2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)

k-core: Hardware Accelerator for *k*-mer Generation and Counting used in Computational Genomics

Simmi M Bose[§], Varsha S Lalapura[†], S Saravanan[§], Madhura Purnaprajna[§]
[§]Dept. of Computer Science and Engineering,
[†]Dept. of Electronics and Communication Engineering,
Amrita School of Engineering, Bengaluru,
Amrita Vishwa Vidyapeetham, India
Email: [§]ammu11kutty@gmail.com, [†]s_varshalalapura@blr.amrita.edu, [§]s_saravanan@blr.amrita.edu, [§]p_madhura@blr.amrita.edu

***Abstract*—In computational genomics, the term *k*-mer typically refers to all the possible subsequences of length *k* from a single read obtained through DNA sequencing. In genome assembly, generating frequency of *k*-mers takes the highest compute time. *k*-mer counting is considered as one of the important analyses and the first step in sequencing experiments. Here, we present an FPGA based fast *k*-mer generator and counter, *k*-core to generate unique *k*-mers and count their frequency of occurrence. The IP core is parameterizable in terms of the line length (*l*) and *k*-mer size (*k*) and is implemented on XCVU095 FPGA. A speed up of about 6.6×, 5.46× and 8.5×, are achieved compared to [1], [2], [3] CPU implementations respectively and about 5.26× with respect to [1]'s GPU version.**

***Index Terms*—*k*-mer, FPGA, genome assembly, acceleration**

I. Introduction

In bioinformatics, sequence assembly uses DNA sequences for reconstruction and usually consist of several terabytes of genome data. Since these data sizes are too large to handle, short fragments called reads are decomposed. Each read is further fragmented into smaller substrings called *k*-mers. Size of these fragments are defined by *k*-mer size *k*. Frequency of the unique *k*-mers in every read is significant in many bioinformatics applications like sequence assembling [4] and sequence detection. In case of the De-Novo assembly [5], multiple reads of the same region of the DNA [6] are accounted to make the assembling process error free at the cost of increasing problem size.

Generating and counting unique *k*-mers can be considered as a big data problem since the data set is very large. An exponentially large sequence of genome data cannot be ported onto implementation platforms with limited compute and memory resources. Many specialized tools and hardware (accelerators) are designed to handle it [1], [7–11]. The compute and memory requirements are intensive which makes *k*-mer counting challenging.

An overview of *k*-mer generation and counting process is described in Figure 1. It is essentially made up of 3 stages. At the first stage, read sequences from a huge genome database are converted into *k*-mers which is *k*-mer generation. At the next stage, unique *k*-mers are filtered. Finally, their frequency is obtained which is the number of times they occurred in the read.The number of *k*-mers that can be generated from a given read of length l is l-k+1. In a DNA sequence with A, C, T, G representing the proteins of the DNA, 4^k possible *k*-mers can be generated. The size of the *k*-mer influences the sequence assembly [12]. A *k*-mer of smaller size decreases the amount of storage of the DNA sequence. However, a large sized *k*-mer is beneficial for reconstruction of the gene.As an example, for a read size l = 12 and *k*-mer size k = 4, the total number of possible *k*-mers = 9. Set of unique *k*-mers can be as many as possible *k*-mers in the worst case. Counting the frequency of occurrence is primarily comparisons within the set and accumulating the true cases. Our contributions are:

- we present a novel, compact IP core for k-mer generation and counting all on the FPGA. We find the unique set of *k*-mers and their frequency of occurrence with parallel processing elements and a pool of parallel comparators,
- since sequence genome data is a large dataset, we encode and organize the data on the FPGA such that there exist memory and compute overlap,
- our IP core is parameterizable in terms of line length l and *k*-mer size k.

The paper is organized into following sections. Various state of the art *k*-mer counters with their pros and cons are discussed in II. Design of *k*-core is discussed in III. Experimental setup and results are discussed in section IV and lastly conclusion in section V.

II. Related Work

Implementation of *k*-mer counters on different hardware platforms with different approaches for counting are discussed here. First successful of its kind is the Jelly Fish [7], which is a lock-free hash table based approach to count *k*-mers. The algorithm is designed for shared memory parallel computers with more than one core. But, the design fails to count unique *k*-mers and runs with lower k values or shorter reads.

BFCounter [8], finds all the *k*-mers that occur more than once in a dataset. Bloom filters which are probabilistic data structures are used to count *k*-mers. The algorithm is a serial implementation on the CPU and *k*-mers counted are non-unique.

Gerbil [1] is an open source *k*-mer counting tool which again uses a hash table approach for the counting *k*-mers. The

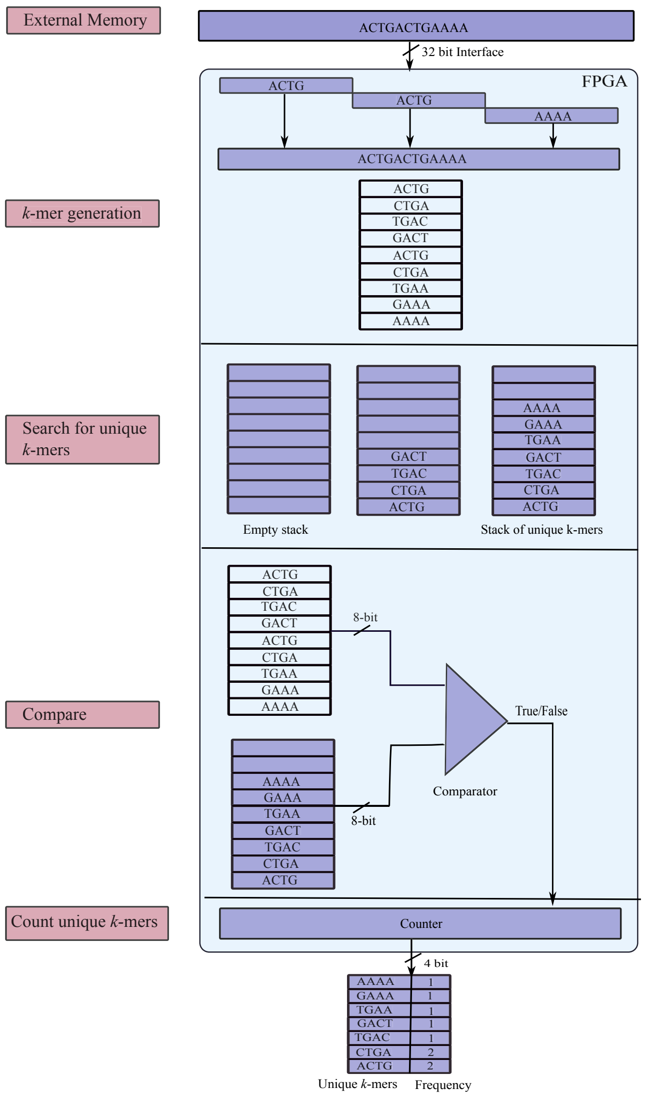

Fig. 1. *k*-mer generation and counting process

tool can operate for *k* value greater than 32. The counter is implemented on CPU with 4 threads. The algorithm is complex in terms of handling the dataset and work-flow.

Turtle [9] replaces a standard Bloom filter by a cache-efficient counterpart. The algorithm uses a complex sorting and compaction-based strategy against a hash table-based strategy for counting *k*-mer. Parallelization is a producer-consumer model. KMC2 [2] uses the sorting based counting approach which has been optimized for *k* lesser than 32. However, performance drops when *k* grows larger. The implementation is on CPU.

k-mer counting algorithm suitable for GPUs calculations based on sorting algorithm [10] is similar to Turtle. The algorithm is implemented on a CPU-GPU heterogeneous environment.

k-mer counting is implemented with the help of an FPGA-attached Hybrid Memory Cube (HMC) [11]. Bloom filters are used filtering out rarely occurring *k*-mers. HMC is considered as a replacement for traditional DRAM in FPGA. *k*-mer are generated external to the FPGA and task of counting *k*-mers uses Bloom Filters implemented on FPGA.

Our design is a compact hand-coded Verilog RTL design for generation and counting of unique *k*-mers all on the FPGA. FPGAs are powerful in terms of parallelization [13], [14]. We exploit parallelism in the design in 2 stages. One is *k*-cores that generate and count *k*-mers in parallel. Next, to speed up the execution, we use a pool of parallel comparators for counting.

III. Design of *k*-core: *k*-mer generator and counter

The design of *k*-core on FPGA is discussed below. Since the dataset is large, we encode and organize the data for processing. Parallel *k*-cores perform the task of *k*-mer generation and counting.

A. Data encoding and organization

k-mers are made up of a string of 4 proteins namely A, C, T and G. These 4 proteins are encoded in 2 bits i.e. A is encoded as “00”, C as “01”, T as “10” and G as “11”. The encoded data are decomposed into short fragments called reads in the external memory. They are read into the on-chip memory as shown in Figure 2.

B. *k*-core

k-core is the compute unit in which *k*-mers are generated and counted. In Figure 2, lines are read in parallel to corresponding *k*-core from the on-chip memory. Each *k*-core has a fast access register that stores one line of the sequence. Computations are carried out on this line. While computations are progressing, the next line is read to the corresponding fast access register in a pipelined fashion. Number of *k*-cores that are executed in parallel are dependent on *k* size. For smaller *k* sizes, many *k*-cores can be accommodated on the FPGA, but for higher *k* size values, fewer *k*-cores can fit. This is the stage one parallelism exploited in our design.

C. Serial *k*-core

A sequential design of *k*-mer generation and counting is as shown in Figure 3. Each *k*-core accesses one line from its fast access register and counts unique *k*-mers in the following way.

1) k-mer generation: A line is converted to set of *k*-mers based on a sliding window of size *k*. The window is moved by a stride of 2 bits to generate *k*-mers. They are stored in a memory called *k*-buf whose depth is total number of *k*-mers in the read and width is *k**2.

2) Unique k-mer search: *k*-mers are filtered such that a set of unique *k*-mers are stored in a buffer called *u*-buf whose width is *k**2 and depth is total number of *k*-mers generated. The search for unique *k*-mers is carried out in the following way.

The first *k*-mer in the k-buf is initially searched for in the empty buffer, u-buf. Since the buffer is empty, the first *k*-mer

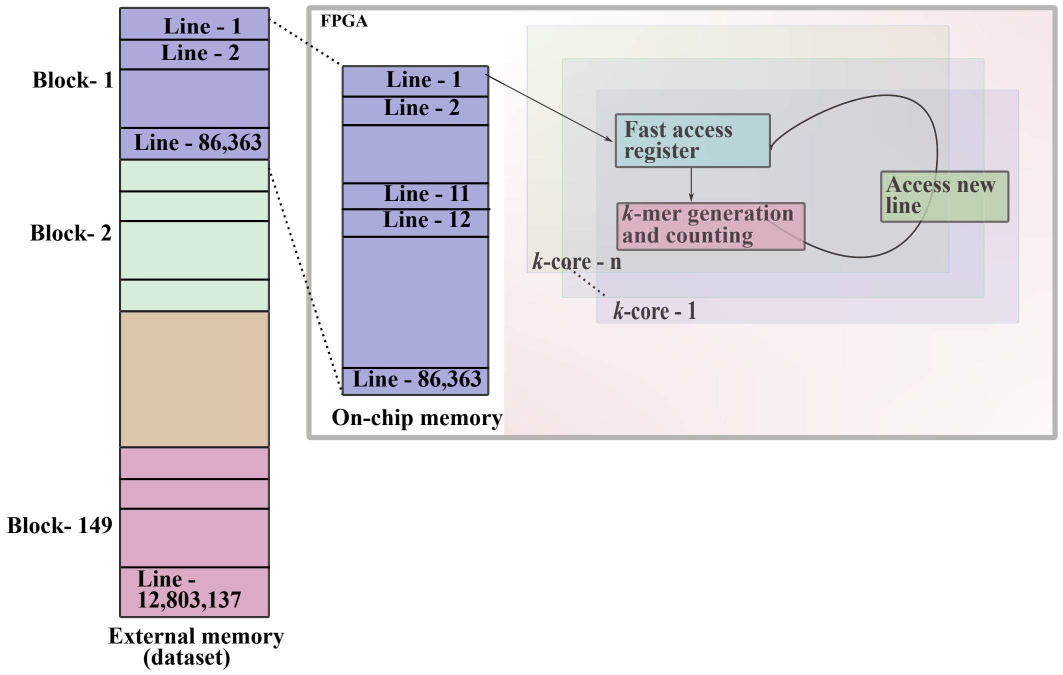

Fig. 2. Data-organization and parallel processing on FPGA: A block of genome sequence is transferred on-chip from which one line is accessed by its respective *k*-core for *k*-mer generation and counting

is added to the buffer. The next generated *k*-mer is searched for in the buffer updated in the previous iteration. If the *k*-mer matches the one in the buffer, it is discarded and the next *k*-mer is iterated. But if the *k*-mer does not match the one in the buffer, then this *k*-mer is stored in the buffer *u*-buf. This process is repeated until the *u*-buf has a unique set of *k*-mers.

3) k-mer counting: One comparator is used to obtain the frequency of *k*-mers in the read. The *k*-mer from *k*-buf is passed to one input of the comparator through a gate. The gate is placed such that it allows the *k*-mer for comparison only if it is unique. The other input of the comparator is all the *k*-mers in the *k*-buf given sequentially. In essence, it is a linear search between the generated *k*-mers. The output of the comparator at each compare step is stored and accumulated for the total sum.

The drawback of this design is that the comparison is a sequential process leading to more clock cycles to complete all the comparisons.

4) Parallel k-core: This is the stage two parallelism exploited in our design. The parallel design differs from its serial counterpart in terms of parallel comparisons taking place as shown in Figure 4. After reading a line to the fast access register, *k*-mers generated as in the serial design. Also, unique *k*-mers are filtered as mentioned before. But, we unroll the sequential loop and parallelize the comparisons such that the unique *k*-mers are compared against all in parallel to obtain their frequency of occurrence. The number of comparators required will be equal to the number of *k*-mers present in the read (if all *k*-mers happen to be unique) and all the comparisons are executed in parallel. The outputs of the comparators are stored and accumulated to obtain the final count of the unique *k*-mers.

Parallel *k*-core is designed as a finite state machine with 7 states as shown in Figure 5. The design is parameterizable in terms of line-size *l* and *k*-mer size *k*.

IV. Experiments and Results

In this section, we present the implementation details of *k*-core. The F.vesca dataset consist of 12,803,137 lines of genome data. While the FPGA can hold 60.8 Megabits on-chip, 86,363 encoded lines are stored on-chip. For a fixed line length *l*, varying number of *k*-cores are executed in parallel depending on the *k*-size to count the unique *k*-mers as shown in Figure 6. We choose *k* sizes as 28, 40 and 56 in the IP core in order to have a fair comparison with the existing implementations. The IP core is implemented on XCVU095 using the Xilinx Vivado 2015.3 Design Suite. Our current version of *k*-core does not account for reverse complement of proteins. Usually a DNA occurs as a double strand of proteins A, C, T, G and each A is paired with a T and vice versa, and each C is paired with a G and vice versa. The reverse complement is formed by interchanging A and T and interchanging C and G. However, this is an additional feature that does not affect performance and can be added as a future enhancement. Also, preprocessing of the F.vesca dataset has to be manually done in order to remove unwanted data.

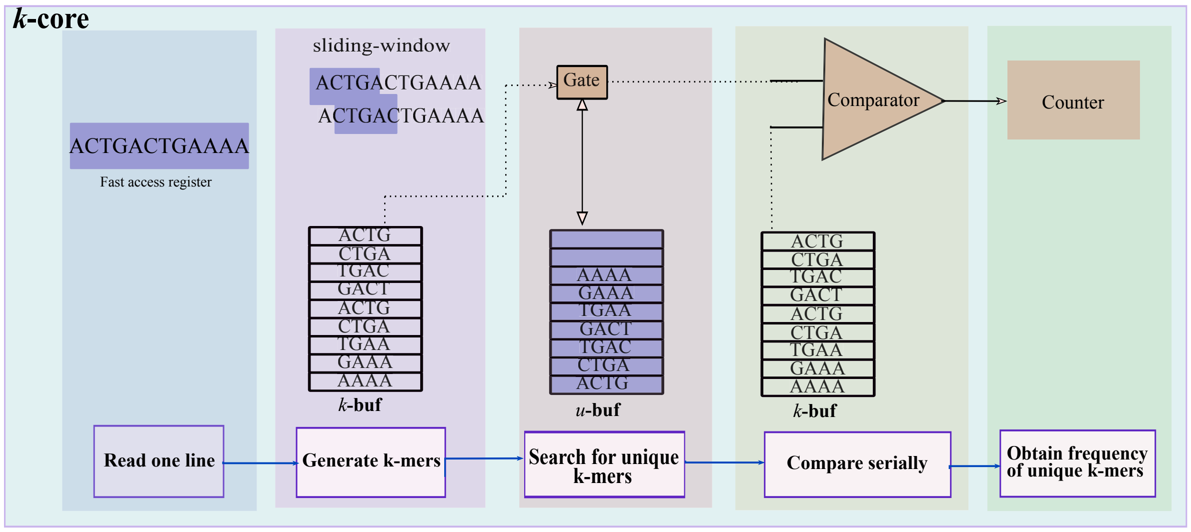

Fig. 3. Serial *k*-core: One line is accessed from the fast access register,*k*-mers are generated, unique *k*-mers are searched and counted in the *k*-core

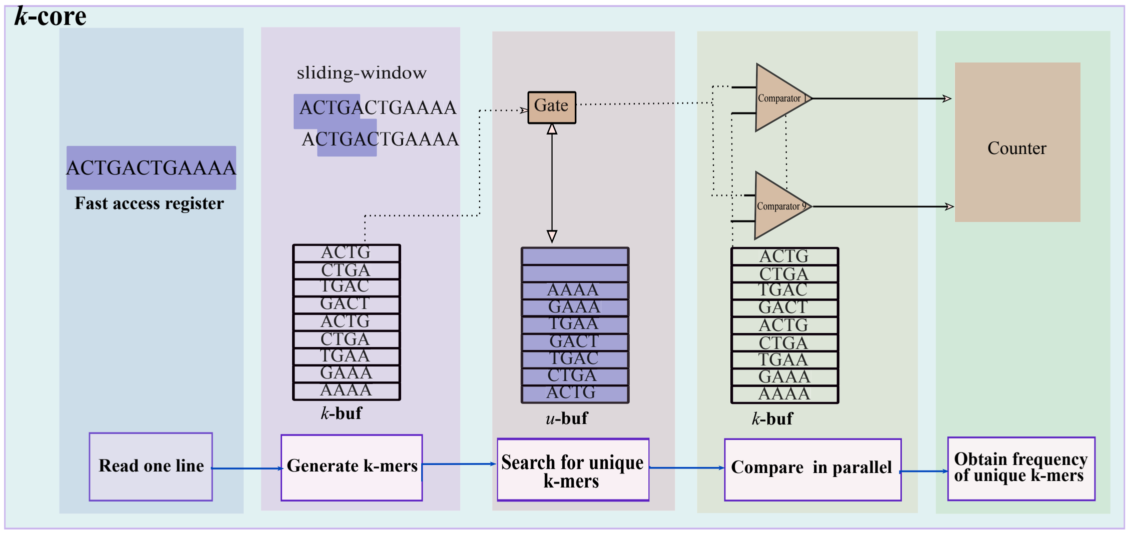

Fig. 4. Parallel *k*-core: One line is accessed from the fast access register,*k*-mers are generated, unique *k*-mers are searched and pool of parallel comparators in the compare state compare and count in the *k*-core

A. *Impact of k-mer size on frequency*

The design is synthesized and implemented for a fixed line length *l* and varying *k* size. The operating frequency is directly linked with the size of *k*-mer, which decides the width of the comparators and memory width (on-chip memory, *k*-buf and *u*-buf).The execution time increases for larger comparisons resulting in reduction in the maximum operating frequency as shown in Table I.

B. *Impact of k-mer size on execution time and resource usage*

For a fixed line size (l = 352), the impact of varying *k*-mer size is analyzed. It is observed that the total execution time increases with increasing value of *k*-mer, shown in Figure 7.

TABLE I
VARIATION OF *k*-MER SIZE ON OPERATING FREQUENCY

k	Max. Frequency (MHz)
28	83.33
40	71.42
56	66.66

As *k*-mer size increases, the number of *k*-mers generated are fewer but the memory width to store the *k*-mers increase. In addition, the width of each comparator also increases, resulting in higher delay per comparison. As a result, the critical path increases resulting in lower operating frequency, increasing

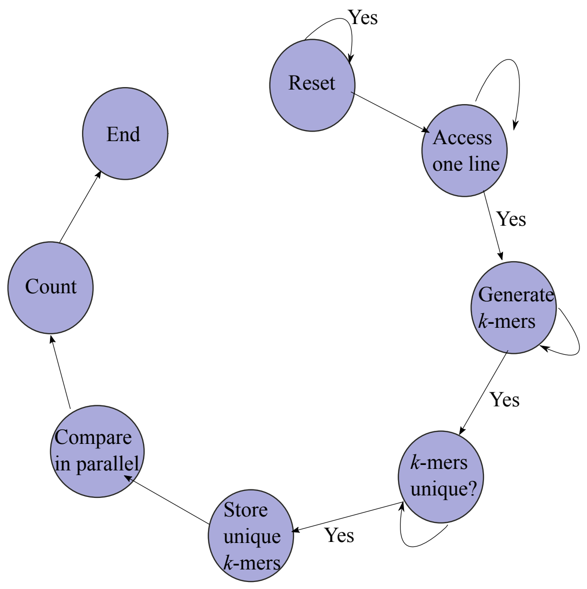

Fig. 5. FSM for parallel *k*-core

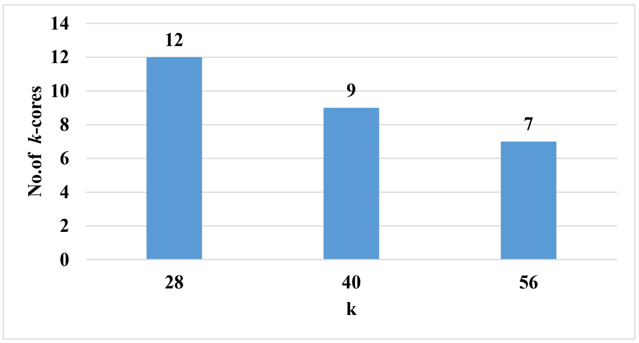

Fig. 6. *k*-core count with respect to *k*-mer size

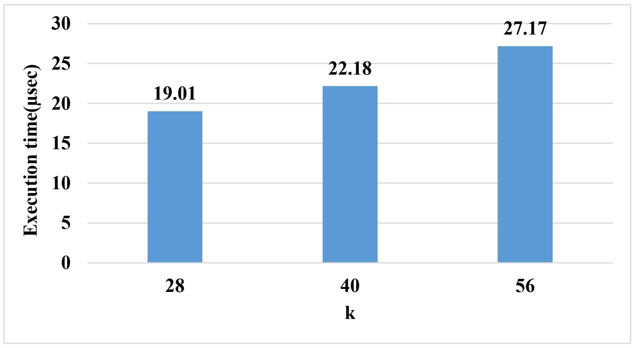

Fig. 7. Impact of *k*-mer size on execution time for 1 line

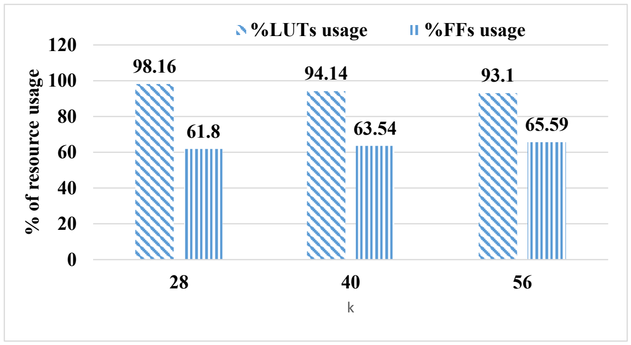

Fig. 8. Impact on resource usage with increasing values of k when all *k*-cores are executed in parallel

the overall execution time. For smaller *k*-mer sizes, parallel *k*-cores generated are many when compared to *k*-cores with larger *k*-mer size.Also when the *k*-mer size is smaller, even though the inputs are small, comparators required are many and for larger *k*-mer size, large inputs and fewer comparators. So the area required for smaller *k*-sized cores is greater compared to larger *k*-sized cores as seen in Figure 8.

C. Impact of data encoding

The *k*-mers are initially stored as ASCII values (8 bits per nucleotide) and later encoded to 2 bits per protein (section III A). The resource usage significantly reduced with this optimization as in Figure 9 and Figure 10.

D. Comparison with state-of-the-art

The recent implementations like Gerbil [1], KMC2 [2] and DSK [3] have been on the CPU. Gerbil also supports GPU version. Specifically, Gerbil design favors *k*-mers of *k* size greater than 32 whereas KMC2's sorting approach is applicable for *k* size less than 32. DSK uses hash table so the implementation is more promising for larger k. However, the techniques to count *k*-mers are far more complex than ours. In our IP core, *k*-mers are generated are on the FPGA and the design takes advantage of parallel *k*-cores and parallel comparisons due to the availability of parallel hardware. The IP core implemented uses only LUTs and FFs for *k*-mer counting and generation. We do not use BRAMs in the design since for the generation and counting of *k*-mers, multi-port RAMs are needed.

k-core achieves a speedup of around 6.6×, 5.46× and 8.5× compared to the Gerbil[1], KMC2[2], DSK[3] CPU implementations respectively for $l = 352$ and $k = 28$. With respect to Gerbil[1] GPU version, *k*-core achieves a speed up of about 5.26× for $k = 28$. The comparison with state of the art implementations are with respect to F.vesca dataset and is as shown in Table II. Here, the line size $l = 352$ is fixed.

E. Comparison with HLS

The high-level synthesis (HLS) implementation of *k*-core is shown in Table II. The c-code is implemented using Xilinx Vivado HLS 2015.3 on Intel i-7 CPU with 8 cores. The implementation is slower compared to the hand-coded RTL design by 13× for $l = 352$ and $k = 28$. The execution time reduces with increase in *k*-size, which is on account of reducing clock cycles at a fixed frequency of 250 MHz. The reduction the number of clock cycles required is due to fewer

TABLE II
PERFORMANCE COMPARISON WITH OTHER STATE OF ART AND HLS FOR THE ENTIRE F. VESCA DATASET

k	k-core (sec)	HLS (sec)	Gerbil-CPU [1] (sec)	Gerbil-GPU [1] (sec)	KMC2 [2] (sec)	DSK [3] (sec)
28	20.31	282.29	135	107	111	173
40	31.55	272.57	143	118	169	253
56	44.17	258.62	151	119	185	232

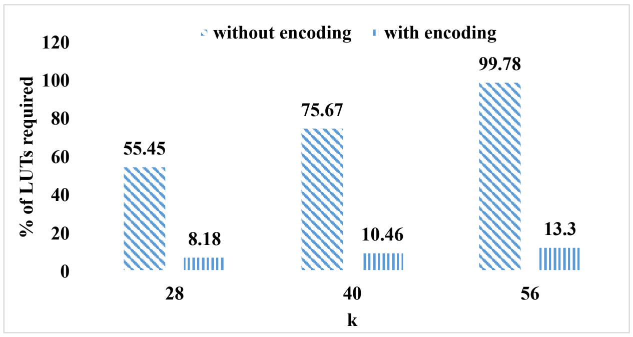

Fig. 9. LUT usage through binary encoding for 1 *k*-core

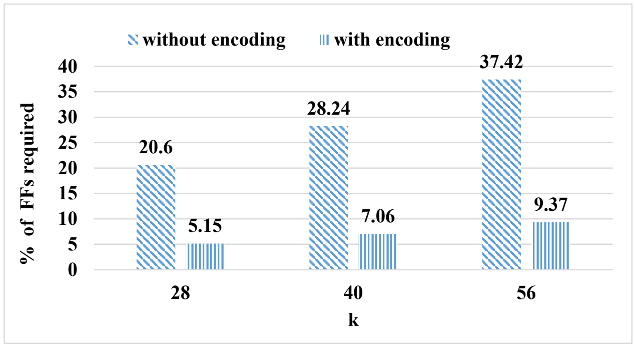

Fig. 10. FF usage through binary encoding for 1 *k*-core

k-mers generated with increasing size of *k*.

V. CONCLUSION

k-mer generation and counting is handled on the XCVU095 FPGA. *k*-mers generated are unique and their frequency of occurrence is measured using comparators as logic. Parallelizing the comparisons improve the operating frequency compared to a sequential design. Encoding the *k*-mers appropriately with 2 bits significantly reduces the resource usage. We present a compact and simple solution compared to existing Gerbil[1], KMC2[2], DSK[3] CPU implementations and Gerbil[1] GPU implementation.

VI. ACKNOWLEDGEMENT

The author (Madhura Purnaprajna) would like to acknowledge the support of funding by the Science and Engineering Research Board, India under the Early Career Research Award (ECR/2016/001765).

REFERENCES

[1] M. Erbert, S. Rechner, and M. Müller-Hannemann, "Gerbil: a fast and memory-efficient k-mer counter with gpu-support," *Algorithms for Molecular Biology*, vol. 12, no. 1, p. 9, 2017.

[2] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, "Kmc 2: fast and resource-frugal k-mer counting," *Bioinformatics*, vol. 31, no. 10, pp. 1569–1576, 2015.

[3] G. Rizk, D. Lavenier, and R. Chikhi, "Dsk: k-mer counting with very low memory usage," *Bioinformatics*, vol. 29, no. 5, pp. 652–653, 2013.

[4] J. R. Miller, S. Koren, and G. Sutton, "Assembly algorithms for next-generation sequencing data," *Genomics*, vol. 95, no. 6, pp. 315–327, 2010.

[5] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen *et al.*, "De novo assembly of human genomes with massively parallel short read sequencing," *Genome research*, vol. 20, no. 2, pp. 265–272, 2010.

[6] M. A. Quail, M. Smith, P. Coupland, T. D. Otto, S. R. Harris, T. R. Connor, A. Bertoni, H. P. Swerdlow, and Y. Gu, "A tale of three next generation sequencing platforms: comparison of ion torrent, pacific biosciences and illumina miseq sequencers," *BMC genomics*, vol. 13, no. 1, p. 341, 2012.

[7] G. Marçais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers," *Bioinformatics*, vol. 27, no. 6, pp. 764–770, 2011.

[8] P. Melsted and J. K. Pritchard, "Efficient counting of k-mers in dna sequences using a bloom filter," *BMC bioinformatics*, vol. 12, no. 1, p. 333, 2011.

[9] R. S. Roy, D. Bhattacharya, and A. Schliep, "Turtle: Identifying frequent k-mers with cache-efficient algorithms," *Bioinformatics*, vol. 30, no. 14, pp. 1950–1957, 2014.

[10] S. Suzuki, M. Kakuta, T. Ishida, and Y. Akiyama, "Accelerating identification of frequent k-mers in dna sequences with gpu," 2014.

[11] N. Mcvicar, C.-C. Lin, and S. Hauck, "K-mer counting using bloom filters with an fpga-attached hmc," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 2017, pp. 203–210.

[12] R. Chikhi and P. Medvedev, "Informed and automated k-mer size selection for genome assembly," *Bioinformatics*, vol. 30, no. 1, pp. 31–37, 2013.

[13] S. S. Vikram, V. Panty, M. Mody, and M. Purnaprajna, "Tilenet: Scalable architecture for high-throughput ternary convolution neural networks using fpgas," in *VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID), 2018 31st International Conference on*. IEEE, 2018, pp. 461–462.

[14] M. Belwal, M. Purnaprajna, and T. Sudarshan, "Enabling seamless execution on hybrid CPU/FPGA systems: Challenges & directions," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2015, pp. 1–8.