6.39
SRR20759035 [3]	L9	333,566	266,118	740	1,051	419	960	1,716	4,469	47,186	6.48
COVHA-P11-F06 [14]	L10	227,197	99,526	754	442	668	931	1,258	2,204	14,865	3.05
SRR21045270 [3]	L11	295,626	201,697	771	1,315	488	914	1,561	4,646	55,519	5.07
SRR20769574 [3]	L12	281,155	277,207	787	1,745	337	819	1,764	6,927	76,005	8.12
SRR20759528 [3]	L13	277,007	268,096	820	1,410	389	971	1,976	5,782	46,630	7.58
SRR19287736 [3]	L14	285,316	229,988	874	1,348	454	1,032	2,025	6,395	37,266	6.91
SRR20938293 [3]	L15	352,827	200,088	948	852	713	1,211	1,895	4,058	18,021	7.30
COVHA-P1-B03 [14]	L16	461,297	141,367	961	647	824	1,193	1,636	3,126	24,646	7.05
SRR20880842 [3]	L17	317,584	215,732	965	1,074	697	1,244	2,058	4,714	57,248	7.22
SRR20882911 [3]	L18	326,287	227,600	1,017	1,149	672	1,285	2,201	5,213	56,489	8.26
SRR20769571 [3]	L19	375,350	253,473	1,021	1,649	588	1,196	2,156	6,807	71,274	9.73
SRR19289605 [3]	L20	327,822	224,743	1,025	1,597	576	1,203	2,245	7,001	65,040	8.12
H2H-S20L002 [17]	L21	943,993	112,699	1,106	1,098	884	1,284	1,841	4,064	43,420	14.53
SRR20769576 [3]	L22	357,220	287,544	1,284	1,913	636	1,555	3,079	9,087	51,091	11.98
SRR20944431 [3]	L23	455,981	245,527	1,584	1,400	1,201	2,031	3,092	6,665	26,169	13.54
SRR19306790 [3]	L24	303,638	189,653	1,893	1,951	1,298	2,341	4,012	9,844	37,074	10.36
SRR19267282 [3]	L25	215,465	193,169	4,766	5,209	2,847	6,642	11,372	24,316	51,977	32.88

on the 16 hardware-supported threads of an AMD Ryzen 7 5800X CPU. The last column in Table 1 shows the results. More than half of the datasets take over 5 hours, and the longest takes 32.9 hours.

In our measurements, all of the CPU's cores and threads are fully utilized until the very end of the execution. This indicates that the CPU parallelization is effective. However, LoFreq is still slow. The fundamental reason is that the computation of each individual column is slow, because of the amount of work involved.

First, as shown in Algorithm 1, the core computation of each individual column has a total of $(N \times K)$ loop iterations. From the previous discussion on data characteristics, it is clear that this number can be very large. Second, the computation in a single loop iteration is slow: `log_sum_exp` on double-precision floating point numbers is an expensive operation, consisting of serialized logarithm and exponential computations, as shown in Equation (4).

Table 2. Breakdown of LoFreq's Execution Time on SARS-CoV-2 Datasets (Metric: Hours (Percentage))

	Pre-processing	Prob Calculation	Other
SRR11177792	0.28 (7.20%)	3.54 (91.76%)	0.04 (1.04%)
SRR12380204	1.47 (3.54%)	40.01 (96.04%)	0.17 (0.42%)
COVHA-P11-F06	0.76 (3.31%)	22.02 (96.30%)	0.12 (0.39%)

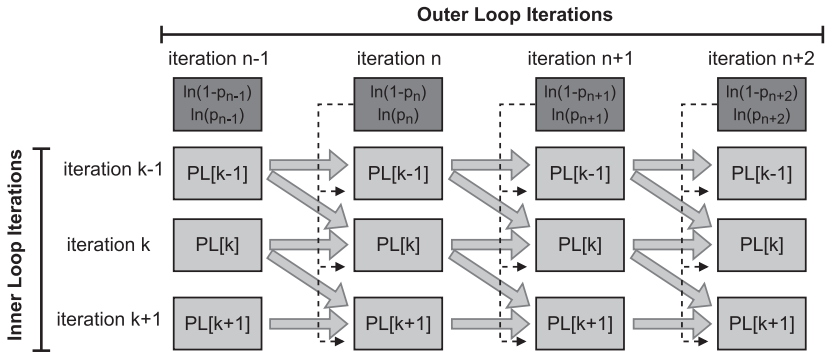


Fig. 2. The dataflow of probability calculation algorithm.

We further identify the performance bottleneck. Table 2 shows the execution time breakdown of LoFreq on several SARS-CoV-2 datasets. This data is collected on the 5800X CPU using a single thread within a single process. Over 90% of the processing time takes place in the core probability calculations (Algorithm 1). This indicates that we must optimize the core probability calculations to accelerate LoFreq.

3.3 Parallelism

Fortunately, there is potential parallelism at multiple levels in the computation. While it is extremely challenging, if not impossible, for general-purpose processors to exploit all of the potential parallelism in the computation, an FPGA is able to effectively do so.

At the top level, columns within one dataset are independent from each other, thus different columns can be processed in parallel (**column-level parallelism**). Different instances of Algorithm 1 can be launched to process columns in parallel. For each SARS-CoV-2 dataset, for example, the total number of columns is fixed (29,903).

In the computation of each individual column, there are another three levels of parallelism. The dataflow of Algorithm 1 is illustrated in Figure 2. First, within one outer loop iteration, different inner loop iterations (including the computation of $PL[0]$ and ln_pval) are independent (**inner-loop parallelism**). Therefore, the degree of parallelism is K in each inner loop. Second, computing the logarithms of p_n and $(1 - p_n)$ can be parallelized with any outer loop iteration prior to the nth one (**operation-level parallelism**).

Finally, in every outer loop iteration, each element in PL depends only on two values (except for $PL[0]$) computed in the previous outer loop iteration. Therefore, the computation of each PL element can start once the input is ready, without having to wait for the previous outer loop iteration to completely finish (**wavefront parallelism**).

3.4 Design Space

The multiple levels of parallelism and the diversity of the datasets pose challenges in the accelerator design and implementation. Moreover, an FPGA has constraints on the design: FPGA resources are

Table 3. Design Space

		Use of Memory for Intermediate Data	
		SRAM	DRAM
Parallelization	Inner Pipelining	Section 4.2.1, 4.4.1	X
	Outer Pipelining	Section 4.2.2, 4.4.1	Section 4.2.2, 4.4.2

finite, and thus must be utilized wisely; certain design options can result in routing failure or low clock frequency, which makes the design impractical. Therefore, a careful design space exploration is necessary.

The accelerator design space is summarized in Table 3. Besides the design knobs of parallelization and memory use, the number of PEs and CUs is another key knob spanning all four quadrants in the table. Designs represented by the top right quadrant (DRAM + Inner Pipelining) are not considered and thus it is marked with an X. The choice of using DRAM only improves functionality, not performance. Therefore, the choice of parallelization is orthogonal to the evaluation of using DRAM. There is no need to evaluate both parallelization strategies for the DRAM designs. And because the outer pipelining designs always outperform the inner pipelining ones (shown in Section 6), only outer pipelining designs are considered and designs in top right quadrant are ignored.

3.4.1 Parallelization. As mentioned in Section 3.3, there are multiple levels of parallelism in the computation. An FPGA is able to exploit all of these levels of parallelism. However, given the resource and design complexity constraints, what levels of parallelism to exploit and to what extent is a challenging yet important question.

To exploit column-level and inner-loop parallelism, the design can exploit the number of hardware units, which is a design knob that needs to be evaluated in every quadrant in Table 3.

Another key knob is how the loop iteration pipelining is done. An inner loop iteration consists mainly of the `log_sum_exp` computation, which can be effectively pipelined in the PE. We use the term **inner pipelining** to refer to this level of pipelining. Meanwhile, we refer to pipelining inner loop iterations across two consecutive outer loop iterations as **outer pipelining**.

Inner pipelining can be easily achieved by the HLS compiler and significantly improves performance. In contrast, **outer pipelining** is more complicated. It is more challenging to implement because of data dependency. It can eventually increase resource use and design complexity, which makes placement and routing more challenging. At the same time, the performance gain from outer pipelining is uncertain.

3.4.2 Memory Use. Another key dimension of the design space is where to store the intermediate data. As illustrated in Algorithm 1, each outer loop iteration uses results computed from the previous outer loop iteration. Therefore, the computed results need to be stored for the next iteration. The size of intermediate data from each outer loop iteration is $8 \times 2 \times (K + 1)$ bytes. This is because 2 intermediate arrays are needed (as shown in Algorithm 1), each of which stores $(K + 1)$ double-precision floating point numbers (size of 8 bytes). Because of the characteristics of K as shown in Table 1, the memory requirement of the intermediate data is small in common cases, but can be exceptionally large occasionally.

On an FPGA board, there is off-chip DRAM (with larger capacity but longer access latency) and on-chip SRAM (with smaller capacity but shorter access latency). There are two design options: using only SRAM and using both SRAM and DRAM.

Exclusively using DRAM to store all intermediate data is not considered, because it is not a reasonable option at all: It is catastrophic to performance, under-utilizing on-chip SRAM resources, and potentially makes routing more challenging.

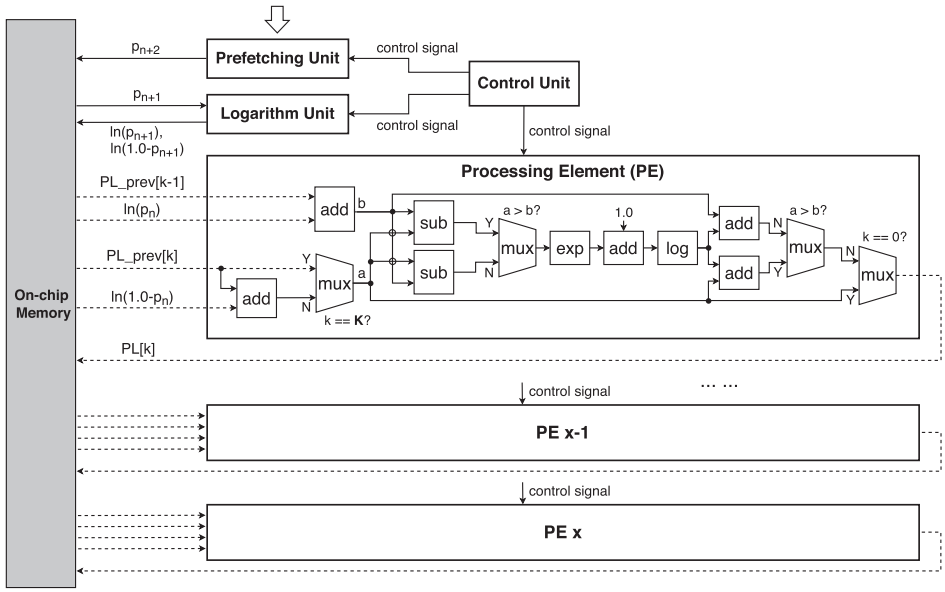


Fig. 3. Design of a column unit (CU).

4 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 accelerator is implemented using the high-level synthesis tools in the Xilinx Vitis Development Platform 2020.2.

This section introduces the FPGA LoFreq accelerator. It first describes the fundamental hardware components, the column units and processing elements. Then, it presents the design space of the accelerator design and discusses the tradeoff among different design points.

4.1 Hardware Units

4.1.1 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 3 shows the design of a CU. The general operation of the CU when using only inner pipelining and storing all intermediate results in SRAM 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_pn and ln_1_pn in the algorithm) are used for this iteration.
   (d) The main **processing elements (PEs)** 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 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 to keep up with the computation. By storing all of the column data in DRAM, the design is decoupled from the values of N for each column and is not memory-resource-limited. The column unit is able to process columns with any value of N (as long as the column data fits in the 64 GB DRAM).

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. 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.

4.1.2 Processing Elements. The core computation of each CU is the computation of $PL_n(X = k)$, which is mostly the `log_sum_exp` calculation of Equation (4). Multiple 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 Kth 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 Kth iteration of the outer loop.

4.2 Processing Element Pipelining

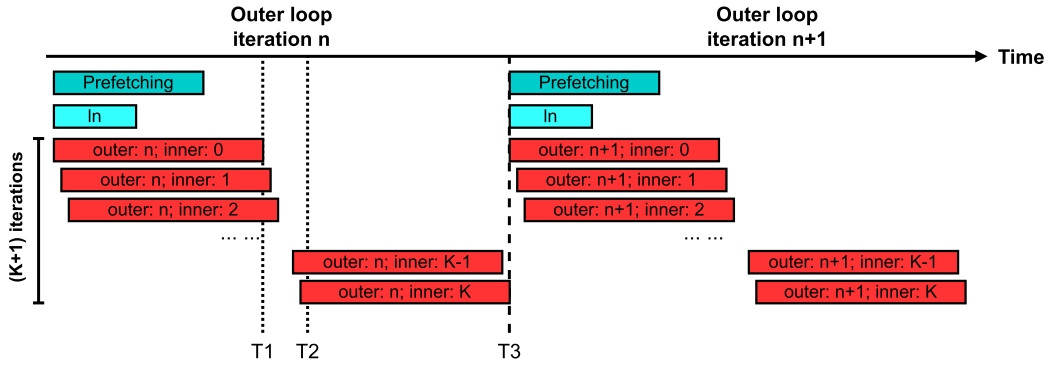
This section explores the pipelining design options. As discussed in the previous section, the PEs are pipelined. However, there are two ways in which they can be pipelined. As a baseline, they are only pipelined *within* a single outer loop iteration, which will be referred to as **inner pipelining**. The design can also be extended to pipeline the PEs both within and *across* outer loop iterations, which will be referred to as **outer pipelining**. This section will further explain inner pipelining, its limitations, and how outer pipelining can overcome those limitations.

For a concise illustration of the issues, $iter(n, k)$ is used to represent the kth inner loop iteration in the nth outer loop iteration. The term `PE_depth` refers to the PE pipeline depth, whose typical value ranges from 80 to 100 on a Xilinx Alveo U250 FPGA.

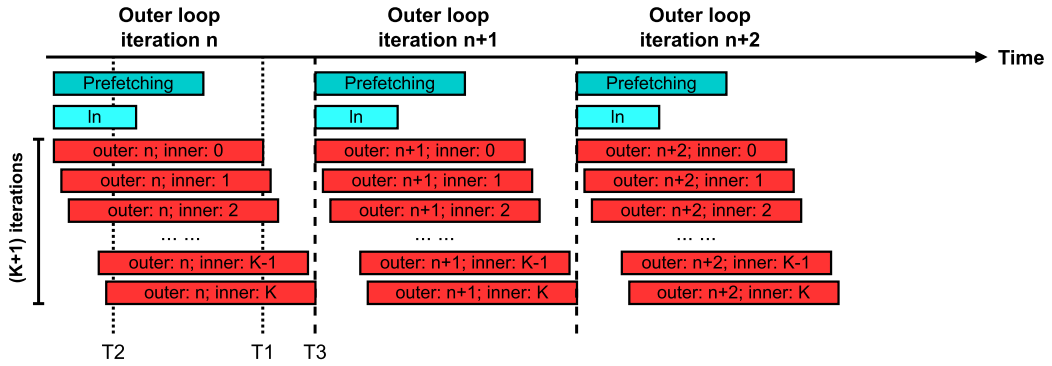
4.2.1 Inner Pipelining. Figure 4 shows the execution of the computation with inner pipelining. The X axis is the time measured in clock cycles and each bar below the axis shows the time span of one of the computations: prefetching, logarithm, or inner loop iterations. The length of a bar shows the number of clock cycles from the start to the finish of that computation. In the range of outer loop iteration n, a new inner loop iteration within an outer loop is initiated every clock cycle, as described in Section 4.1.2. The prefetching and logarithm computation (ln) are completely overlapped with $iter(n, 0)$, as an inner loop iteration always takes longer than prefetching and computing logarithm. Meanwhile, the $(n+1)$-th outer loop iteration starts only after the nth outer loop iteration is completed.

Figure 4(a) shows the drawback of inner pipelining: The fully pipelined PEs suffer from under-utilization because of **pipeline bubbles** between consecutive outer loop iterations.

Consider $iter(n+1, 0)$ as an example. As previously mentioned, $iter(n+1, 0)$ takes the computed results of $iter(n, 0)$ as input (recall that $PL_{n+1}[k]$ requires $PL_n[k]$ and $PL_n[k-1]$). In Figure 4, $T1$



(a) The pipeline must fully drain when K is large.



(b) The pipeline can issue no new operations for at least K cycles between outer loop iterations.

Fig. 4. Computation in a CU with inner pipelining.

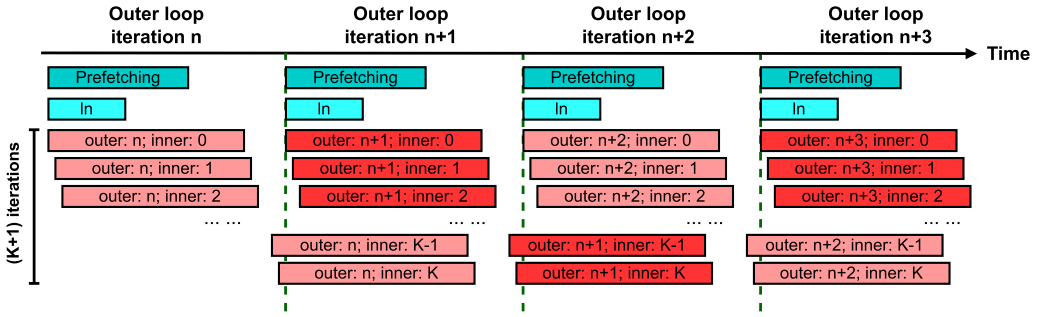
is the time when the input data of $iter(n+1, 0)$ is ready. $T2$ is the time after which the fully pipelined PE has issued $iter(n, K)$. At this point, there is no more computation for iteration n, so there are pipeline bubbles until $iter(n, K)$ completes. $T3$ is the time when $iter(n+1, 0)$ actually starts, because that is when all inner loop iterations in the nth outer loop have completed. With inner pipelining, the PEs suffer from pipeline bubbles for $(T3 - max(T2, T1))$ clock cycles.

Figure 4(a) shows the duration of the bubbles when K is so large that $T2 > T1$. In this case, the pipeline can issue no new computations for $(\texttt{PE\_depth} - 1)$ cycles. In comparison, Figure 4(b) shows a case where the duration of the bubbles is smaller. In this case, because $T2 < T1$ the pipeline can issue no new computations for $(T3 - T1)$ cycles, which equals to K, while K is smaller than or equal to PE_depth. Therefore, the duration of the pipeline bubbles will be small when K is small.

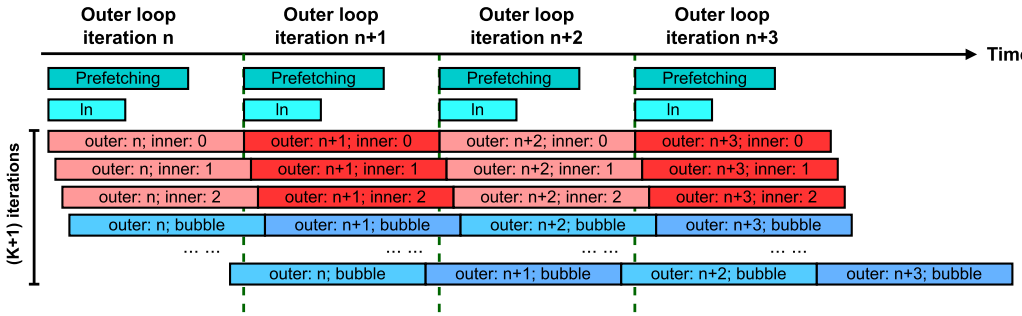
Thus, the duration of the pipeline bubbles in inner pipelining can be summarized as below:

$$Bubbles = \begin{cases} \texttt{PE\_depth} - 1 & \text{if } K > (\texttt{PE\_depth} - 1) \\ K & \text{if } K <= (\texttt{PE\_depth} - 1). \end{cases} \tag{6}$$

Given the characteristics of K as shown in Table 1, the majority of columns would have a K that falls into the first case. Therefore, a CU with inner pipelining can suffer from pipeline bubbles ranging from 80 to 100 clock cycles for every outer loop iteration. As there are in total N outer loop iterations, the total number of clock cycles on pipeline bubbles can be as large as $100 \times N$.



(a) Outer pipelining fully pipelines outer loop iterations when K is larger than or equal to the PE pipeline depth.



(b) Outer pipelining inserts bubbles (partially pipelining outer loop iterations) when K is smaller than the PE pipeline depth.

Fig. 5. Computation in a CU with outer pipelining.

4.2.2 Outer Pipelining. Outer pipelining allows PEs to initiate a new computation every clock cycle, eliminating bubbles in the PE pipeline. Therefore, the next outer loop iteration can start as early as possible and be overlapped with the current outer loop iteration. In common cases, this alleviates the pipeline bubble problem and improves performance.

Figure 5 illustrates the execution of outer pipelining. Outer pipelining works differently in two different cases. The earliest possible time for $iter(n+1, 0)$ to start is when both conditions are met: The time when $iter(n, 0)$ has finished (data dependency) and when $iter(n, K)$ has started (so the PE becomes available). Based on the order of the finishing time of $iter(n, 0)$ and the starting time of $iter(n, K)$, outer pipelining needs to work differently.

In the first case, when K is large enough ($K > (\texttt{PE\_depth} - 1)$), consecutive outer loop iterations are fully pipelined. Figure 5(a) shows this scenario. Consecutive outer loop iterations are highlighted with different colors for better visualization. As shown in Figure 5(a), $iter(n + 1, 0)$ starts exactly one clock cycle after $iter(n, K)$ starts. In this case, outer pipelining completely eliminates the pipelining bubbles found in inner pipelining. As mentioned in Section 4.2.1, this can reduce nearly $100 \times N$ clock cycles in execution time.

In the other case, when K is not large enough, $iter(n, 0)$ finishes later than $iter(n, K)$ starts. In this case, even when the PE becomes available, the next outer loop iteration cannot start yet because of the data dependency on $iter(n, 0)$. Therefore, bubbles must be inserted to ensure that $iter(n + 1, 0)$ does not start until $iter(n, 0)$ is completed. Figure 5(b) illustrates this scenario: $iter(n, 0)$ is still not done one clock cycle after $iter(n, 2)$ is done. Therefore, bubbles (bars marked with text "bubble") need to be continuously inserted until $iter(n, 0)$ is done. It should be noted that

outer pipelining still reduces clock cycles even with these bubbles. As shown in Figure 5(b), there is a two-clock-cycle reduction for each outer loop iteration compared to inner pipelining.

4.2.3 Tradeoffs. While outer pipelining always reduces the number of cycles needed to complete the overall computation (by at least a small amount), it requires additional resources. This can impact the overall design, complicate timing closure, and create problems for placement and routing. Therefore, it is necessary to evaluate the tradeoff between using inner pipelining on its own compared to both inner and outer pipelining.

4.3 PEs vs. CUs

Given the finite amount of FPGA resources, there is a tradeoff between the number of PEs per CU and the overall number of CUs. However, this performance tradeoff is not clear cut, as the benefits and disadvantages of each design depend on the datasets being processed and the behavior of the host system (to be discussed in Section 5).

More PEs in a column unit can exploit more inner-loop parallelism, but less column-level parallelism. While fewer columns can be processed in parallel, each column will be processed faster. This is particularly helpful when there is imperfect load balancing, because the majority of CUs sit idle while the final columns are processed. With more PEs in a column unit, this tail latency will be reduced, as the final columns can be processed faster, minimizing the impact of the load imbalance.

In contrast, a larger number of CUs, each with fewer PEs, can achieve higher levels of column-level parallelization. Furthermore, there will be fewer outer pipelining bubbles, as discussed in Section 4.2.2. This can potentially lead to faster processing times for the bulk of the computation.

4.4 Memory Use

This section explores the design option of where to store the intermediate data. As discussed in Section 3.4.2, the memory size required is proportional to K. There are two potential options: using only on-chip SRAM for intermediate data and using both SRAM and DRAM. The design adopting the first option will be referred to as the **SRAM design**, and the other the **DRAM design**.

This section closely examines the two designs and analyzes the tradeoffs.

4.4.1 The SRAM Design. The SRAM design potentially has multiple benefits. First, every single memory access will be fast, as the on-chip SRAM accesses take 1 to 3 clock cycles. Second, it has the minimum amount of DRAM traffic, and this has advantages in both performance and resource utilization. With less DRAM communication, the design will be much less likely to cause DRAM contention and a related performance drop. From the resource perspective, the design requires less DRAM-related logic, which can lead to resource savings.

However, this option is fundamentally limited. To ensure the computed results are correct, the on-chip SRAM buffer for intermediate data needs to be as large as $8 \times 2 \times (K + 1)$ for the largest possible K, as explained in 3.4.2. Using the max values of K in Table 1, i.e., 200,863, as an example, it requires 3.06 MB on-chip SRAM for each CU. This can not only limit the number of CUs that can be implemented, but also increase the difficulty of placement and routing. Moreover, in reality, the largest possible K can be hard to know. Thus, it is difficult to determine an SRAM size that is guaranteed to be large enough for any column.

4.4.2 The DRAM Design. The DRAM design resolves the storage limitation of the SRAM design. It uses the DRAM to store intermediate data and overlaps the DRAM load and store with computations. In this way, it benefits from the 64 GB DRAM capacity without slowdown.

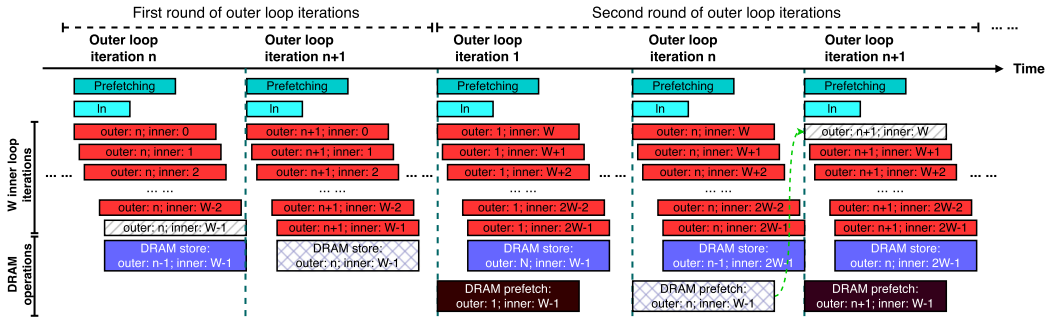


Fig. 6. Computation in the DRAM design.

The main reason why each outer loop requires $8 \times 2 \times (K + 1)$ bytes for intermediate data is because of the loop computation order. All outer loop iterations are processed in order, and the $(n + 1)$-th outer loop iteration does not start until all inner loop iterations in the nth outer loop iteration finish. All computed results from $(K+1)$ inner loops (including the computation of $PL[0]$ and ln_pval) need to be stored for the use in the next outer loop iteration. Therefore, the memory requirement is proportional to K.

The DRAM design uses an alternative strategy: Each outer loop iteration computes only a fixed number (denoted by W) of—instead of all $(K + 1)$—inner loop iterations before proceeding to the next outer loop iteration. W is a constant value that is determined before the hardware is implemented. As a result, there can be more than one round of outer loop iterations when W is smaller than $(K + 1)$, as visualized in Figure 6: In the first round, the first W inner loop iterations in every outer loop iteration are processed; in the second round, the next W inner loop iterations in every outer loop iteration are processed, and so on. (In columns where $(K + 1)$ is smaller than or equal to W, there is only one round of outer loop iterations.) This process repeats until all inner loop iterations in every outer loop iteration are processed. In this way, only a fixed amount ($8 \times 2 \times W$ bytes) of on-chip SRAM is needed to process columns with arbitrarily large K.

However, the dataflow of LoFreq poses a challenge: A large number of intermediate values must be saved for a much longer period of time compared to the SRAM design. In Figure 6, the output of $iter(n, W-1)$ (the left single-hatched bar) is used not just by $iter(n+1, W-1)$, but also $iter(n+1, W)$ (the right single-hatched bar), which complicates the problem: $iter(n + 1, W)$ is computed N outer loop iterations after $iter(n, W - 1)$. Therefore, the output of $iter(n, W - 1)$ needs to be saved for N outer loop iterations after being produced. Every block of W inner loop iterations requires saving one such value. This can be understood using Figure 2: Blocking the inner loop iterations can be thought of as making a cut horizontally. As long as W is smaller than $(K+1)$, such a cut will always cross a dataflow arrow in every outer loop iteration. One such cross means that one output value needs to be saved. Such an output value needing to be saved will be referred to as *boundary values*.

In total, there will be N boundary values to be saved in every round of outer loop iterations. Given the characteristics of N as shown in Table 1, it is only practical to store them in DRAM. While the sharp increase in DRAM accesses seems daunting, the DRAM design effectively hides DRAM access latency by exploiting the fact that the final use of a boundary value is N outer loop iterations away from where it is computed. Because N is large in common cases, there is enough time for the DRAM design to store (to DRAM) and prefetch (from DRAM) every boundary value. To be more specific, boundary values will be stored to DRAM one outer loop iteration after they are computed, and they will be prefetched from DRAM one outer loop iteration before their use. Figure 6 shows an example: In the first round, the boundary value from $iter(n, W - 1)$ is stored to DRAM in the $(n + 1)$-th outer loop iteration (shown in the left crosshatched bar); in

the second round, that boundary value is prefetched in the *n*th outer loop iteration (shown in the right cross-hatched bar) for its upcoming use. In this way, DRAM accesses are done in parallel with computations.

4.4.3 Tradeoffs. Using a small and fixed-size of SRAM buffer, the DRAM design can handle any column, regardless of its K value. However, even though DRAM accesses can be parallelized with other computations, there can still be a potential slowdown caused by DRAM contention, particularly when the design has multiple CUs working in parallel. Moreover, the design requires more complicated control and DRAM access logic, which can lead to an increase in resource use.

In contrast, the SRAM design is limited, as it cannot handle columns with K exceeding the on-chip SRAM capacity. However, it tends to have better performance and less resource use. Moreover, in cases where it is certain that N will be small (thus, K will be small), the limitation of not being able to handle large K will not be a problem.

5 HOST SYSTEM DESIGN AND IMPLEMENTATION

LoFreq is written in standard C as a single-threaded program. 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 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 enqueueing 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 3.2, 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

Table 4. The Hardware Configurations and Total Resource Usage for Different FPGA Designs

	Hardware Configuration					Individual CU Resource Use			Total Resource Use			
	PE Pipelining	Memory Use (KB)	No. of CUs	No. of PEs per CU	Frequency (MHz)	FF	LUT	DSP	CLB	FF	LUT	DSP
I/S/32	I	S (1024)	3	32	300	10.81%	20.13%	13.7%	78.15%	33.25%	66.48%	39.53%
I/S/16	I	S (1024)	7	16	300	5.52%	10.45%	6.19%	89.03%	40.44%	79.90%	43.45%
I/S/8	I	S (1024)	14	8	300	2.97%	5.61%	3.63%	91.13%	44.88%	84.57%	50.86%
I/S/4	I	S (1024)	24	4	300	1.76%	3.23%	2.35%	91.87%	49.11%	84.32%	56.49%
O/S/8	O	S (1024)	14	8	300	2.18%	4.13%	4.57%	86.10%	41.35%	72.00%	73.90%
O/S/4	O	S (128)	24	4	300	1.37%	2.43%	2.69%	89.30%	45.66%	73.92%	74.54%
O/D/8	O	D (64)	13	8	300	2.41%	4.31%	4.68%	86.50%	41.98%	70.06%	70.33%

affects the tradeoff 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.

6 EVALUATION

6.1 Experimental Setup

System Configuration. The system used for evaluating the accelerator designs 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) [7]. This 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 uses Xilinx Vivado 2020.02 for logic optimization, placement, and routing. Aggressive optimization strategies are used in placement and routing: The *SSI_SpreadLogic_high* strategy is used for placement, and the *AggressiveExplore* strategy is used for routing [6]. Physical optimizations were enabled. Individual column units were implemented within one Super Logic Region [8] to minimize boundary crossings. Moreover, latency and resource use of logarithm and exponential operations are configured for successful placement and rounting. All of the evaluated FPGA designs are shown in Table 4.

Datasets. Table 1 describes the real-world datasets used throughout this evaluation. These datasets come from the NCBI SARS-CoV-2 data repository [3] and recent clinical studies [14, 17].

Metrics. The performance and power of different designs are evaluated and compared, with a focus on performance. The performance is evaluated by measuring the wall clock time of execution. Two types of power numbers are measured: FPGA-only power (without CPU power) and whole system power (CPU + FPGA). The FPGA-only power is measured using the Xilinx Board Utility tool (Xbutil 2020.2). During execution runs, the tool runs along on the host CPU, sampling and reporting the FPGA power number at a sampling rate of 1 Hz. The whole system power is measured using a P3 P4460 Kill-A-Watt monitor.

Baseline. The stock multi-process implementation of LoFreq [5] is used as the baseline in the 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.

6.2 Design Space

Table 4 shows all the points in the design space and their hardware configurations and resource utilizations. The designs are named based on their configurations. In the Hardware Configuration section, the key design options are presented. In the PE Pipelining column, **I** and **O** stand for inner pipelining and outer pipelining, respectively. In the Memory Use column, **S** stands for using only SRAM and **D** for using DRAM for intermediate data. The size (in KB) of the on-chip SRAM buffer used in each CU is shown next to the memory type. For all *S* designs, the largest *K* a design can handle is limited by that size. In contrast, O/D/8 can handle columns with arbitrarily large *K* while using a fixed amount of on-chip SRAM (64 KB). The size is 64 KB because the constant value *W* (introduced in Section 4.4.2) is empirically chosen to be 4,096.

The first four designs implement inner pipelining and use only SRAM for intermediate data. They differ in the number of CUs and PEs per CU. The last three designs all implement outer pipelining. O/S/8 and O/S/4 differ only in the number of PEs; O/S/8 and O/D/8 differ in memory use and the number of CUs. Only power-of-2 numbers are considered for the number of PEs per CU. 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.

Implementing outer pipelining and implementing DRAM prefetch/store add significantly to the difficulty of placement and routing. For a design with 16 or 32 PEs, implementing outer pipelining leads to routing failure (due to congestion). Thus, they are absent from the table. For O/D/8, it can only implement 13 CUs instead of 14 as in O/S/8 for the same reason.

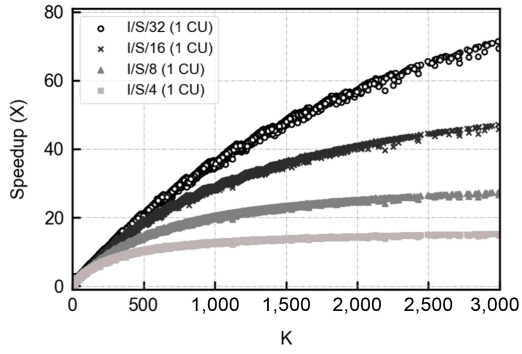
The rest of this section evaluates these designs mainly on their performance, but also presents power consumption results. In Section 6.3, the performance of processing individual columns using a single CU is evaluated, with a focus on the impact of three key design knobs: the number of PEs, PE Pipelining, and Memory Use. In Section 6.4, the end-to-end performance is evaluated on all datasets from Table 1, for the complete designs, each with multiple CUs. End-to-end power consumption results are presented in Section 6.5.

6.3 Probability Calculation Speedup Results

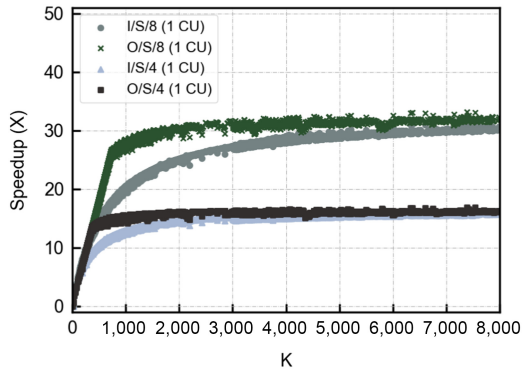
This section presents speedup results for just the probability calculation on a column, which is the computation that is accelerated by the FPGA. The computational cost of this workload is dictated by the column's values for *N* and *K*. *N* determines the number of outer loop (starting at line 4 in Algorithm 1) iterations and *K* determines the number of inner loop (starting at line 12 in Algorithm 1) iterations. The performance of one single CU from each design in Table 4 is measured.

Figure 7 presents the speedup results. Each figure compares a different group of designs, highlighting the performance impact of turning a certain design knob. All three figures show the speedup results measured on columns from datasets in Table 1. Each mark represents a column data point. The Y axis value is the speedup. 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.

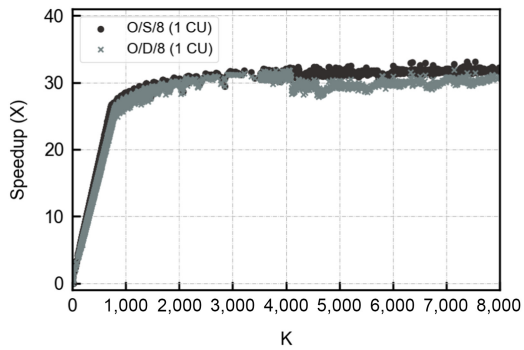
6.3.1 Number of PEs. Figure 7(a) shows the speedup results of the designs varying only the number of PEs, which dictates the level of inner-loop parallelization.



(a) Tuning the number of PEs.



(b) Outer Pipelining vs. Inner Pipelining.



(c) DRAM vs. SRAM.

Fig. 7. Single column unit speedup results.

At any given value of K, using a larger number of PEs increases the speedup. However, when K is small, the speedup from increasing the PEs is marginal. Using the point where K is 1,000 as an example, the speedup results are: 35× (I/S/32), 28× (I/S/16), 20× (I/S/8), and 13× (I/S/4). The performance scaling is limited. This is mostly because PEs are underutilized with only inner pipelining, as discussed in Section 4.2.1. In contrast, when K is larger, increasing PEs leads to a larger increase in speedup. Using where K equals 3,000 as an example, the speedups are 69× (I/S/32), 46× (I/S/16), 27× (I/S/8), and 15× (I/S/4). In this case, increasing the number of PEs from 4 to 8 nearly doubles the speedup.

However, as shown in Table 4 (Individual CU Resource Use section), doubling the number of PEs consistently doubles the resource utilization, which can directly reduce the number of CUs that can be implemented.

6.3.2 PE Pipelining. Figure 7(b) shows the speedup of the outer pipelining designs and highlights their improvement over the inner pipelining designs. To show how the improvement changes with K, the results of K from 1 to 8,000 are shown.

O/S/8 (marked by green x) shows a significant improvement over I/S/8 (marked by grey dot). The performance improvement can be visualized as the area between the the curves of marks. The larger the area, the greater the performance improvement is. In contrast, the improvement of O/S/4 over I/S/4 is smaller. For O/S/8, the improvement is most significant on columns whose K ranges from 700 to 3,000; for I/S/4, that range is from 300 to 700.

The results also show outer pipelining leads to better performance improvement when the number of PEs is larger and confirms that outer pipelining always improves performance on any columns compared to inner pipelining. In both designs, there is a stage where the speedup grows linearly as K increases. That is the region of K where outer pipelining needs to insert bubbles.

6.3.3 DRAM. Figure 7(c) shows the performance impact of the design option of using DRAM to store intermediate data. In short, the performance is as good as O/S/8 on columns whose K is less than 4,096 but suffers on columns whose K is larger than 4,096.

This is because the fixed on-chip buffer in O/D/8 is just large enough to hold the intermediate data of 4,096 inner loop iterations. Therefore, for columns whose K is less than 4,096, there is no on-board DRAM prefetch or store for the intermediate data. This indicates that intensive on-board DRAM accesses result in a noticeable performance penalty.

6.4 End-to-end Performance

This section is organized to answer the following critical questions about the designs:

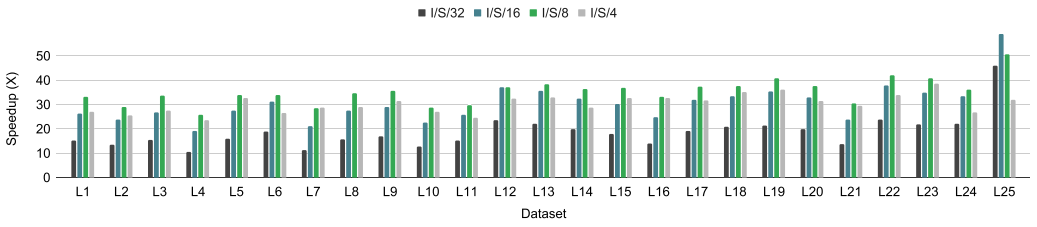
(1) What is the best configuration in terms of the number of CUs and the number of PEs?
(2) Is outer pipelining always better?
(3) Is there a performance penalty for the DRAM design? If so, by how much?

To that end, this section presents the end-to-end performance of all the designs in Table 4. All designs are evaluated on end-to-end speedup over the multiprocess CPU baseline on processing complete datasets from Table 1. As previously mentioned, the datasets are divided into two groups based on the average value of K among all columns in that dataset. In all figures presented in this section, the X axis shows the datasets, and they are placed from left to right in ascending order of the average value of K.

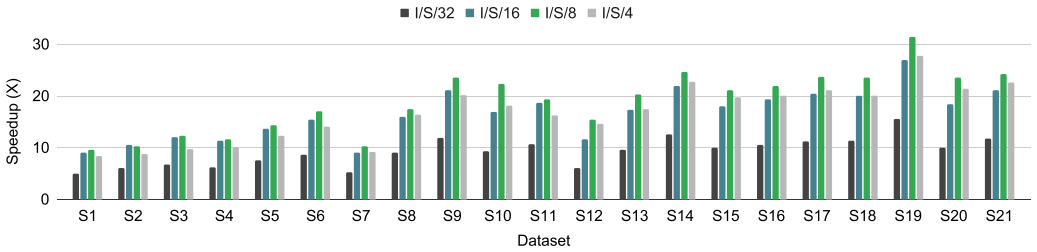
To unleash the full performance of each FPGA design, we evaluate these designs using the multi-process LoFreq described in Section 5. A fixed number of chunks is chosen for each design: 72 (I/S/32), 112 (I/S/16, I/S/8, O/S/8, O/D/8), 160 (I/S/4). It is unsurprising that more chunks are needed as the 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.

6.4.1 CUs vs. PEs. Figure 8 highlights the tradeoff between having more CUs and having more PEs per CU. In both sub-figures, the height of the bar indicates the end-to-end speedup.

Figure 8(a) shows the speedup results on datasets with larger K. The highest speedup among the four designs is consistently over or around 30×. The largest observed is 59× using I/S/16 on



(a) Speedup Results on datasets with larger K.



(b) Speedup Results on datasets with smaller K.

Fig. 8. End-to-end speedup of I/S designs over the CPU baseline.

L25, which has the largest overall K value among all datasets. Figure 8(b) shows the speedup on datasets with smaller K, where the most common speedup ranges from 10× to 20×.

By comparing the results in Figures 8(a) and 8(b), as well as the results within each figure from left to right, it is clear that the end-to-end speedup is larger as the average value of K grows larger. This is consistent with the results shown in Figure 7(a).

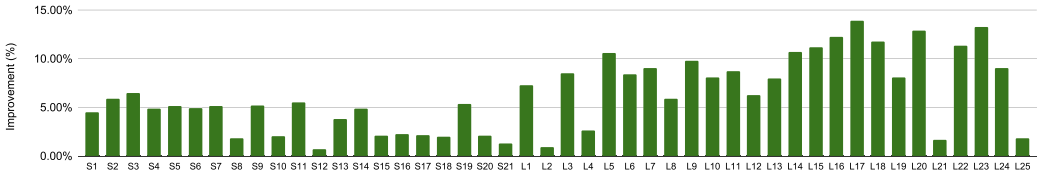
Among the four designs varying only in the number of PEs and CUs, I/S/8 performs the best on almost all datasets (with two exceptions: L25 and S2). This is a strong indication that I/S/8 strikes the best balance between column level and inner-loop level parallelization. In L25, as the dataset is divided into chunks, one chunk of data has a significantly larger workload than the rest and becomes the tail bottleneck: It was the only unfinished chunk for the final 7 minutes of the execution of I/S/8. As the computation within each chunk is sequential, I/S/8 was slower than I/S/16 in processing the tail chunk. L25 is the only dataset exhibiting such a long single chunk tail latency.

6.4.2 PE Pipelining. Figure 9 highlights the performance improvement of outer pipelining over inner pipelining. In Figures 9(a) and 9(b), the Y axis shows the percentage of reduction in execution time of O/S/8 over I/S/8, and of O/S/4 over I/S/4, respectively.

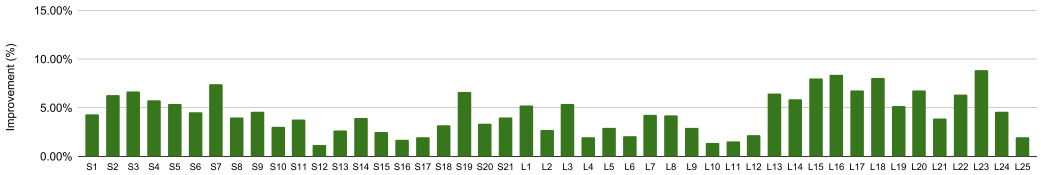
In both configurations, implementing outer pipelining always leads to an improvement in the end-to-end performance. Moreover, how the improvement changes across datasets, which is essentially how the improvement changes over K values, is consistent across O/S/8 and O/S/4. This can be seen by comparing Figures 9(a) and 9(b) in how the heights of the bars change from left to right.

The performance improvement of O/S/8 over I/S/8 is almost always larger than that of O/S/4 over I/S/4. This is most obvious on the large K dataset group. This confirms again that outer pipelining is more effective when there are 8 PEs rather than 4 PEs, which aligns with the results previously shown in Figure 7(b).

Figure 10 directly compares the performance between the two outer pipelining designs. O/S/8 consistently outperforms O/S/4. Considering that O/S/8 already improves upon the best design

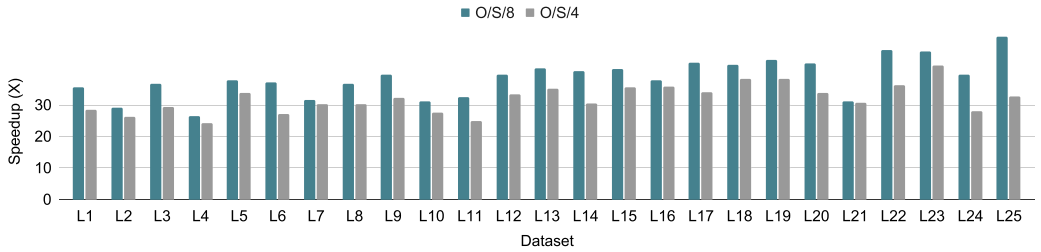


(a) Improvement of O/S/8 over I/S/8.

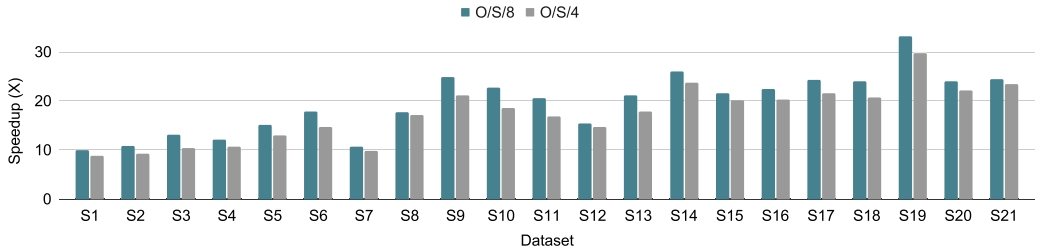


(b) Improvement of O/S/4 over I/S/4.

Fig. 9. End-to-end performance improvement from outer pipelining.



(a) Results on datasets with larger K.



(b) Results on datasets with smaller K.

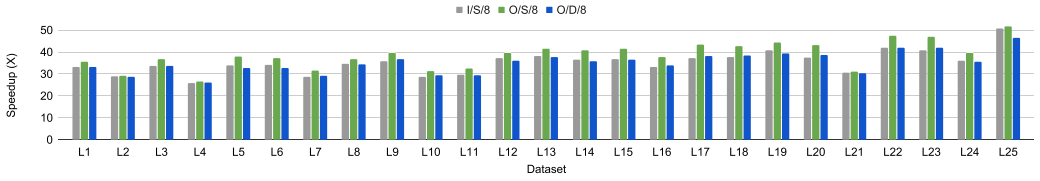
Fig. 10. End-to-end speedup of O/S designs over the CPU baseline.

from the previous section (I/S/8), this effectively shows that O/S/8 is the optimal design in terms of performance. Thus, O/S/8 is chosen for a closer examination.

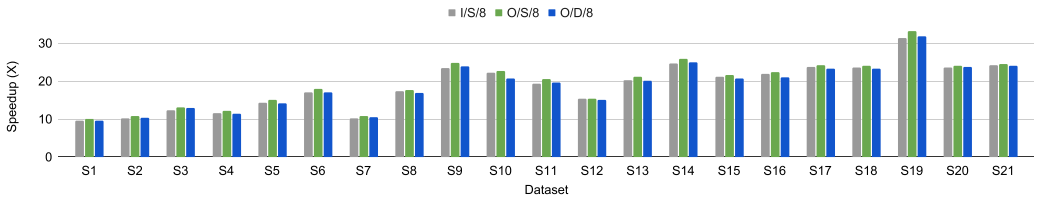
In Figure 9(a), the performance improvement is commonly larger than or around 10% on the larger K datasets (L1 to L25); the improvement is around 5% on the smaller K datasets (S1 to S21).

Referring back to Figure 7(b), the proportion of columns whose K is between 700 and 3,000 largely dictates the overall improvement of O/S/8 over I/S/8. Consider the datasets from $L15$ to $L24$. Their mean, median, and 90th percentile of K indicates that a major portion of the columns fall into the range where the improvement is significant.

There are cases when the overall improvement is small, less than 2%. This is most likely to happen when a large number of columns have K values larger than 3,000. $L25$ is such a case. In



(a) Speedup Results on datasets with larger K.



(b) Speedup Results on datasets with smaller K.

Fig. 11. End-to-end speedup of O/D/8.

those cases, the performance is dictated by these columns, on which outer pipelining does not achieve much improvement.

6.4.3 DRAM Design. Figure 11 highlights the performance of the DRAM design (Design O/D/8). There is a consistent slowdown compared to Design O/S/8. The first reason is that the total number of CUs is smaller (13 compared to 14), which reduces the column-level parallelism.

Another source of slowdown is the performance drop on larger columns, as shown in Figure 7(c). In Figure 11(b), where the 99th percentile of K for all datasets is less than 4,096, the slowdown of O/D/8 is smaller compared to that of Figure 11(a). In both figures, the slowdown tends to be larger and larger from the left to right.

Still, even with the slowdown from having one less CU and the performance penalty caused by constant DRAM accesses, the end-to-end speedup from O/D/8 is comparable to that from I/S/8. The only exception is on $L25$, where the K are exceptionally large.

This not only highlights the advantage of outer pipelining, but indicates that the speedup of design O/D/8 is still significant, outperforming designs I/S/4, I/S/16, and I/S/32.

6.5 End-to-end Power Analysis

The FPGA-only (without CPU) power and whole system (with CPU) power are presented and analyzed in this subsection. For both types of power, the measurement is done throughout complete execution runs on datasets. This subsection first shows how FPGA-only power consumption varies across accelerator designs and then analyzes the whole system (including CPU and FPGA) power and energy efficiency.

6.5.1 FPGA-only Power and Energy. FPGA-only power numbers are sampled throughout an execution run at an interval of 1 second, based on the methodology described in Section 6.1. This measurement is done individually for each design on six representative datasets. The distribution of measured FPGA-only power numbers is shown in the box plots in Figure 12. In the box plots, the top of the rectangle box is the 75th percentile value (Q3), the bottom is the 25th percentile value (Q1), and the line within the rectangle box is the median value. The Q3, Q1, and median values exhibit the power characteristics of different designs.

From all six box plots in Figure 12, it is shown that the power consumption varies across accelerator designs. The characteristics of such variations are consistent across all six datasets. The overall

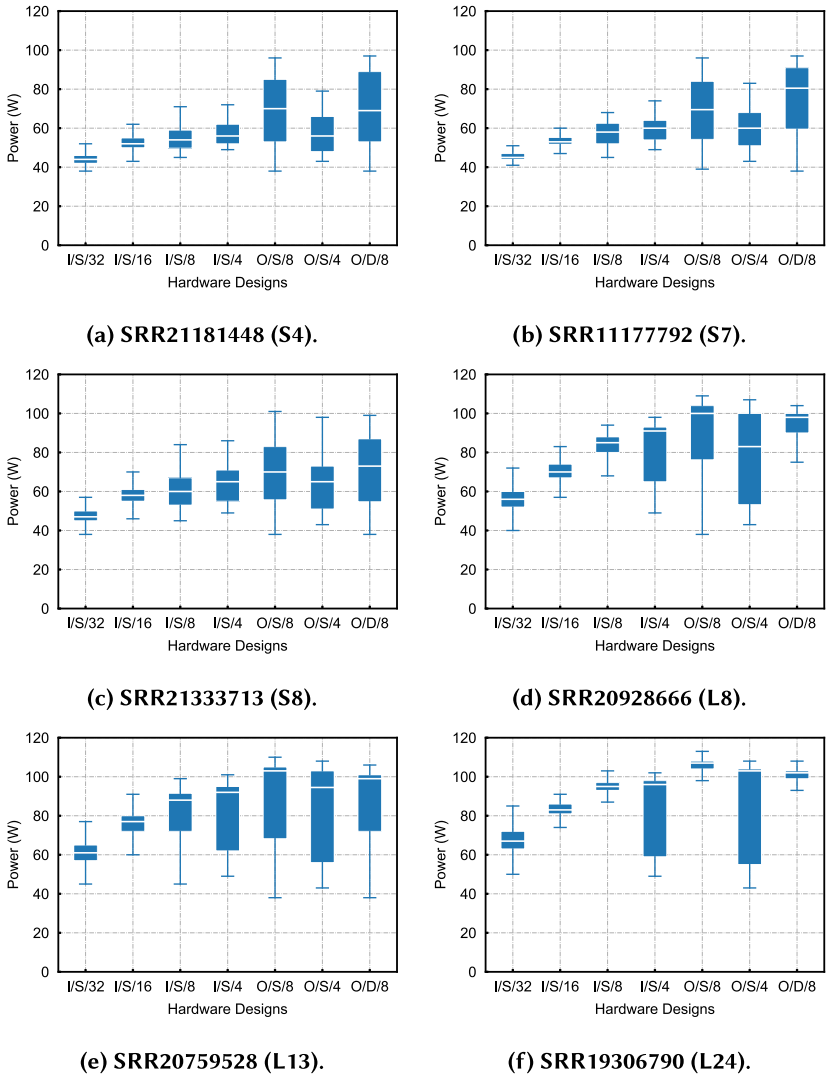
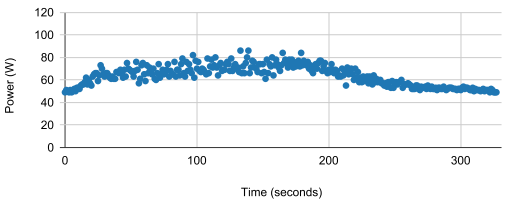


Fig. 12. Distribution of measured FPGA-only power during the end-to-end execution on multiple datasets.

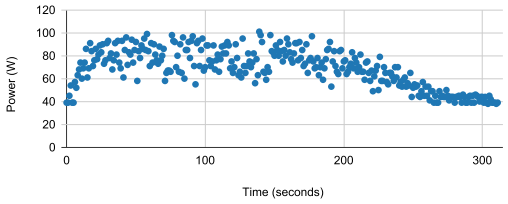
power consumption of the first four designs (I/S/32, I/S/16, I/S/8, I/S/4) increases as the number of PEs per CU decreases. This is because designs with less PEs per CU implement more CUs and have a higher overall resource utilization, as shown in the Total Resource Use column in Table 4. For example, the CLB utilization rate of I/S/32 and I/S/4 are 78.15% and 91.87%, respectively.

The design choice of outer pipelining tends to increase power consumption. On all six datasets, the power of O/S/8 is consistently higher than that of I/S/8, and similarly for O/S/4 versus I/S/4. Meanwhile, the impact of memory use is less obvious. The power of O/S/8 and O/D/8 are at a similar level.

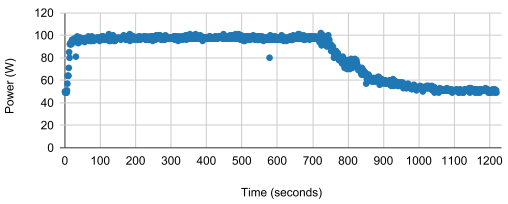
To help understand the box plots, Figure 13 shows the raw sampled FPGA-only power numbers over time throughout complete execution runs. Two datasets with largely different characteristics are chosen for demonstration: *SRR*21333713 (with small K) and *SRR*19306790 (with large K). The



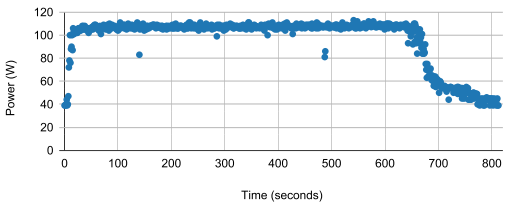
(a) Design I/S/4 on SRR21333713 (S8).



(b) Design O/S/8 on SRR21333713 (S8).



(c) Design I/S/4 on SRR19306790 (L24).



(d) Design O/S/8 on SRR19306790 (L24).

Fig. 13. Measured FPGA-only power consumption throughout execution runs.

large variation of power consumption during an execution run is the main reason why some boxes in Figure 12 are tall.

For datasets with larger K, such as the one shown in Figures 13(c) and 13(d), the power consumption is high for the majority of time (active phase) and then gradually decreases towards the end (tail phase). During the tail phase, the power is reducing, because there is not enough chunks of columns remaining to keep the FPGA highly utilized as more chunks are finished. This characteristic is less obvious but still exists on the dataset with smaller K. On datasets with smaller K, as shown in Figures 13(a) and 13(b), because computation workload of columns tends to be lighter, the FPGA is more often under-utilized even during the active phase. Therefore, the power consumption is less stable during the active phase.

Overall, designs with higher performance, such as $O/S/8$ and $I/S/8$, tend to be higher in power. The FPGA-only energy consumption is not directly measured due to the lack of tool support. However, despite the higher power, the higher performance designs have shorter execution time. Based on the execution time and the sampled power numbers, the total energy consumption of the higher performance designs are still lower than that of the lower power designs.

6.5.2 Whole System Power and Energy. The whole system power is measured using the Kill-A-Watt monitor, as described in Section 6.1. During the measurement, the whole system power is mostly stable throughout an execution run. Therefore, for each execution run, the most common power consumption number is chosen from the monitor for analysis. While not directly measured using the monitor, energy consumption of individual execution runs are estimated from execution time and power numbers.

Based on the estimation, it is clear that using the accelerator leads to a total energy consumption reduction compared to the CPU baseline. The execution run on dataset *SRR*20928666 is used as an example. In our measurement, the power of the CPU baseline (described in 6.1) is $155W$. For the accelerator system, the design with the highest power consumption (O/S/8) is chosen for analysis. The accelerator system has a power of $260W$ (including CPU and FPGA power). Despite the higher power, the execution time is much shorter based on the end-to-end speedup results from Section 6.4. Thus, the total energy consumption of the accelerator system is much less. The CPU

baseline and the accelerator system take 23,000 seconds and 630 seconds, respectively. Based on the power and execution time, the approximate energy consumption reduction is 22×.

6.6 Summary

The evaluation section has comprehensively explored the design space of the accelerator. This subsection first summarizes the overall results and then draws conclusions about the accelerator design space exploration.

6.6.1 Overall Results. First, design O/S/8 achieves the highest speedup over the baseline and thus the best performance. This highlights the impact of implementing outer pipelining and choosing a proper number of CUs and PEs. The power consumption varies across designs, and higher performance designs (O/S/8 and I/S/8) tend to have a higher power consumption. Despite the higher power, the higher performance designs are more energy efficient because of shorter execution time. Similarly, compared to the baseline CPU system, the accelerator system (including CPU and FPGA) tends to be much more energy efficient, thanks to greatly reduced execution time.

In terms of functionality, all designs can produce correct results (exact same results as the software) on columns whose K is smaller than 65,536. Meanwhile, only design O/D/8 produces correct results, regardless of how large K is: It is able to produce correct results processing columns with any sizes of K (as long as they fit in the 64 GB on-board DRAM). The performance of O/D/8 is worse than O/S/8 but is similar to I/S/8.

6.6.2 Conclusions. The design space exploration has identified O/S/8 and O/D/8 as the two overall best designs but with different tradeoffs. O/S/8 has the best performance but is limited in functionality; O/D/8 has no limitations in functionality at all but is slower compared to O/S/8.

All three design knobs (the number of PEs and CUs, PE pipelining, DRAM use) are critical to the accelerator design and implementation. Implementing outer pipelining is key to performance, as it is always better than inner pipelining. Choosing the number of CUs and PEs is also critical to performance. Meanwhile, using DRAM for intermediate data storage eliminates restrictions on functionality so the accelerator can produce correct results on any columns.

6.6.3 Portability Discussion. The accelerator design space does not change across different hardware platforms: What the design knobs are (the number of PEs and CUs, PE pipelining, DRAM use) remains the same. Meanwhile, when the hardware platform changes, all three knobs need to be tuned to accommodate the new resource and platform constraints. Tuning the number of PEs and CUs is straightforward. The other two knobs also need to be tuned, because their performance impact and resource use can be different. Thus, when the target hardware platform changes, the infrastructure of the accelerator remains the same, while individual design knobs need to be tuned for best performance.

7 RELATED WORK

A wide range of FPGA-based accelerators have been developed for important bioinformatics computation, such as Read Mapping [9, 10, 12, 15, 16, 18, 19, 21, 24, 28, 32, 33], Variant Calling [1, 11, 22, 31, 37], Alignment Refinement [36], and Base Quality Score Recalibration [26].

For Variant Calling, most of the existing work focuses on the GATK HaplotypeCaller [27, 29] and its application to human genome data [1, 11, 22, 31, 34]. Unlike LoFreq, which is alignment-based, GATK is a local assembly-based variant caller. The GATK HaplotypeCaller is specifically designed to perform variant calling on low-depth human genome data, while LoFreq specializes in calling low-frequency variants on genome data with high depth.

The state-of-the-art implementation of LoFreq is the stock multi-process implementation [5]. Kille et al. [23] proposed a Poisson-Binomial approximation technique to LoFreq. This technique computes an approximation, instead of the exact pvalue, with lightweight computation to filter out columns. This technique is orthogonal to our contributions to accelerate the exact computation.

Nvidia Clara Parabricks is a software suite of common Whole-Genome Sequencing analysis tools that are optimized for running on high-end Nvidia GPUs [4]. 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 four Nvidia V100 GPUs [2].

8 CONCLUSION

This article has presented the design of an FPGA-based accelerator for the LoFreq variant caller that can achieve up to 51.7× 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 article 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 are tradeoffs along several dimensions, including parallelism, pipelining, and intermediate storage. 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.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. We appreciate Dr. Todd Treangen, Bryce Kille, Yunxi Liu, Yilei Fu, and Nicolae Sapoval for their consistent help.

REFERENCES

[1] Intel. 2017. *Accelerating Genomics Research with OpenCL and FPGAs.* Retrieved from January 14, 2022 https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-accelerating-genomics-opencl-fpgas.pdf.

[2] Nvidia. 2021. *GPU-Accelerated Tools Added to NVIDIA Clara Parabricks v3.6 for Cancer and Germline Analyses.* Retrieved from April 7, 2022 https://developer.nvidia.com/blog/gpu-accelerated-tools-added-to-nvidia-clara-parabricks-v3-6-for-cancer-and-germline-analyses/.

[3] NIH. 2022. *NCBI SARS-CoV-2 Resources.* Retrieved from January 15, 2022 https://www.ncbi.nlm.nih.gov/sars-cov-2/.

[4] Nvidia. 2022. *Nvidia Clara Parabricks Documentation.* Retrieved from January 14, 2022 https://docs.nvidia.com/clara/parabricks/3.7.0/index.html.

[5] Wilm et al. 2012. *Source Code Repository of LoFreq.* Retrieved from January 14, 2022 https://github.com/CSB5/lofreq.

[6] Xilinx. 2021. *Vivado Design Suite User Guide - Implementation.* Retrieved from April 7, 2022 https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug904-vivado-implementation.pdf.

[7] Xilinx. 2022. *Xilinx Alveo U250 Accelerator Card.* Retrieved from January 15, 2022 https://www.xilinx.com/products/boards-and-kits/alveo/u250.html.

[8] Xilinx. 2012. *Xilinx Large FPGA Methodology Guide.* Retrieved from April 7, 2022 https://www.xilinx.com/support/documents/sw_manuals/xilinx2012_3/ug872_largefpga.pdf.

[9] Mohammed Alser, Taha Shahroodi, Juan Gómez-Luna, Can Alkan, and Onur Mutlu. 2020. SneakySnake: A fast and accurate universal genome pre-alignment filter for CPUs, GPUs and FPGAs. *Bioinformatics* 36, 22–23 (12 2020), 5282–5290. DOI: https://doi.org/10.1093/bioinformatics/btaa1015

[10] James Arram, Thomas Kaplan, Wayne Luk, and Peiyong Jiang. 2017. Leveraging FPGAs for accelerating short read alignment. *IEEE/ACM Trans. Comput. Biol. Bioinform.* 14, 3 (May 2017), 668–677. DOI: https://doi.org/10.1109/TCBB.2016.2535385

[11] Subho S. Banerjee, Mohamed el Hadedy, Ching Y. Tan, Zbigniew T. Kalbarczyk, Steve Lumetta, and Ravishankar K. Iyer. 2017. On accelerating pair-HMM computations in programmable hardware. In *27th International Conference on Field Programmable Logic and Applications (FPL'17)*. 1–8. DOI: https://doi.org/10.23919/FPL.2017.8056837

[12] Zülal Bingöl, Mohammed Alser, Onur Mutlu, Ozcan Ozturk, and Can Alkan. 2021. GateKeeper-GPU: Fast and accurate pre-alignment filtering in short read mapping. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'21)*. 209–209. DOI: https://doi.org/10.1109/IPDPSW52791.2021.00039

[13] Pierre Blanchard, Desmond J. Higham, and Nicholas J. Higham. 2020. Accurately computing the log-sum-exp and softmax functions. *IMA J. Numer. Anal.* 41, 4 (08 2020), 2311–2330. DOI: https://doi.org/10.1093/imanum/draa038

[14] Daniel Butler, Christopher Mozsary, Cem Meydan, Jonathan Foox, Joel Rosiene, Alon Shaiber, David Danko, Ebrahim Afshinnekoo, Matthew MacKay, Fritz J. Sedlazeck, Nikolay A. Ivanov, Maria Sierra, Diana Pohle, Michael Zietz, Undina Gisladottir, Vijendra Ramlall, Evan T. Sholle, Edward J. Schenck, Craig D. Westover, Ciaran Hassan, Krista Ryon, Benjamin Young, Chandrima Bhattacharya, Dianna L. Ng, Andrea C. Granados, Yale A. Santos, Venice Servellita, Scot Federman, Phyllis Ruggiero, Arkarachai Fungtammasan, Chen-Shan Chin, Nathaniel M. Pearson, Bradley W. Langhorst, Nathan A. Tanner, Youngmi Kim, Jason W. Reeves, Tyler D. Hether, Sarah E. Warren, Michael Bailey, Justyna Gawrys, Dmitry Meleshko, Dong Xu, Mara Couto-Rodriguez, Dorottya Nagy-Szakal, Joseph Barrows, Heather Wells, Niamh B. O'Hara, Jeffrey A. Rosenfeld, Ying Chen, Peter A. D. Steel, Amos J. Shemesh, Jenny Xiang, Jean Thierry-Mieg, Danielle Thierry-Mieg, Angelika Iftner, Daniela Bezdan, Elizabeth Sanchez, Thomas R. Campion, John Sipley, Lin Cong, Arryn Craney, Priya Velu, Ari M. Melnick, Sagi Shapira, Iman Hajirasouliha, Alain Borczuk, Thomas Iftner, Mirella Salvatore, Massimo Loda, Lars F. Westblade, Melissa Cushing, Shixiu Wu, Shawn Levy, Charles Chiu, Robert E. Schwartz, Nicholas Tatonetti, Hanna Rennert, Marcin Imielinski, and Christopher E. Mason. 2021. Shotgun transcriptome, spatial omics, and isothermal profiling of SARS-CoV-2 infection reveals unique host responses, viral diversification, and drug interactions. *Nat. Commun.* 12, 1 (12 Mar. 2021). DOI: https://doi.org/10.1038/s41467-021-21361-7

[15] Mau-Chung Frank Chang, Yu-Ting Chen, Jason Cong, Po-Tsang Huang, Chun-Liang Kuo, and Cody Hao Yu. 2016. The SMEM seeding acceleration for DNA sequence alignment. In *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'16)*. 32–39. DOI: https://doi.org/10.1109/FCCM.2016.21

[16] Yu-Ting Chen, Jason Cong, Jie Lei, and Peng Wei. 2015. A novel high-throughput acceleration engine for read alignment. In *IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 199–202. DOI: https://doi.org/10.1109/FCCM.2015.27

[17] Harsha Doddapaneni, Sara Javornik Cregeen, Richard Sucgang, Qingchang Meng, Xiang Qin, Vasanthi Avadhanula, Hsu Chao, Vipin Menon, Erin Nicholson, David Henke, Felipe-Andres Piedra, Anubama Rajan, Zeineen Momin, Kavya Kottapalli, Kristi L. Hoffman, Fritz J. Sedlazeck, Ginger Metcalf, Pedro A. Piedra, Donna M. Muzny, Joseph F. Petrosino, and Richard A. Gibbs. 2020. Oligonucleotide capture sequencing of the SARS-CoV-2 genome and subgenomic fragments from COVID-19 individuals. *bioRxiv: The preprint server for biology* (27 July 2020), 2020.07.27.223495.

[18] Daichi Fujiki, Arun Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. 2018. GenAx: A genome sequencing accelerator. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. 69–82. DOI: https://doi.org/10.1109/ISCA.2018.00017

[19] Daichi Fujiki, Shunhao Wu, Nathan Ozog, Kush Goliya, David Blaauw, Satish Narayanasamy, and Reetuparna Das. 2020. SeedEx: A genome sequencing accelerator for optimal alignments in subminimal space. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*. 937–950. DOI: https://doi.org/10.1109/MICRO50266.2020.00080

[20] Nathan D. Grubaugh, Karthik Gangavarapu, Joshua Quick, Nathaniel L. Matteson, Jaqueline Goes De Jesus, Bradley J. Main, Amanda L. Tan, Lauren M. Paul, Doug E. Brackney, Saran Grewal, Nikos Gurfield, Koen K. A. Van Rompay, Sharon Isern, Scott F. Michael, Lark L. Coffey, Nicholas J. Loman, and Kristian G. Andersen. 2019. An amplicon-based sequencing framework for accurately measuring intrahost virus diversity using PrimalSeq and iVar. *Genome Biol.* 20, 1 (01 2019). DOI: https://doi.org/10.1186/s13059-018-1618-7

[21] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. 2019. Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between FPGA and GPU. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'19)*. 127–135. DOI: https://doi.org/10.1109/FCCM.2019.00027

[22] Sitao Huang, Gowthami Jayashri Manikandan, Anand Ramachandran, Kyle Rupnow, Wen-mei W. Hwu, and Deming Chen. 2017. Hardware acceleration of the pair-HMM algorithm for DNA variant calling. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 275–284.

[23] Bryce Kille, Yunxi Liu, Nicolae Sapoval, Michael Nute, Lawrence Rauchwerger, Nancy Amato, and Todd J. Treangen. 2021. Accelerating SARS-CoV-2 low frequency variant calling on ultra deep sequencing datasets. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'21)*. IEEE, 204–208.

[24] Joël Lindegger, Damla Senol Cali, Mohammed Alser, Juan Gómez-Luna, Nika Mansouri Ghiasi, and Onur Mutlu. 2022. Scrooge: A fast and memory-frugal genomic sequence aligner for CPUs, GPUs, and ASICs. *arXiv preprint arXiv:2208.09985* (2022).

[25] Yunxi Liu, Joshua Kearney, Medhat Mahmoud, Bryce Kille, Fritz J. Sedlazeck, and Todd J. Treangen. 2021. Rescuing low frequency variants within intra-host viral populations directly from Oxford Nanopore sequencing data. DOI: https://doi.org/10.1101/2021.09.03.458038

[26] Michael Lo, Zhenman Fang, Jie Wang, Peipei Zhou, Mau-Chung Frank Chang, and Jason Cong. 2020. Algorithm-hardware co-design for BQSR acceleration in genome analysis ToolKit. In *IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'20)*. 157–166. DOI: https://doi.org/10.1109/FCCM48280.2020.00029

[27] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Y. Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran V. Garimella, Dr. David Altshuler, S. Gabriel, Mark J. Daly, and Mark A. DePristo. 2010. The genome analysis toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Res.* 20 9 (2010), 1297–303.

[28] Corey B. Olson, Maria Kim, Cooper Clauson, Boris Kogon, Carl Ebeling, Scott Hauck, and Walter L. Ruzzo. 2012. Hardware acceleration of short read mapping. In *IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 161–168. DOI: https://doi.org/10.1109/FCCM.2012.36

[29] Ryan Poplin, Valentin Ruano-Rubio, Mark A. DePristo, Tim J. Fennell, Mauricio O. Carneiro, Geraldine A. Van der Auwera, David E. Kling, Laura D. Gauthier, Ami Levy-Moonshine, David Roazen, Khalid Shakir, Joel Thibault, Sheila Chandran, Chris Whelan, Monkol Lek, Stacey Gabriel, Mark J. Daly, Ben Neale, Daniel G. MacArthur, and Eric Banks. 2018. Scaling accurate genetic variant discovery to tens of thousands of samples. DOI: https://doi.org/10.1101/201178

[30] Jesse J. Salk, Michael W. Schmitt, and Lawrence A. Loeb. 2018. Enhancing the accuracy of next-generation sequencing for detecting rare and subclonal mutations. *Nat. Rev. Genet.* 19, 5 (01 May 2018), 269–285. DOI: https://doi.org/10.1038/nrg.2017.117

[31] Davide Sampietro, Chiara Crippa, Lorenzo Di Tucci, Emanuele Del Sozzo, and Marco D. Santambrogio. 2018. FPGA-based PairHMM forward algorithm for DNA variant calling. In *IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP'18)*. 1–8. DOI: https://doi.org/10.1109/ASAP.2018.8445119

[32] Arun Subramaniyan, Jack Wadden, Kush Goliya, Nathan Ozog, Xiao Wu, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2021. Accelerated seeding for genome sequence alignment with enumerated radix trees. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*. 388–401. DOI: https://doi.org/10.1109/ISCA52012.2021.00038

[33] Yatish Turakhia, Gill Bejerano, and William J. Dally. 2018. Darwin: A genomics co-processor provides up to 15,000X acceleration on long read assembly. *SIGPLAN Not.* 53, 2 (Mar. 2018), 199–213. DOI: https://doi.org/10.1145/3296957.3173193

[34] Pengfei Wang, Yuanwu Lei, and Yong Dou. 2019. Comparative analysis of FPGA-based pair-HMM accelerator structures. *Electronics* 8, 9 (2019), 965.

[35] Andreas Wilm, Pauline Poh Kim Aw, Denis Bertrand, Grace Hui Ting Yeo, Swee Hoe Ong, Chang Hua Wong, Chiea Chuen Khor, Rosemary Petric, Martin Lloyd Hibberd, and Niranjan Nagarajan. 2012. LoFreq: A sequence-quality aware, ultra-sensitive variant caller for uncovering cell-population heterogeneity from high-throughput sequencing datasets. *Nucleic Acids Res.* 40, 22 (2012), 11189–11201.

[36] Lisa Wu, David Bruns-Smith, Frank A. Nothaft, Qijing Huang, Sagar Karandikar, Johnny Le, Andrew Lin, Howard Mao, Brendan Sweeney, Krste Asanović, David A. Patterson, and Anthony D. Joseph. 2019. FPGA accelerated INDEL realignment in the cloud. In *IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*. 277–290. DOI: https://doi.org/10.1109/HPCA.2019.00044

[37] Peipei Zhou, Jiayi Sheng, Cody Hao Yu, Peng Wei, Jie Wang, Di Wu, and Jason Cong. 2021. MOCHA: Multinode cost optimization in heterogeneous clouds with accelerators. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'21)*. Association for Computing Machinery, New York, NY, 273–279. DOI: https://doi.org/10.1145/3431920.3439304

Received 2 November 2022; revised 2 February 2023; accepted 14 April 2023