A Hardware/Software Co-Design of K-mer Counting Using a CAPI-Enabled FPGA

Abbas Haghi*, Lluc Alvarez*†, Jordà Polo*, Dionysios Diamantopoulos‡, Christoph Hagleitner‡, Miquel Moreto*†
*Barcelona Supercomputing Center (BSC), Barcelona, Spain
{abbas.haghi, lluc.alvarez, jorda.polo, miquel.moreto}@bsc.es
†Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
‡IBM Research Europe, Zurich, Switzerland
{did, hle}@zurich.ibm.com

***Abstract*—Advances in Next Generation Sequencing (NGS) technologies have caused the proliferation of genomic applications to detect DNA mutations and guide personalized medicine. These applications have an enormous computational cost due to the large amount of genomic data they process. Although leveraging FPGAs can improve the processing time of such amount of data, the limited memory capacity of FPGAs often restricts the potential gains. To overcome this limitation, IBM CAPI (Coherent Accelerator Processor Interface) supported platforms provide FPGAs with direct access to the CPU memory. This paper proposes a hardware/software co-design for k-mer counting, one of the most time-consuming phases of genomic applications. The proposed co-design targets CAPI-enabled FPGAs and is integrated into SMUFIN, a state-of-the-art reference-free method for finding DNA mutations. Results show that the proposed co-design outperforms the CPU-only design by a factor of 2.14×, it consumes 2.93× less energy, and it requires 1.57× less memory.**

***Keywords*-FPGA; co-design; acceleration; CAPI; genomics; k-mer; k-mer counting;**

I. Introduction

In the last years, the emergence of Next Generation Sequencing (NGS) platforms has opened the door to significant advances in the field of personalized medicine. Knowing the genome of the patients helps doctors in personalized treatment decision making, in which each patient takes specific medical treatments according to their individual characteristics or genetic information [1]. However, the computational cost of exploiting all the data generated with NGS is extremely high.

K-mer counting is a simple yet time-consuming phase of many genomics applications. One of the main challenges of k-mer counting is the huge amount of memory it requires, specially when dealing with large datasets like a complete human genome. Some methods have been proposed to speed up k-mer counting [2]–[14], being the reduction of the memory requirements one of the main goals of many works [2]–[8].

The computational capabilities of accelerators like FPGAs have persuaded engineers from different fields to propose hardware/software co-designs to accelerate their applications. However, the limited memory capacity of FPGAs is a crucial impediment to designing accelerators for applications that process massive amounts of data, as it is the case of k-mer counting. For this reason, in the last years chip manufacturers have put a lot of effort in developing communication protocols between processors and accelerators that provide accelerators with direct access to the processor memory. For instance, the IBM CAPI [15] protocol allows FPGAs to directly access the processor memory at a speed of up to 22GB/s. This feature makes CAPI-based systems ideal for accelerating workloads with large memory requirements such as genomics algorithms.

This paper presents a hardware/software co-designed accelerator for k-mer counting using a CAPI-enabled FPGA. The main contributions of this paper are: (i) an RTL design for FPGAs to accelerate the processes of extracting reads and generating k-mers; (ii) modifications to the software algorithm to remove dependencies between parallel threads, reduce overheads and use less memory; and (iii) 3 data compaction mechanisms to minimize the memory and disk requirements. We integrate the co-designed accelerator in an adapted version of SMUFIN [16], a state-of-the-art reference-free algorithm that identifies somatic mutations, and we evaluate it on a CAPI-enabled high-performance computing node with a dual-socket POWER9 processor and an FPGA. Results show that the co-design outperforms the CPU-only design by 2.14× while consuming 2.93× less energy and 1.57× less memory.

This paper is organized as follows: Section II introduces the background on genomic concepts, SMUFIN and k-mer counting, and Section III discusses the related work. Section IV presents the co-designed accelerator, which is evaluated in Section V. Section VI remarks the conclusions of this work.

II. Background

A. DNA Reads, K-mers and K-mer Counting

NGS technologies sample the genome and divide it into small pieces. These samples are called *reads* and their length is usually between 50 and 200 nucleobases [17]. DNA nucleobases are represented by one of the letters A, C, G or T, so a compact 2-bit numerical representation is often used.

A *k-mer* is a *K* length sub-string of a read, so a read of length R has *R-K+1* possible k-mers. K-mer frequency counting is a key step of many genomic algorithms such as genome assembly, error detection and variant calling.

B. SMUFIN Overview

SMUFIN [16] is a state-of-the-art application that detects DNA mutations by comparing normal and tumoral DNA

 DOI 10.1109/FPL50879.2020.00020

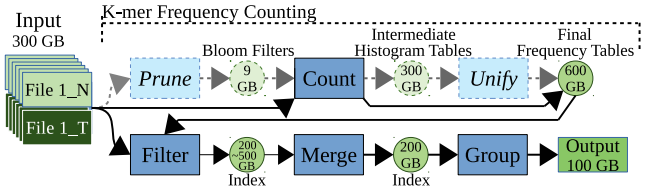


Fig. 1. SMUFIN mandatory and optional (light color) phases.

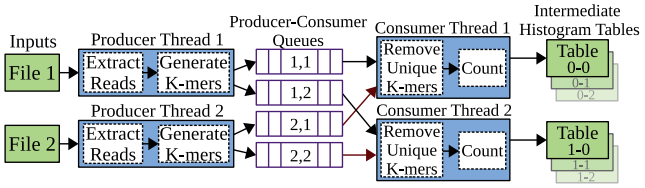


Fig. 2. Procedure of the count step.

samples from the same patient without requiring a reference genome. To do so, SMUFIN counts the frequency of the k-mers in the two sets of reads (normal and tumoral) separately and compares them to find imbalances that determine candidate DNA positions with mutations. SMUFIN processes the whole human genome, which is stored on multiple compressed files that require 200 to 400GB of disk storage.

The original SMUFIN algorithm [16] was designed to run on multiple nodes of a supercomputer with enough memory to store the whole data required for human genome analysis. Its successor, SMUFIN2 [18], rethinks the initial design to be more scalable and flexible. To do so, SMUFIN2 uses two levels of partitioning to distribute data among processes and threads. The first level splits data for processing into one or more partitions, and each one of these partitions can then be distributed either concurrently in multiple nodes or sequentially in a single node. In this paper we use an adapted version of SMUFIN2 where the first level partitions are based on batches of input files instead of a logical data distribution. For the second level partitions we use the same mechanism as SMUFIN2, which is based on the first 5-mers of a k-mer. In the rest of this paper we use the term SMUFIN to refer to our adapted version that partitions the data in batches of files, and partitioning refers to the second level of partitioning. The adapted version of SMUFIN2 is explained in the next subsections and it is also used as the CPU baseline in the evaluation presented in Section V.

Fig.1 shows a diagram of the SMUFIN application, which consists of 4 main phases: *count, filter, merge* and *group*. Unlike other applications, SMUFIN does not reconstruct genome sequences, yet like many *de-novo* assembly algorithms, its first phase is k-mer counting.

C. SMUFIN K-mer Counting Structure

K-mer counting is one of the most time-consuming phases of SMUFIN, up to 35% of the whole processing time [13]. Since the input of the count step is the whole human genome, it requires a huge amount of memory. For this reason, this algorithm is a very good candidate for being accelerated using CAPI-supported FPGAs, as they can access host memory directly without the need of making multiple copies of the genome data. To reduce the memory requirements of this step, 2 optional steps, *prune* and *unify*, can be used [13].

The *prune* step creates Bloom filters from all input files to identify and discard unique k-mers later in the count step. Bloom filters reduce the memory footprint of the count step by reducing the total amount of k-mers to be analyzed and the size of the intermediate histogram tables to be stored on disk. However, generating the Bloom filters is time-consuming as three hash functions are applied to each k-mer [3], [14].

The *count* step, due to the large memory requirements for the whole input, processes input files in batches. It reads a batch of input files, generates k-mers, and counts their frequency in the normal and in the tumoral inputs. Then it stores the k-mers and their two corresponding frequency counters on disk as intermediate histogram tables. Afterwards, it reads and analyzes the next batch of input files, repeating this process until all input files are analyzed. The resulting intermediate histogram tables are represented as hash tables with the format *pair<(uint64) k-mer, (uint16) counter[2]>*.

Fig. 2 shows the structure of the count step. This step has configurable numbers of producer and consumer threads, 2 of each type in the example. SMUFIN uses a k-mer partitioning mechanism that spreads k-mers among partitions based on their first 5 bases. The number of partitions can be configured between 1 and 128. Based on the statistical distribution of the k-mers, a lookup table is used to assign partitions to k-mers with the aim of distributing k-mers evenly among partitions. The number of producer threads is equal or less than the number of files in each batch, and the number of consumer threads is equal to the number of partitions.

Producer threads read data from one input file, extract reads, generate k-mers, convert k-mers to their numeric representation, check the lookup table to determine the partition where each k-mer belongs, and insert the k-mers into the producer-consumer queues of each partition. Consumer threads are in charge of counting the number of k-mers of each partition. To do so, they read k-mers from the producer-consumer queues, discard unique k-mers using the Bloom filters generated in the prune step, count the frequency of each k-mer, and populate the intermediate histogram tables that are stored on disk. The name of these tables on disk consists of two numbers, its partition and its index. For example, if there are 100 partitions and the count step analyzes 16 batches of files, the partition numbers are 0 to 99 and the indexes are 0 to 15.

The *unify* step merges all the intermediate histogram tables of each partition into one final frequency table for that partition. The intermediate histogram tables are sequentially loaded from disk and, for each partition, the final frequency table is updated. Final frequency tables are represented as hash tables with a different layout than the intermediate histogram tables, so the unify step converts them to the final layout that is suitable for next steps. Finally, the unify step re-checks all the final frequency tables to discard unique k-mers that can appear due to false positives in the Bloom filters.

D. Coherent Accelerator Processor Interface

CAPI developed by IBM provides a coherent access to the host memory for accelerators (FPGAs), avoiding the performance and storage overheads of having multiple copies of data. Three versions of CAPI have been released so far. CAPI1 and CAPI2 work on top of the PCIe communication bus, while OpenCAPI, which is the third version of CAPI, uses a customized communication link. For data sizes of more than 100MB, the data transfer speed from the FPGA BRAM memory to the host memory and vice versa is 3.3, 13 and 22GB/s for CAPI1, CAPI2 and OpenCAPI, respectively.

III. Related Work

A lot of effort has been put on designing memory efficient k-mer counting algorithms. Jellyfish [2] presents an algorithm using a lock-free hash table that minimizes parallelization overheads. BFCounter [3] proposes using a Bloom filter to discard unique k-mers and reduce the size of the histogram tables. The concept of partitioning is introduced in DSK [4], KMC [5] and MSPKmerCounter [6], which are all disk based algorithms. Among them, KMC uses a sorted mechanism instead of hash tables. Some other works such as Turtle [7] and KMC2 [8] have combined these ideas to improve performance.

Many works propose using GPUs and FPGAs to speed up the k-mer counting algorithm. Gerbil [9] is an open source k-mer counting software implementation with GPU support. Li et al. [10] port the k-mer counting algorithm of KCM2 to a GPU. McVicar et al. [11] and Wertenbroek et al. [12] offload the Bloom filter generation and the k-mer counting to a cloud of FPGAs with Hybrid Memory Cube. Cadenelli et al. modify the memory layout of k-mer counting and present an OpenCL co-design targeting GPUs [13] and FPGAs [14]. Compared to our work, other FPGA k-mer counting accelerators [11], [12], [14] use Bloom filters and hash tables, while we eliminate Bloom filters, accelerate k-mer generation, and use sorted vectors. Apart from k-mer counting, other genomic applications [19], [20] use co-designed accelerators for the alignment and assembly of long reads.

CAPI-supported platforms have also been used to accelerate other kind of memory intensive workloads. Sefat et al. [21] exploit CAPI to offload matrix-multiplications of a deep neural network to FPGAs, Chen et al. [22] accelerate a medical ultrasound imaging algorithm using a CAPI-enabled FPGA, and Castellane et al. [23] propose a co-designed accelerator for SHA3 cryptographic key calculations.

IV. K-mer Counting Accelerator Method

This paper presents a hardware/software co-design for k-mer counting. The proposed design consists of restructuring the count step to enable the usage of the FPGA and improving the unify step. The accelerated count phase is compatible with the rest of SMUFIN phases, which are left unaltered, and could also be integrated in other genomic applications.

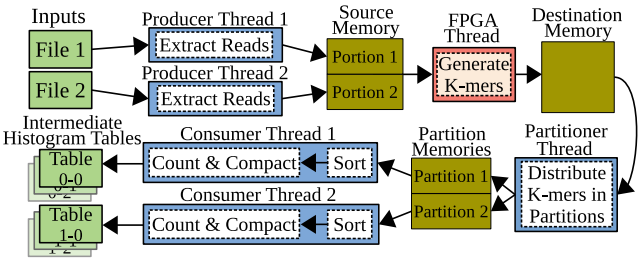


Fig. 3. Proposed procedure of the count step.

A. Prune Step

The proposed accelerator does not use Bloom filters, so the prune step is not needed. As explained in Section II-C, the Bloom filters created in the prune step reduce the memory and disk requirements by reducing the amount of k-mers to be analyzed. This allows the count step to use larger batches of input files and to generate smaller intermediate histogram tables, which decreases the memory usage and execution time of the unify step. The proposed design skips the prune step at the cost of using more disk space, and the count and unify steps are modified to use less memory and perform faster even without Bloom filters and for smaller batches of input files. Removing the unique k-mers is postponed to the unify step.

B. Count Step

Fig. 3 illustrates the proposed design for the count step. The main changes with respect to the original design consist of offloading the generation of k-mers to the FPGA, adjusting the behavior of the producer and consumer threads, and using a sort mechanism instead of hash tables in the consumer threads.

The producer-consumer queues of the original count step create inter-thread dependencies. The proposed design uses *partition memories* to pass data from producers to consumers instead of queues. The producer and FPGA threads first generate all the k-mers for a batch of input files, then one partitioner thread stores them in the partition memories, and finally the consumer threads start working. Each partition belongs to only one consumer, minimizing thread dependencies.

1) Producer Threads: Producer threads read the input files, extract the reads, and write them into the source memory. The reads in the source memory are marked as normal or tumoral so that the FPGA can determine their type. To avoid adding storage overheads, we use the first byte of each read to encode its type. If the read belongs to a normal file the first base is written as A, C, G, T or N (for unknown), otherwise the first read is incremented by 1, so it becomes B, D, H, U or O.

To maximize parallelism and avoid synchronization overheads, a configurable number of producer threads read small parts of different files and fill different portions of the source memory. Note that, when a producer thread finishes reading a file, the extracted reads may not be enough to fill the entire portion of that thread's source memory. When this happens, the producer thread fills the remaining space with 'S' reads indicating to the FPGA that these reads have to be skipped.

2) FPGA Design: The FPGA design is controlled by an FPGA thread. The FPGA thread continuously checks the status

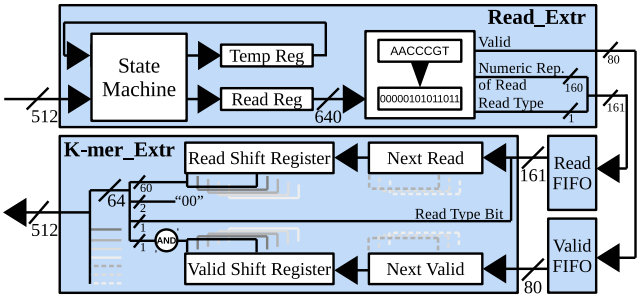


Fig. 4. Block diagram of FPGA k-mer generator modules.

of producer threads and, when they all have filled their portion of source memory, triggers the FPGA design. The FPGA design reads data from the source memory, generates the k-mers, and writes them into the destination memory. Fig. 4 shows a block diagram of the FPGA design, which consists of two main modules: the *Read_Extr* module extracts reads from the input data and encodes them with a 2-bit representation, and the *K-mer_Extr* module generates k-mers.

The Read_Extr module is responsible for extracting reads from input words. CAPI2 defines that the data width of the FPGA is 64 bytes. Since the read length is 80 bases (bytes) for our input data set, the reads are split between input words. The *state machine* determines, in each state, what portion of the incoming input data belongs to one read and what portion to the next read. The incomplete reads are stored in *Temp Reg* and, once completed, they are put in *Read Reg*.

The Read_Extr module adds 1 bit to each read to define its type. The type is determined from the first base of each read, as explained previously. If the first base is A, C, G, T or N, the type bit is set to 0 for normal. Otherwise, it is set to 1 for tumoral and the first base is converted back to A, C, G, T or N. The read and its type are stored in the *Read_FIFO* in 161-bit words (80 bases/read $\times$ 2 bits/base + 1 type_bit).

The Read_Extr module also identifies k-mers with unknown bases, which are represented as 'N' in the input data. These k-mers are marked as invalid and are omitted later by the partitioner thread. To do so, a valid bit is assigned to each base of a read and is set to 1 if the base is not 'N'. The valid bits of the reads are stored in a separate *Valid_FIFO* with a width of 80 bits, so each read in the Read_FIFO has a corresponding entry in the Valid_FIFO. In addition to unknown bases, the Read_Extr module handles reads with all their bases set to 'S', which is the format that producer threads use to specify invalid entries in the source memory. These reads are skipped by not writing them in the Read_FIFO nor in the Valid_FIFO.

The next FPGA module is the K-mer_Extr, which is in charge of generating the k-mers. This module reads data from the Read_FIFO and separates the first bit, which contains the type of the read (tumoral or normal), from other bits that contain the read itself. In this FPGA design the k-mer length is set to 30. As the data width of the FPGA is 64 bytes, 8 k-mers (each 8 bytes) of the read in the *Read Shift Register* are picked during each clock cycle, concatenated with the type bit of their read and their corresponding valid bits, and then written to the destination memory. The valid bit is obtained by an AND operation on the valid bits in the *Valid Shift Register*, which correspond to all the bases of the k-mer. During the same clock cycle, the Read and the Valid Shift Registers are shifted 16 and 8 bits, respectively. If in a given clock cycle the Read Shift Register contains less than 8 k-mers, k-mers of the *Next Read* are generated to ensure a total of 8 k-mers are processed in each clock cycle, and the Next Read and the Valid Read registers are shifted accordingly.

The FPGA design has two pipeline stages, one for each module. Although Read_Extr can analyze 64 bytes of data at each clock cycle, the K-mer_Extr module is the bottleneck because the output data of this module is 5$\times$ larger than its input data. With data width equal to 64 bytes at the output side, only 1/6 of the output data of the K-mer_Extr module is sent to the destination memory at each clock cycle. Therefore, the Read and Valid FIFOs can become full and impose a wait time on the Read_Extr module. With a clock frequency of 250MHz the peak bandwidth is 16GB/s, or 2G*k-mer*/s nominally, but in practice 1.7G*k-mer*/s is obtained.

3) Partitioner thread: When the FPGA action finishes, a partitioner thread in the CPU checks the destination memory, terminates invalid k-mers, determines the partition of each k-mer, and copies the k-mers to their corresponding partition memory. Meanwhile, the producer threads start loading the rest of the reads in the source memory.

4) Consumer Threads: The consumer threads count the frequency of the generated k-mers in partition memories. Each consumer is responsible for one partition memory. Each consumer thread first sorts and then counts the k-mers of its partition. After sorting, the counting is easily accomplished by comparing each k-mer with the previous one and, if they match, incrementing the frequency counter of the k-mer. After calculating the frequency of all the k-mers, consumer thread stores its intermediate histogram table on disk.

The benefits of using sort instead of hash tables, in addition to improved performance, are the lower requirements for memory and disk. The Google sparse hash tables used in the CPU-only version need 4 extra bytes per item [13]. So in that design, to avoid writing extra bits on disk, a loop iterates on all the elements of the hash tables and their values are extracted and copied in another part of memory that is then saved on disk. This adds extra time and memory usage that is eliminated in the proposed design. As there are no hash tables in our design, k-mers and their corresponding counters are directly put in a byte aligned array of memory.

The performance of writing the intermediate histogram tables to disk heavily depends on the speed of the I/O subsystem. Hence, compacting the data can lead to better performance and less storage requirements.

5) Data Compaction: As explained in Section II-C, the intermediate histogram tables keep two frequency counters per k-mer in a format of *pair<(uint64) k-mer, (uint16) counter[2]>*. The consumer threads use three techniques to compress the information of these tables.

The first technique consists of grouping the batches of files by their type, so that a batch only contains normal or tumoral

files. With this technique all the k-mers of the batch of files have the same type so, instead of two frequency counters per k-mer (one tumoral and one normal), only one counter per k-mer is required. First we feed the system with normal files and then with tumor files. As the number of normal and tumoral files is the same, the type of each k-mer is easily distinguishable from the index of its table name in later steps. If the index of a table is in the first half of all indexes, that table contains normal k-mers, otherwise it contains tumoral k-mers.

The second technique is based on the observation that, in practice, the frequency of 99% of the k-mers is below 255. To exploit this, we use a variable length counter controlled by a bit *cntr_ex_en* (counter extension enable) per k-mer that indicates the size in bytes of the counter field (0 for one byte and 1 for two bytes). This compaction method is applicable when the k-mer field contains at least 1 unused bit. For example, with a k-mer field of 64 bits, this method is suitable for k-mer lengths of 31 and lower, where at most 62 bits of the k-mer field are used (31 bases $\times$ 2 bits/base) and at least 2 bits are free and can be used to encode the *cntr_ex_en* bit.

The third technique reduces the space required to store one k-mer. This is achieved by replacing the first 5-mer of each k-mer with its index in the partitioning lookup table. This technique is only applicable when the number of partitions is more than 35. In this case each partition will contain less than 32 5-mers, so the index number of each 5-mer in the lookup table will be in the range of 0 to 31. Therefore, a 10-bit 5-mer is replaced by its 5-bit index. In our FPGA design the k-mer length is fixed to 30 and each k-mer occupies 8 bytes (61 bits = 30 bases $\times$ 2 bits/base + *Cntr_ex_en*). With this data compaction method, each k-mer fits in 56 bits (61 bits_before_compaction - 10 bits/5-mer + 5 bits/index). Note that this method is only useful when the number of saved bits reduces the k-mer field by one byte. For example, with a k-mer length of 31, this method is not useful because 58 bits are needed after compaction, which still requires 8 bytes.

These three compaction methods allow to save 30% of the space needed for storing the k-mers and their frequency counters. In the proposed design, hash table containers are no longer used. Instead, intermediate histogram tables are written sequentially in a byte aligned array of memory. In this array, each k-mer occupies 7 bytes and its corresponding frequency counter occupies one or two bytes, depending on its value.

C. Unify Step

The unify step has three responsibilities: merging all the intermediate histogram tables of each partition into a final frequency table of that partition, discarding unique k-mers, and converting the final frequency tables layout. The proposed unify step loads only the first k-mer of all intermediate histogram tables in memory, and determines the smallest k-mer of the loaded ones for each partition. If two or more k-mers are equal to the smallest one, their values are added together and stored in the final frequency table while, it is discarded if it is unique. Then the next k-mers of the tables are loaded and this procedure continues for all the k-mers of all the intermediate histogram tables.

The new unify design has four major changes compared to the CPU version: (i) it merges all intermediate histogram tables at once; (ii) it only loads the first k-mers of each table in memory; (iii) it removes unique k-mers before inserting them into the final frequency tables; and (iv) it stores final frequency tables as sorted vectors. The first modification makes the design faster while the others make it more memory efficient.

TABLE I
FPGA RESOURCES UTILIZATION (%) FOR THE CAPI-RELATED IP CORES AND FOR THE PROPOSED K-MER COUNTING ACCELERATOR.

	LUT	FF	DSP	BRAM	URAM
CAPI	10.04	9.86	1	32.99	-
K-mer counting	11.25	0.63	-	2.57	-
Total	21.29	10.49	0.04	35.56	0

V. EVALUATION

A. Experimental Setup

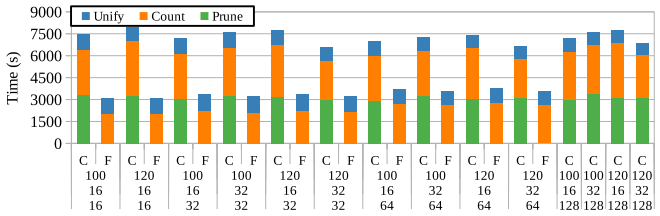
The experiments are done on a POWER9 system with an attached FPGA card. The POWER9 has 2 sockets with 20 cores running at 2.3GHz. Each core has 4 threads so the platform provides 160 threads in total. The 512GB RAM of the system consists of 16 32GB DDR4 DIMMs running at 2666MHz, and two 2TB Micron SATA SSDs are used as storage. The FPGA board is a CAPI2 enabled AlphaData ADM-PCIE-9V3. This board utilizes a Xilinx Virtex UltraScale Plus XCVU3P-2 (FFVC1517) FPGA with 394k lookup tables (LUTs), 788k Flip-Flops (FF), 2280 DSPs, 25.3Mb block RAMs (BRAM) and 90Mb ultra RAMs (URAM). The FPGA code is written in VHDL and compiled using Vivado v2018.1.

The input DNA samples are a customized human genome based on the Hg19 reference. Random somatic variants including single-nucleotide polymorphism and single-nucleotide variants with random insertions, deletions and inversions are applied to the input. In silico sequencing is simulated using ART Illumina21. The total input size is 312GB and consists of 256 gzip compressed FASTQ files, 128 for normal samples and 128 for tumoral. The read length is 80bp so each input file contains approximately 16M reads. To increase the speed of reading and writing files, half of input and output files are saved on one disk and the other half on the other disk.

In the evaluation we use the adapted version of SMUFIN as CPU baseline, as explained in Section II. SMUFIN accepts k-mer lengths between 24 and 32. We use a k-mer length of 30 in both the CPU baseline and the FPGA co-design.

B. Results

Table I shows the resource utilization of the FPGA. In this design the FPGA runs at a frequency of 250MHz. The k-mer counting accelerator occupies only a small fraction of the FPGA resources, while the CAPI modules present a higher resource utilization, specially in FF and BRAM. Even consuming few resources, the proposed accelerator reaches the maximum bandwidth of CAPI2, so a more complex design would not provide any benefit. For future generations of CAPI



Configuration set. From top to bottom: design (C: CPU-only / F:CPU+FPGA), number of consumer threads, number of producer threads, batch size.

Fig. 5. Execution time for different designs and configurations.

that provide more bandwidth, like OpenCAPI, there is room to scale up the proposed design and utilize more FPGA resources.

The execution time of both designs is summarized in Fig. 5. In this exploration we try all the possible combinations of batch sizes (16, 32, 64, 128), producer threads (16, 32) and consumer threads (100, 120). Other values for these parameters reduce the performance for both designs. The CPU+FPGA design results for a batch size of 128 are not shown because the system runs out of memory in this configuration. The CPU-only design uses producer-consumer queues of 32K elements as it gives the best performance. The best configuration for the CPU+FPGA design is 120 consumer threads, 16 producer threads and batches of 16 files, while for the CPU-only design it is 120 consumer threads, 32 producer threads and batches of 32 files. Comparing the best execution times of both designs, the CPU+FPGA design presents a speedup of 2.14×.

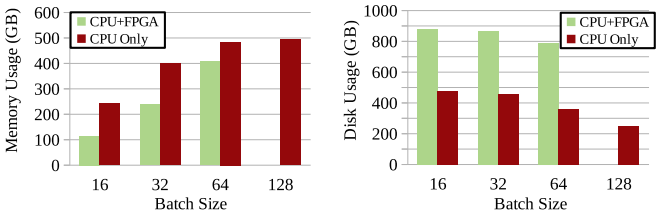
Fig. 5 also shows the distribution of the execution time among the different steps. The CPU+FPGA design does not use Bloom filters, so the time-consuming prune step is not needed. Even without Bloom filters, the count step is 1.34× faster by offloading work to the FPGA. Although the unify step of the CPU+FPGA design merges larger tables, its execution time is similar to the unify step of the CPU-only design with the prune step enabled.

Results show that the count step of the CPU+FPGA design presents less variability in the execution times when changing the number of producer and consumer threads. For a batch size of 32 files, the performance difference between the fastest and the slowest configuration in the CPU+FPGA design is 5.94%, while for the CPU-only design it is 24.59%. The higher variability of the execution times of the CPU-only design is due to the synchronization between producer and consumer threads in the producer-consumer queues. If the number of threads is not tuned properly, producer and consumer threads experience idle time when the queues are full or empty, negatively affecting performance. In contrast, the CPU+FPGA design decouples the operation of the two types of threads, resulting in a more stable performance.

In addition to the proposed FPGA co-design, we evaluate different design alternatives to try to further increase performance. One approach is to offload the partitioner thread to the FPGA, but we observe this provides negligible speedup and it requires to reprogram the FPGA when the number of partitions changes. We also try to offload the work of the consumer threads to the FPGA, but results show that performing this work using 160 parallel threads is faster than doing it in a single FPGA. Finally, to observe the effect of the FPGA acceleration on the execution time of the co-design, we replace the FPGA design by an analogous CPU code and we observe that the k-mer generation and the whole count step are slowed down by 150× and 10×, respectively.

TABLE II
EXECUTION TIME AND DISK SPACE REQUIREMENTS OF THE COUNT AND UNIFY STEPS WITH THE PRUNE STEP ENABLED AND DISABLED.

Prune Status	Time (s)				Disk (GB)
	Prune	Count	Unify	Total	
Enabled	3335	3049	1050	7434	475
Disabled	-	3994	16323	20317	1101



(a) Memory (b) Disk

Fig. 6. Memory and disk usage of the count step.

As explained in Section II-C, the prune step that generates the Bloom filters for the count step is optional. To quantify the impact of the prune step, Table II compares the execution time of the whole k-mer counting phase with and without the prune step for the CPU-only design. This experiment uses 16 producer threads, 100 consumer threads, and batches of 16 files. The system runs out of memory for batches with more than 16 files when the prune step is disabled. Results show that enabling the prune step improves performance by 2.73× because, although the execution of this step takes 3335 seconds, it provides important speedups in the count and unify steps. With the prune step enabled, the count step avoids the computation of unique k-mers, which reduces the execution time of this step by 1.31× and makes the intermediate histogram tables smaller. The unify step takes 15.54× more time when the prune step is disabled because it needs to merge larger tables, and the execution time of this step grows exponentially with the size of tables to be merged. The unify step can merge half of the intermediate histogram tables at once when the prune step is enabled, while without the prune step only 1/12 of these tables fits in memory at once. When the prune step is enabled the required disk space is 475GB while, with the prune step disabled, it reaches 1101GB. This shows that 57.86% of the required disk space is occupied by unique k-mers which do not provide useful information.

The number of files in each input batch affects disk and memory usage. Fig. 6 illustrates the memory and disk storage requirements of the count step for different batch sizes. The memory usage of the prune and unify steps are not dependent on the number of files in each batch and are almost constant. The prune step uses 90GB of memory, while the unify step requires 470GB of memory with the CPU-only design and 300GB of memory with the CPU+FPGA design, i.e., 1.57×

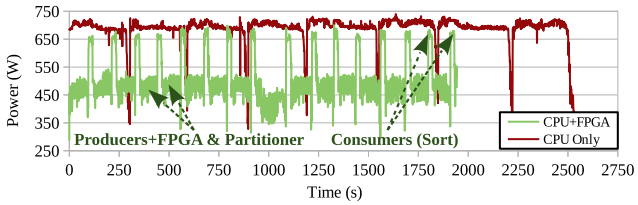


Fig. 7. Power consumption of the count step over time.

less memory than the CPU-only design. As seen in Fig. 6(a), the memory usage of the count step scales up with the batch size and the CPU+FPGA design needs less memory for any batch size below 128. However, its count step does not fit in the system memory for batches of 128 files. Regarding the best performing configurations (from Fig. 5), the CPU+FPGA design needs 114GB of memory while the CPU-only design requires 400GB. Regarding the disk utilization, Fig. 6(b) shows that, as the batch size increases, the disk usage decreases in both designs. This happens because, with larger batches, more files are merged together at once and more data is packed in the intermediate histogram tables, which reduces the required disk space. In conclusion, the CPU+FPGA design is faster and its count step requires 3.51× less memory. These memory and performance improvements are achieved at the cost of using 1.93× more disk space, 881GB in the CPU+FPGA design versus 456GB in the CPU-only design.

The power consumption of the count step is illustrated in Fig. 7. We measure the power consumption of the whole node with in-band readings from Linux to the OCC (On Chip Controller) [24]. It can be observed that the power consumption of the producers is lowered in the CPU+FPGA design by more than 200W. Peaks in the CPU+FPGA design belong to the sort part of consumer threads. There are 16 of them as the total input is analyzed in 16 batches. The power consumption of the unify step of the CPU+FPGA design, which is not shown in Fig. 7, is also 170W below that of the CPU-only design. All together, for the whole count phase, the CPU+FPGA design consumes 0.42kWh of energy while the CPU-only one uses 1.23kWh, so the total energy-to-solution is improved by a factor of 2.93×.

VI. Conclusions

K-mer counting is one of the most time-consuming phases of many genomic applications. Although FPGAs are very well suited to accelerate the k-mer counting algorithm, their reduced memory capacity is a big limiting factor to handle the vast amount of data that is processed with this algorithm. To overcome this limitation, the IBM CAPI interface allows FPGAs to directly access the processor memory.

This paper presents a hardware/software co-designed accelerator for k-mer counting on CAPI-enabled FPGAs. The proposed approach consists of an FPGA design to accelerate the generation of k-mers and a combination of optimizations on the software side to eliminate thread dependencies, replace hash-tables for sorted vectors, and re-define the memory layout using three data compaction mechanisms. The co-designed accelerator is able to efficiently count k-mers and unify the results without the need of Bloom filters, so the time consuming phase to generate them can also be avoided. All together, the proposed co-design greatly accelerates the k-mer counting algorithm while reducing the memory requirements, the thread synchronization overheads, and the sensitivity to the algorithm parameters. Results show that the presented co-design achieves a speedup of 2.14× over the CPU-only design while reducing the energy-to-solution and the memory requirements by 2.93× and 1.57×, respectively. The proposed co-design can be easily integrated in the k-mer counting phase of any genomic application and can be scaled up in future CAPI-based systems that provide higher bandwidth between the FPGA and the processor memory.

Acknowledgment

This work has been supported by the European HiPEAC Network of Excellence, by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P), by the Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328), and by the IBM/BSC Deep Learning Center initiative. Ll. Alvarez has been supported by the Spanish Ministry of Economy, Industry and Competitiveness under the Juan de la Cierva Formacion fellowship No. FJCI-2016-30984. M. Moreto has been supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship No. RYC-2016-21104.

References

[1] M. A. Hamburg and F. S. Collins, "The path to personalized medicine," *New England Journal of Medicine*, vol. 363, no. 4, pp. 301–304, 2010.

[2] 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.

[3] 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.

[4] 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.

[5] S. Deorowicz, A. Debudaj-Grabysz, and S. Grabowski, "Disk-based k-mer counting on a PC," *BMC bioinformatics*, vol. 14, no. 1, p. 160, 2013.

[6] Y. Li and X. Yan, "MSPKmerCounter: a fast and memory efficient approach for k-mer counting," *arXiv preprint arXiv:1505.06550*, 2015.

[7] 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.

[8] 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.

[9] 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.

[10] H. Li, A. Ramachandran, and D. Chen, "GPU acceleration of advanced k-mer counting for computational genomics," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2018, pp. 1–4.

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

[12] R. Wertenbroek and Y. Thoma, "k-mer counting with FPGAs and HMC in-memory operations," in *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, 2018, pp. 233–240.

[13] N. Cadenelli, J. Polo, and D. Carrera, "Accelerating k-mer frequency counting with GPU and non-volatile memory," in *2017 IEEE 19th International Conference on High Performance Computing and Communications*. IEEE, 2017, pp. 434–441.

[14] N. Cadenelli, Z. Jaksić, J. Polo, and D. Carrera, "Considerations in using OpenCL on GPUs and FPGAs for throughput-oriented genomics workloads," *Future Generation Computer Systems*, vol. 94, pp. 148–159, 2019.

[15] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, "CAPI: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7–1, 2015.

[16] V. Moncunill, S. Gonzalez, S. Beà, L. O. Andrieux, I. Salaverria, C. Royo, L. Martinez, M. Puiggròs, M. Segura-Wang, A. M. Stütz *et al.*, "Comprehensive characterization of complex structural variations in cancer by directly comparing genome sequence reads," *Nature biotechnology*, vol. 32, no. 11, p. 1106, 2014.

[17] B. Berger, N. M. Daniels, and Y. W. Yu, "Computational biology in the 21st century: Scaling with compressive algorithms," *Communications of the ACM*, vol. 59, no. 8, pp. 72–80, 2016.

[18] D. Carrera Perez, J. Polo, N. Cadenelli, D. Torrents Arenales, and M. Planas, "A computer-implemented and reference-free method for identifying variants in nucleic acid sequences," Jan. 2020, US Patent App. 16/315,982.

[19] L. Guo, J. Lau, Z. Ruan, P. Wei, and J. Cong, "Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between FPGA and GPU," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 127–135.

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

[21] M. S. Sefat, S. Aslan, J. W. Kellington, and A. Qasem, "Accelerating hotspots in deep neural networks on a CAPI-based FPGA," in *2019 IEEE 21st International Conference on High Performance Computing and Communications*. IEEE, 2019, pp. 248–256.

[22] J. Chen, S. Zhou, and H. Min, "Implementation of parallel medical ultrasound imaging algorithm on CAPI-enabled FPGA," in *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2016, pp. 311–314.

[23] A. Castellane and B. Mesnet, "Enabling fast and highly effective FPGA design process using the CAPI SNAP framework," in *International Conference on High Performance Computing*. Springer, 2019, pp. 317–329.

[24] T. Rosedahl, M. Broyles, C. Lefurgy, B. Christensen, and W. Feng, "Power/performance controlling techniques in OpenPOWER," in *International Conference on High Performance Computing*. Springer, 2017, pp. 275–289.