Future Generation Computer Systems 94 (2019) 148–159

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Considerations in using OpenCL on GPUs and FPGAs for throughput-oriented genomics workloads

Nicola Cadenelli [a,b,*], Zoran Jakšić [a], Jordà Polo [a], David Carrera [a,b]

[a] Barcelona Supercomputing Center (BSC), C. Jordi Girona 1-3, 08034, Barcelona, Spain
[b] Universitat Politècnica de Catalunya (UPC) - BarcelonaTECH, Spain

HIGHLIGHTS

- Refactoring of OpenCL GPU code to efficiently run on multiple FPGAs.
- Multi-kernel FPGA design for k-mer generation that saturates on-board DRAM bandwidth.
- Time, energy, and power evaluation of GPU and FPGAs offloading.
- Analysis of how accelerators parts (i.e., off-chip memory and PCIe) can hinder performance.
- Estimation of how next FPGA boards constitute a real asset for more energy-efficient genomics workloads.

ARTICLE INFO

Article history:
Received 11 June 2018
Received in revised form 4 October 2018
Accepted 17 November 2018
Available online 22 November 2018

Keywords:
FPGAs
GPUs
OpenCL
Genomics
K-mer
Energy-to-solution

ABSTRACT

The recent upsurge in the available amount of health data and the advances in next-generation sequencing are setting the ground for the long-awaited precision medicine. To process this deluge of data, bioinformatics workloads are becoming more complex and more computationally demanding. For this reasons they have been extended to support different computing architectures, such as GPUs and FPGAs, to leverage the form of parallelism typical of each of such architectures.

The paper describes how a genomic workload such as k-mer frequency counting that takes advantage of a GPU can be offloaded to one or even more FPGAs. Moreover, it performs a comprehensive analysis of the FPGA acceleration comparing its performance to a non-accelerated configuration and when using a GPU. Lastly, the paper focuses on how, when using accelerators with a throughput-oriented workload, one should also take into consideration both kernel execution time and how well each accelerator board overlaps kernels and PCIe transferred.

Results show that acceleration with two FPGAs can improve both time- and energy-to-solution for the entire accelerated part by a factor of 1.32x. Per contra, acceleration with one GPU delivers an improvement of 1.77x in time-to-solution but of a lower 1.49x in energy-to-solution due to persistently higher power consumption. The paper also evaluates how future FPGA boards with components (i.e., off-chip memory and PCIe) on par with those of the GPU board could provide an energy-efficient alternative to GPUs.

1. Introduction

The recent upsurge in the available amount of health data and the advances in next-generation sequencing are setting the ground for the long-awaited precision medicine. However, even if publicly accessible genomics and biomedical datasets are becoming more and more popular and sequencing a human genome has become much faster and cheaper than a few years ago, the workloads that process this deluge of data are becoming more and more complex and more computationally demanding. In order to make precision medicine possible at scale within reasonable computer and power envelops, different players from academia, industry, healthcare, and government agencies are working together in the attempt to improve the performance and energy efficiency of such workloads.

Owing to this reason, bioinformatics workloads have been ported to different computing architectures, such as GPUs and FPGAs, to leverage the form of parallelism typical of each of such architectures. GPUs are popular due to their embarrassingly parallel architecture that offers multithreaded SIMD (Single Instruction Multiple Data) with thousands of cores. On their hand, FPGAs are known for their lower performance per watt than GPUs and CPUs. Despite this, they have traditionally used RTL-based (Register-Transfer Level) languages, such as Verilog and VHDL, leading to

* Corresponding author at: Barcelona Supercomputing Center (BSC), C. Jordi Girona 1-3, 08034, Barcelona, Spain.
E-mail addresses: nicola.cadenelli@bsc.es (N. Cadenelli), zoran.jaksic@bsc.es (Z. Jakšić), jorda.polo@bsc.es (J. Polo), david.carrera@bsc.es (D. Carrera).

https://doi.org/10.1016/j.future.2018.11.028

longer development cycles and poor code maintainability. However, FPGAs have recently become more accessible thanks to high-level languages like OpenCL, a portable programming language that allows executing the same code across a variety of platforms. Even though OpenCL offers code portability, performance portability between hardware platforms is not guaranteed and rare to achieve due to the fundamental differences among architectures. Consequently, the porting of an application from one architecture to another requires a consistent refactoring of the offloaded code to adopt device-specific optimizations. Additionally, being a more mature product, discrete GPU boards are usually equipped with high-performance on-board memory (i.e., GDDR5X or HBM2) and a high-speed full-duplex interconnection (i.e., PCIe Gen3 x16 with a dual copy engine). Whereas, discrete FPGA boards are, as of today, a still younger product that is catching up and that usually offers much less performing memory (e.g., DDR4) and a slower half-duplex connection with the host system (e.g., PCIe Gen3 x8 with a single copy engine). As a result, when evaluating the usage of these kinds of boards for throughput-oriented workloads, one cannot look only at the execution time of kernels completely disregarding transfers between the host and the device.

To evaluate the efficiency of GPUs and FPGAs for throughput-oriented genomics workloads, we test them with SMUFIN [1], a state-of-the-art variant calling method that performs a direct comparison of normal and tumor genomic samples from the same patient without the need of a reference genome, leading to more comprehensive results. Software implementations of this method are meant to run at scale to process repositories with thousands of human DNA samples to set the ground for precision medicine. For these reasons, the SMUFIN method is an important real-world use case to analyze. In its latest implementation [2], the initial part of this workload, that consists of a k-mer frequency counting algorithm, was adapted to exploit GPUs. In this paper, we describe how we ported this algorithm to FPGAs, and we compare its performance against the original CPU-only and GPU-accelerated versions.

The contributions in this work can be summarized as follows:

- We describe how the OpenCL GPU algorithm for k-mers generation and shuffling was redesigned from scratch to run in FPGAs using a multi-kernel approach efficiently. Plus, we show how this approach could be extended to multiple FPGAs using a data partitioning mechanism.
- We carry out a scalability analysis of the multi-kernel approach outlining how the solution scales linearly with the number of kernels replicas until a point where performance is limited by off-chip memory and by the high resource utilization – that in turn lowers the clock frequency of the design. Additionally, we show how the random memory accesses of Bloom filters kernels to off-chip DDR4 limit the scalability.
- We evaluate the impact on the entire k-mer frequency counting algorithm. Results show that two FPGAs can outperform a CPU-only execution by a factor of 1.32x in both time- and energy-to-solution. Instead, GPU offloading yields an improvement of 1.77x in time-to-solution but of a lower 1.49x in energy-to-solution.
- We provide samples of the power consumption to show how the GPU noticeably increases the power envelope of the node. On the other hand, the two FPGAs do not increase the power consumption; maintaining it on the same range or even lowering by few Watts. With these power samples, we clarify why the GPU offers a lower improvement in energy-to-solution than it does in time-to-solution (1.49x vs. 1.77x). Whereas, acceleration with two FPGAs exhibits an identical improvement in both metrics.
- We characterize how the inferior on-board memory and PCIe capabilities of current FPGA boards hinder the performance of the entire workload. Besides, we outline how, thanks to their lower power consumption, next FPGA boards constitute a real asset for more energy-efficient genomics workloads.

The remainder of this paper is as follows. Section 2 introduces k-mer frequency counting, an overview of the SMUFIN method and its GPU acceleration. Section 3 surveys related work. Next, Section 4 presents the FPGA acceleration method in detail. Section 5 discusses results of the proposed changes. Finally, Section 6 concludes.

2. Background

2.1. Genomics and k-mer frequency counting

A typical input of a genomics application consists of sequenced DNA samples usually taking hundreds of GB. Such samples are stored as heavily compressed data and include short sequenced strings of DNA nucleobases called *reads*. Each sequenced genome sample typically contains 10^9 to 10^{10} reads; depending on some factors such as depth of coverage, which indicates how many times each DNA position is represented in the sequenced genome. The length of each read is in the order of 10 s to 100 s of bases that are represented by the four character alphabet {A, C, G, T}. In a sequenced DNA sample, along with each base, there's also an associated score that measures its quality; doubling the amount of data. Data that, for a whole human genome, can easily reach around 300 GiB. As a consequence of this enormous amount of data, offloading genomics applications to accelerator like FPGAs and GPUs has become a common trend.

Moreover, many genomics applications require splitting DNA reads into smaller pieces called *k-mers*. k-mers of a nucleic acid read are all the possible sub-sequences within the original read which have a length k. The amount of k-mers in a read of length M is $M - k + 1$. For instance, the number of 8-mers in a sequence of 10 bases is $10 - 8 + 1 = 3$, meaning ACGGCAGCTG has the following 8-mers: ACGGCAGC, CGGCAGCT, and GGCAGCTG. Counting the frequencies of k-mers is widely used for genome assembly and error detection, but it also has other applications such as sequence alignment and variant calling [3,4].

2.2. SMUFIN and its k-mer frequency counting

SMUFIN is a state-of-art method whose peculiarity is the comparison of normal and tumor genomic samples of the same patient without the need for a reference genome [1]. The basic idea behind this method can be summarized in the following steps: (i) input two sets of nucleic acid reads, normal and tumoral; (ii) build frequency counters of normal and tumoral substrings in the input reads; and (iii) compare normal and tumoral counters to find imbalances, which are then extracted as candidate positions for DNA mutations. The method is meant to run at scale to process repositories with thousands of human DNA samples to extract candidate somatic mutations for each patient. This output, together with the clinical records of each patient, will help biologists to identify common groups of mutations among patients with the same disease and vice versa. In other words, this method could allow the discovery of what mutations lead to a particular disease. Software implementations of this method are going to be executed at scale making both time-to-solution and energy-to-solution crucial factors.

The latest software implementation of the SMUFIN method [2] relies upon an initial k-mer frequency counting algorithm - Fig. 1.

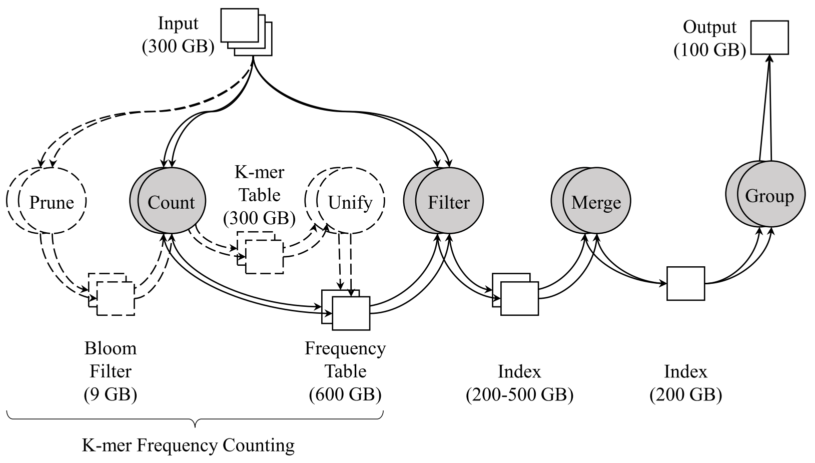


Fig. 1. SMUFIN's variant calling architecture: overview of units and its data flow. In this example, computation is distributed among 2 different nodes. Each node reads the entire input and uses a data partitioning scheme to discard k-mers not belonging to their partition. Prune and Unify units are optional and used to run in configurations with less than 2 TB of DRAM.

Due to the peculiarity of the method, that takes as input normal and tumor genomic samples of the same patient, the algorithm counts k-mers from both kind of input sample building a frequency table of around 600 GB per each patient and it is used in the later units of the software as input. This implementation of the k-mer counting algorithm is designed to handle whole genome k-mers for values of k in the range of $24 < k < 32$. Within this range k-mers are unique enough to provide useful results. For values outside this range the results would become either too general (for $k < 24$) producing meaningless results or too selective (for $k > 32$) risking to miss potential mutations. This algorithm consist in the following three units:

- *Prune*: Produces a Bloom filter that tells whether each k-mer is observed more than once, or not, in both normal and tumoral input genomes. This filter is built using a chain of two simple Bloom filters where only those k-mers already in the first filter (i.e., seen before), plus eventual false positives, are propagated to the second one. Once processed both genomes entirely, the second and smaller Bloom filter of the chain constitutes a structure that tells whether a k-mer is unique or not in the two genomes. This filter is used in the following Count unit to discard unique k-mers that are not relevant for the algorithm, reducing the memory requirement. In fact, albeit the size of the second Bloom filter can be as high as 9 GiB, it allows sparing hundreds of GB required to store unique k-mers in a simple list or directly in the frequency table.
- *Count*: Builds a frequency table of normal and tumoral k-mers using the Bloom filter generated in the previous unit to discard unique k-mers. Intermediate tables are eventually swapped to storage if the system does not have enough DRAM available.
- *Unify*: Combines frequency tables swapped in the previous unit. This unit also changes the memory layout of the tables to its expected form required by the following units. Memory layout that can change according to the kind of DNA analysis being executed in the next unit of the application. Lastly, this unit also removes false positives given by the Bloom filter (i.e., unique k-mers). An operation that, at this point, is very simple because it only requires to check if the sum of the normal and tumoral counters in the frequency is equal to one. This unit is not accelerated so it's not discussed further in this paper and it's not taken into account in the result section.

This latest implementation of the method, makes use of a data partitioning scheme to spread the k-mers, thus the workload, among different CPU threads but also among multiple nodes if needed. This partitioning scheme is based on a simple base-match criterion defined a priori that uses statistical data on k-mer distribution to spread the amount of k-mers equally into partitions. The work we present here focuses only on single node configurations; thus the partitioning scheme is used to spread the work among multiple CPU threads; usually with as many partitions as the number of CPU *consumer* threads. The main benefit of this partitioning of the k-mers is that it enables CPU consumer threads to work on dedicated data structures – chain of two Bloom filters or frequency table depending on which unit is being executed – removing the need to synchronize accesses to the data structure. Note that, the aggregation of all frequency tables or bloom filters can be seen as a big unique data structure covering the entire k-mers domain.

Additionally, the latest implementation enables the offloading of some computation intensive operations of the Prune and Count units to a GPU. Offloading, takes place by splitting the input genomes into many chunks – hereinafter referred to as input chunks – and adopting a 5-step double buffering pipeline to overlap CPU/GPU computation and PCIe transfers. At each cycle of the pipeline each step works on a different chunk of data. In more detail, the five steps comprise: (i) CPU loader threads read and decompress input files from storage to an input chunk in DRAM; (ii) PCIe transfers of an input chunk containing DNA sequences and quality markers from host DRAM to GPU on-board memory; (iii) GPU kernels execution to generate k-mers from input DNA sequences; (iv) PCIe transfers of k-mers from GPU to host DRAM; and (v) CPU consumer threads that insert k-mers into the chain of Bloom filters in the Prune unit or into the frequency tables in the Count unit. The GPU algorithm counts of five different kinds of kernels executed one after the other per each input chuck. The first four kernels are executed only once per each input chuck. Their aim is to generate all k-mers from the input DNA reads and to shuffle them according to partition – thus, CPU consumer thread – they belong to. Differently, the last kind of kernel is executed, per each input chunk, as many time as many partitions. Its purpose is to unburden CPU consumer threads of some or all Bloom filters lookups taking advantage of the high bandwidth memory typical of GPUs. The kernels can be summarized as follows:

- *Zero-out kernel*: Resets all device-side data shared among different kernels from the previous cycle of the pipeline.
- *Encode kernel*: Generates all k-mers from the DNA reads and builds a global histogram to count how many k-mers belong to each partition in the entire input chunk. Then, OpenCL

work-items of each work-group collaborate to store all k-mers of each group already shuffled at a group level. Finally, it also stores information like the global histogram and the offset of each work-group so to be used as input in the next kernels.
- *Prefix-sum kernel*: Performs an exclusive prefix-sum on the global histogram.
- *Shuffle kernel*: Reads and rewrite all k-mers generated in the Encode kernels to shuffle them by the partition they belong using the exclusive prefix-sum values as offsets.
- *Bloom Filter kernel*: Tests whether each k-mer is the second-level Bloom filters or not. This is the only kernel that behaves differently depending on the unit being executed. In fact, in the Prune unit, this kernel collaborates with the CPU consumer threads in checking if each k-mer is already in the second-level. If that is the case, there is no need to add that same k-mer again to the chain of Bloom filters. When this happens, the kernel marks the k-mer as invalid so that the CPU threads will know to ignore it. In other words, the GPU helps the CPU preventing useless lookups and adds to the chain of Bloom filters. In the Count unit instead, this kernel unburdens the CPU of all the lookups to the Bloom Filters and marks as invalid those k-mers not in the filter – k-mers seen only once in the whole input.

As Algorithm 1 shows, the code resembles a parallel count sort that uses SMUFIN data partitioning scheme to shuffle k-mers by the partition they belong to. With input chunks of hundreds of MB and containing millions of DNA reads, the code takes advantage of the high level of parallelism offered by GPUs and showed a reduction of the time-to-solution for the entire k-mer frequency counting algorithm [2]. In the following sections we describe how this GPU code was redesigned to work on FPGAs.

3. Related work

Thanks to high-level languages such as OpenCL, designing FPGAs became more accessible and we expect that always more code from all possible domains will be ported to FPGAs. However, to achieve good performance an extra effort to port and optimize the code for FPGAs must be made. In this direction, IntelFPGA Design Examples offers basic OpenCL example for a range of different optimizations. Among these, [5] shows how channels and multiple kernels can be used to spread the work to multiple pipeline (kernels). Similarly, [6] studies how the multi-kernel design can be used for relational databases. However, very little was found about comparing OpenCL performance portability between GPUs and FPGAs on real-world fully fledged applications [7–9]. In fact, most of the literature focuses either on synthetic benchmarks like in [10,11], or only on the kernel time completely disregarding PCIe transfers between host and device, and without showing the overall impact on the application. In particular, in [7] the authors showed how Cherenkov angle reconstruction algorithm, an algorithm used in high energy physics, is 3.6x slower on an FPGA than on two GPUs but that, when PCIe transfer times are accounted for, the FPGA is only 1.4x slower. Proving that does not matter how fast a kernel is, if the transfer component is much larger, the overall speedup will be significantly reduced. Furthermore, because of the low power profile offered by the FPGA, the single FPGA implementation is 3.4x more energy-efficient than the dual-GPU implementation.

With regards of genomics, different efforts has been done to exploit FPGAs using OpenCL for Smith–Waterman algorithm [12–16] and for DNA Assembly with De Bruijn Graphs [17]. Others instead [9], explored the acceleration of the PairHMM algorithm for the GATK (Genome Analysis Tool Kit) mapping the algorithm into a 2D systolic array. Results shown an improvement of peak performance of 3.4x over the performance obtained with a GPU NVIDIA Tesla K40 and of a 2x over the best-practice with Intel AVX technology using 44 cores. However, the overall speedup of the entire application over the best Intel AVX implementation was of a lower 1.2x; confirming how important it's to consider the overall impact on the application.

The work presented in [18] studies how k-mer counting using Bloom filters on FPGA can get one order of magnitude faster when using newer memory technologies such as HMC (Hybrid Memory Cube) instead of DDR memory, proving how FPGAs have the potential to become more competitive with the adoption of more recent memory technology. Genomics workloads and pipelines are, in general, a good fit for resource disaggregation but their large-scale exploitation hasn't been explored much and usually focuses on adapting the existing algorithm. With SMUFIN instead, the work presented in [19] shown that NVMe over Fabrics storage can be shared across multiple instances running on different nodes with a minimal penalty in performance.

4. Acceleration method

The characterization of the application presented in [2] showed that the main bottleneck for the Prune and Count units is the randomness of accesses to host DRAM given by lookups and insertions in Bloom Filters and hash tables. In this paper, we do not discuss a method to improve the data locality, nor we intend to. We focus instead, on how we redesigned the accelerated GPU code for FPGAs and on FPGA-specific techniques required to achieve good performance on FPGAs.

First, we present a method to reduce global memory access that could also be used with GPUs. Next, we describe in detail how the OpenCL kernels were redesigned to run on FPGAs. Finally, we demonstrate how the redesigned code can take advantage of multiple FPGAs at the same time.

4.1. Reducing global memory usage and accesses

As described in the previous section, the GPU algorithm writes to memory all the k-mers two times. At first, in the encode kernel, k-mers are written to an extra buffer for staging. Then, in the shuffle kernel, k-mers are read back to the device, shuffled, and written to the final output buffer. This additional memory trip and the extra buffer for staging k-mers are required by the shuffling algorithm. Algorithm that shuffles in an out-of-place manner because it's impossible to know a priori the offsets to use when writing the k-mers to the output buffer. This because, these offsets are the result of the exclusive sum-prefix on the global histogram which changes from one input chunk to another. This method works well on GPUs where the memory bandwidth offered by GDDR5X or HBM2 is in the order of hundreds of GB/s. However, on today's FPGA boards, which use DDR4, memory bandwidth is one order of magnitude lower and global memory accesses should be minimized.

To prevent this second memory trip on FPGAs, we developed a different strategy that relies on the partitioning scheme adopted by SMUFIN to evenly distribute the number of k-mers between the different partitions (i.e., CPU consumer threads). Knowing that the distribution is balanced and ensuring that each input chunk is large enough to have tens of millions of k-mers, one can assume that given any input chunk the number of k-mers generated is evenly distributed among the partitions. In such a way, predefined fixed offsets can be used to write directly to the output buffer without the need to stage to a third buffer. Still, synchronization among all work-items is required to prevent threads from overwriting each others results. Also, to make this solution compatible with the host code, the exclusive sum-prefix is still needed by the CPU consumer threads to know how many k-mers are in each partition. Obviously, there is always the chance that in a particular input chunk, one partition gets more k-mers than expected; overwriting the k-mers of the next partition and leading to wrong results. To make this unlikely to occur, this solution was complemented by allocating output buffers slightly bigger than required; thus, wasting a bit of memory and adding more PCIe traffic. But, it's a simple and effective solution that proved to work already with a over-provisioning ratio of 1.1. However, this does not completely prevent overwrites from happening and it's essential that the application is able to detect these kind of episodes. For this, the application ensures that each partitions has no more k-mers than allowed in each output chunk. That is, for partition *i* if *histogram[i]* > *offset[i+1] - offset[i]* then, an unbalance occurred and some k-mers of partition *i* were written in the region of memory dedicated to partition $i+1$. When this occurs, the host program detects it, and resolves, rescheduling the kernels on only half of the input chunk and adding a *bubble* in the first two stages of the pipeline. In the next cycle, the second half of the chunk is processed, and the first two stages will stall due to the bubble. Clearly, this is not an optimal solution as it slightly increases the execution time. But, it is good enough because, with the over-provisioning of the output buffer, overwrites are very unlikely to happen.

4.2. FPGA-specific optimizations

OpenCL programming language is designed to be used on different kind of hardware platforms (e.g., CPUs, GPUs, DSPs, FPGAs). However, its performance varies from architecture to architecture. Moreover, optimizations typical of GPU programming can lead to poor performance on FPGAs and the other way around. This fact is mostly due to the different parallelism offered by GPUs and FPGAs (i.e., multithreaded SIMD vs. Pipeline) and it is a key concept when porting code from one architecture to another. For this reasons, many GPU algorithms require the refactoring of the whole accelerated code to make it best suit FPGAs.

In our case, the GPU algorithm relies on a parallel histogram and shuffle algorithm that uses millions of threads to process the just as many DNA reads in each input chunk like outlined in Section 2.2. In detail, the code in Algorithm 1, makes use of 64-bit atomic operations (line 25) and also of local and global barrier functions (lines 4, 19, and 37); making it a bad fit for FPGAs or even unfeasible for those devices that lack some of these functionalities (e.g., 64-bit atomics). Motivated by these factors, we redesigned the algorithm from the ground up to take advantage of FPGA-specific features and optimizations.

4.2.1. Parallel paradigms

To better adapt the algorithm to FPGAs, the OpenCL NDRange kernels used for the GPU were replaced with OpenCL tasks — single work-item kernels using only one thread. With only one thread, the global histogram can be built directly without the need of atomic operations nor memory barriers. In such a way, the code gets extremely simplified; making it a simple build for the FPGA compiler and reducing the number of resources needed. Moreover, when using tasks, also the Zero-out and Prefix-sum kernels can be removed. In fact, the only reason why these two kernels are separated is that there is no way to create a global memory barrier that synchronizes all the work-items of an NDRange if not to conclude the kernel. Similarly, also the Bloom Filter kernel was changed to a task by merely adding a loop to process all k-mers.

Even if we changed the parallelism model to a pipeline, loop unrolling could always be exploited to process multiple DNA reads in parallel like outlined in the listing 3. In fact, using loop unrolling (line 8 and 17), the encode kernel can process many DNA reads in parallel and generate one k-mer from each read at every cycle.

Table 1
Hardware specification.

	Host CPU (2x Xeon E5-2680v3)	GPU (GeForce GTX 1080 Ti)	FPGAs (Arria 10 1150 GX - Nallatech 510T)
Compute units	24 Cores	3584 Cores	427 K ALMs
Maximum frequency	3200 MHz	1582 MHz	450 MHz
Memory size	512 GB	11 GB	8 GB
Memory technology	LR-DDR4	GDDR5X	DDR4
Memory bandwidth	72.5 GB/s	484 GB/s	37.5 GB/s
Maximum TDP	2×120 W	250 W	112.5 W
PCIe interface	–	Gen3 x16	Gen3 x8
PCIe dual copy engine	–	Yes	No

4.2.2. Kernels replication and channels

To prevent the need of synchronization when writing k-mers to global memory, we adopted a single-producer/multiple-consumers solution that takes advantage of Intel's OpenCL channels to directly stream the k-mers from the producer to the consumer kernels like Fig. 2 depicts. In this solution, the producer (encode) kernel processes N DNA reads in parallel using loop unrolling and delivers, at each FPGA cycle, one k-mer to each of the N consumer (shuffle) kernels. Consumer kernels that, in turn: read the k-mers from the dedicated channel; and store, shuffling, them to a dedicated global memory buffer and builds a private histogram. In this way, k-mers are equally spread among the different consumer kernels, each of them receiving the same amount of k-mers set by the host program.

While kernel replication removes the need of synchronization, the usage of channels allows to directly stream data from the producer to the consumer kernels; eliminating the need to write all k-mers to global memory in the producer kernel and to read them back in the consumer kernels.

In general, this approach provides a way to equilibrate possible throughout unbalances between the producer and consumer code. In our case, the number of DNA reads processed in parallel in the producer must match the number of consumer kernels and should be chosen accurately. Ideally, the more the consumer kernel is replicated, the more FPGA resources are used, and the higher should be the throughput. However, performance does not scale linearly because at some point the maximum memory bandwidth is reached; thus, the replication factor depends not only on the FPGAs but also on the board used and its memory. We describe this in more detail in the result section. Moreover, now that the solution has multiple different output buffers, one can also decide to replicate the Bloom Filter kernels to be able to filter the k-mers of one partition from all, or some, output buffers in parallel.

4.2.3. On-chip and global memory

The only downside of SMUFIN partitioning scheme is that given a DNA read; there is no relation between consecutive k-mers and the partition to which they belong. This is because the partitioning criterion only takes into account the k-mer itself. As a consequence, when shuffling, the partitions of two consecutive k-mers and so the offsets in the output buffer are completely random; resulting in random memory accesses. To mitigate this effect and perform memory writes in bulk, the consumer kernels are designed to create a small cache in the on-chip memory of the FPGA (*kmers_cache* in Algorithm 4). This cache allows to store up to eight k-mers per each partition and to flush to global memory all eight k-mers of one partition using one unique 512-bit wide request to use the memory bandwidth at best.

4.3. Multi-FPGA support

As described in [2], when there is no space to store the second level Bloom filters in the accelerator memory, the application

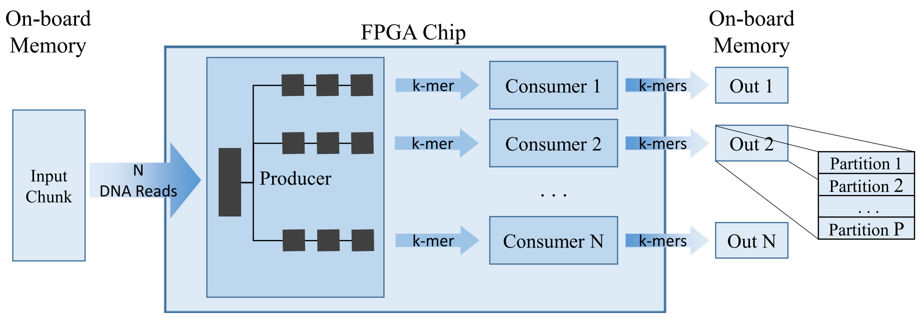


Fig. 2. Overview of the FPGA Producer–Consumers kernels and data flow. The figure depicts an FPGA design with one producer and N consumers kernels. The Producer kernel reads and processes N DNA reads in parallel using loop unrolling (i.e., #pragma unroll N) and sends one k-mer to each Consumer kernel at each cycle. K-mers are streamed from the Producer kernels directly into the Consumer kernels via Intel OpenCL Channels preventing accesses to the FPGA main memory. The Consumer kernels shuffle the incoming k-mers and write them to the main memory in bulks using 512-bit wide memory writes.

virtualizes the device memory in host DRAM. In such way, the host program itself takes care of copying each Bloom filter in the device memory before being used at each cycle. Meaning that, at each cycle, the entire 9 GB of the filters are transferred from the host to the device; putting pressure on the PCIe bus. Despite this, performance and system analysis carried out by the authors in [2], showed that neither the GPU nor the PCIe Gen3 x16 bus were ever the bottleneck of the application. Instead, it was the CPU consumer threads. However, when using devices with a narrower, thus slower, PCIe connection and slower on-board memory technology, transfers between the host and the accelerator take more time and might become the bottleneck of the software pipeline. To overcome this, one possible path is to use multiple accelerators at the same time.

We implemented the support to multiple FPGAs assigning a subset of partitions to each FPGAs and copying all input chunks to all accelerators. The changes in the OpenCL kernel code were minimal and regarded only the FPGA shuffle kernel that was instructed (line 14 of Algorithm 4) to output only those k-mers belonging to partitions assigned to the FPGA where the kernel is running (parameter *pid_in* and *pid_in* of Algorithm 4). In this manner, each FPGAs only need to process, and store, the Bloom filters of the partitions assigned to it. The advantages of this solution are multiples. Firstly, the Bloom Filter kernels are distributed and executed in parallel on multiple FPGAs decreasing the kernel time. Secondly, spreading the Bloom Filters to the different accelerators reduces the memory requirement on each accelerator. And, when the overall memory of all FPGAs is enough to fit all the buffers, there is no need to virtualize the accelerator memory; reducing the PCIe transfers. In fact, when this is the case, in the Prune unit offloaded Bloom Filters can be updated every few cycles of the software pipeline. In the Count unit instead, where the Bloom filters are only used for lookups, the filters are copied to the FPGAs memory only at the beginning of the execution; significantly reducing the PCIe transfers. The last advantage is that, because kernel parameters are used to tell each FPGA its subset of partitions, this implementation allows to reuse the same FPGA design with an arbitrary number of FPGAs without requiring to recompile.

5. Results

In order to evaluate the impact of the presented work, we compare execution time, energy and power consumption of the latest version of SMUFIN running with and without accelerators. For simplicity's sake, and because the execution time of non-accelerated units is constant, we report only the results of accelerated units relative to k-mers frequency counting — Prune and Count. The evaluation also includes an analysis of the scalability of the single-producer/multiple-consumers design and a discussion on downsides and benefits of the accelerator boards used (see Table 1).

Table 2
FPGA Resources used, Clock Frequency, and Compilation Time.

# Kernels replicas	Logic	I/O pins	DSP blocks	Memory bits	RAM blocks	FPGA frequency	Compilation time
1	14%	35%	3%	11%	21%	223.9 MHz	101 min
2	19%	35%	5%	15%	27%	234.3 MHz	142 min
4	27%	35%	11%	22%	42%	227.5 MHz	272 min
8	44%	35%	21%	39%	72%	211.9 MHz	351 min

5.1. Experimental setup

We conducted our experiments on a machine with two Intel Xeon CPU E5-2680v3 with 48 CPU threads in total, sixteen 32-GB DDR4 DIMMs running at 2133 MHz for a total of 512 GB of DRAM. As storage, we used one 1600 GB HGST Ultrastar SN100 Series NVMe SSD used as normal storage and one 375 GB Intel Optane SSD DC P4800X used as memory extension. Both NVMe were formatted using ext4. The machine ran CentOS 7.4 with a 3.10.0–693 kernel with OS swapping disabled. Source codes were compiled with g++ version 4.8.5 and the -03 flag set.

For the GPU, we used one GeForce GTX 1080 Ti with 11 GB of GDDR5X, Nvidia Driver version 390.30 offering OpenCL version 1.2. The GPU was used with with ECC enabled and GPUBoost disabled.

On the FPGAs side, we used a Nallatech 510T board equipped with two independent Arria 10 1150 GX FPGAs for a denser compute power. Each of the two FPGA has 427 K Adaptive logic modules (ALMs), 1.7 M registers, 2.7 K M20 K memory blocks (53 Mb), 12.7 Mb MLAB memory blocks, and 1,518 variable-precision DSP blocks; and sports a maximum frequency of 450 MHz. with a theoretical peak performance of 1.366 TFLOPS. Both FPGAs come with 16 GB of 2133 MHz DDR4 memory in a 4-bank configuration but, at the time of writing, only 2 banks per FPGA – 8 GB – could be used with OpenCL for a nominal DDR4 memory bandwidth of 37.5 GB/s. Even if hosted in the same board, the two FPGAs are independent and, as shown in Fig. 3, they only share the PCIe Gen3 x16 bus and an on-board dedicated interface to talk to each other. FPGA code was compiled and executed using IntelFPGA SDK for OpenCL version 17.1 Build 270 and Nallatech board support package version R001.005.004 for HPC offering OpenCL version 1.0 with an embedded profile.

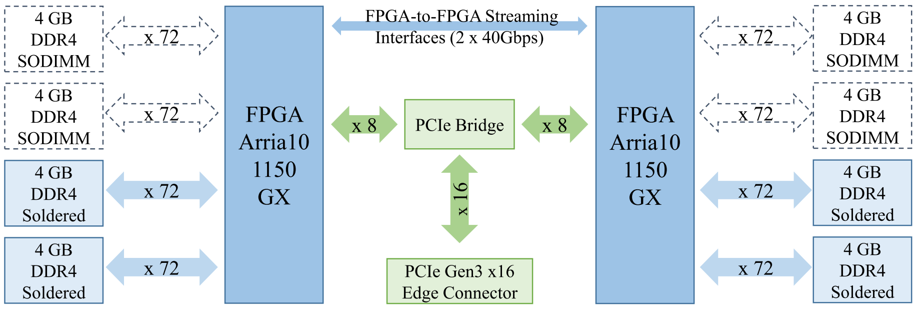
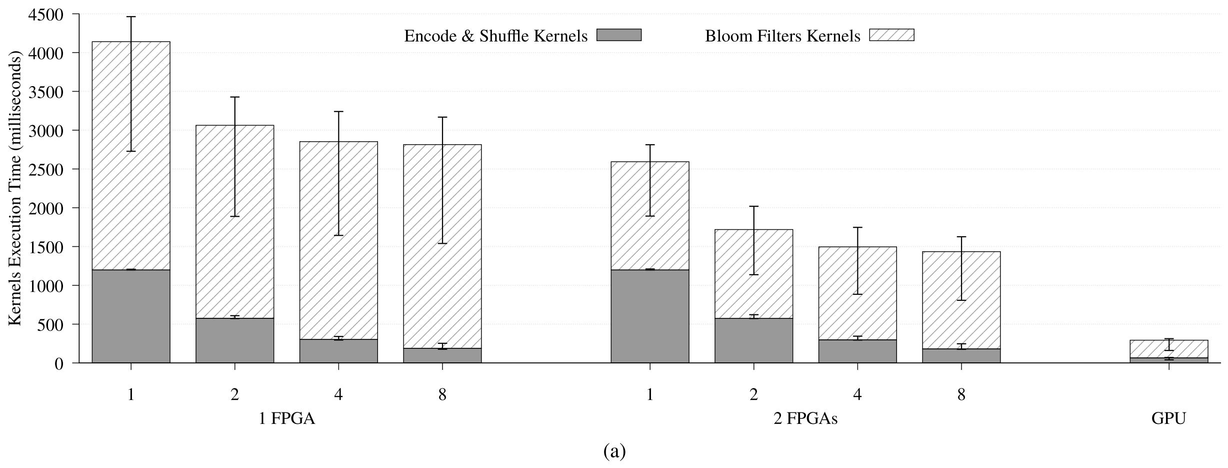


Fig. 3. Diagram of the Nallatech 510T dual-FPGA board.



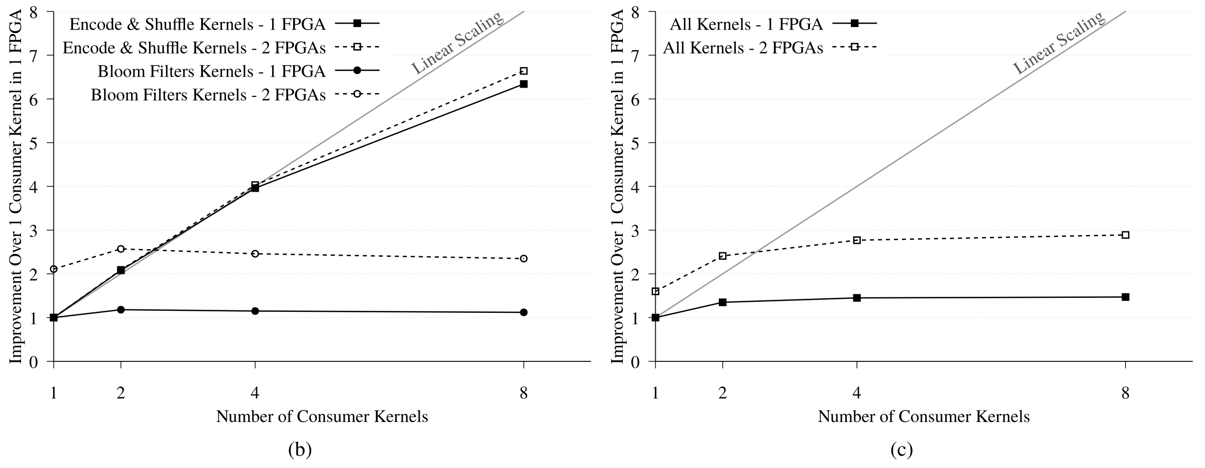


Fig. 4. Performance comparison of OpenCL kernels running on FPGAs (with different kernel replications) and on the GPU (a) and scalability of the FPGA kernels (b) and (c). Data is relative to the 1630 circa cycles of the software pipeline required to process the whole input.

5.2. Evaluation methodology

We executed using four different hardware configurations: (i) CPU only, (ii) CPU plus one FPGA, (iii) CPU plus two FPGAs, and (iv) CPU plus one GPU. In each configuration, we always used 8 CPU loader threads and 48 CPU consumer threads; thus, 48 partitions and as many Bloom Filters in the accelerators. Accelerators unburdened both kinds of CPU threads and removeed communication from loader to consumer threads. Communication that in the CPU only configuration takes place via 8 $\times$ 48 dedicated single-producer single-consumer queues. Note that, in configurations (ii) and (iv) that use only one accelerator, the device memory was not enough to fit all Bloom filters; thus, the application virtualized the device memory in host DRAM as outline in [2]. Meanwhile, the overall memory of the two FPGAs of configuration (iii) was enough not to require the virtualization; thus, reducing the PCIe traffic in both Prune and Count unit. In fact, as discussed in 4.3, only for configuration (iii), in the Prune unit Bloom filters are updated once every eight cycles of the software pipeline instead than at

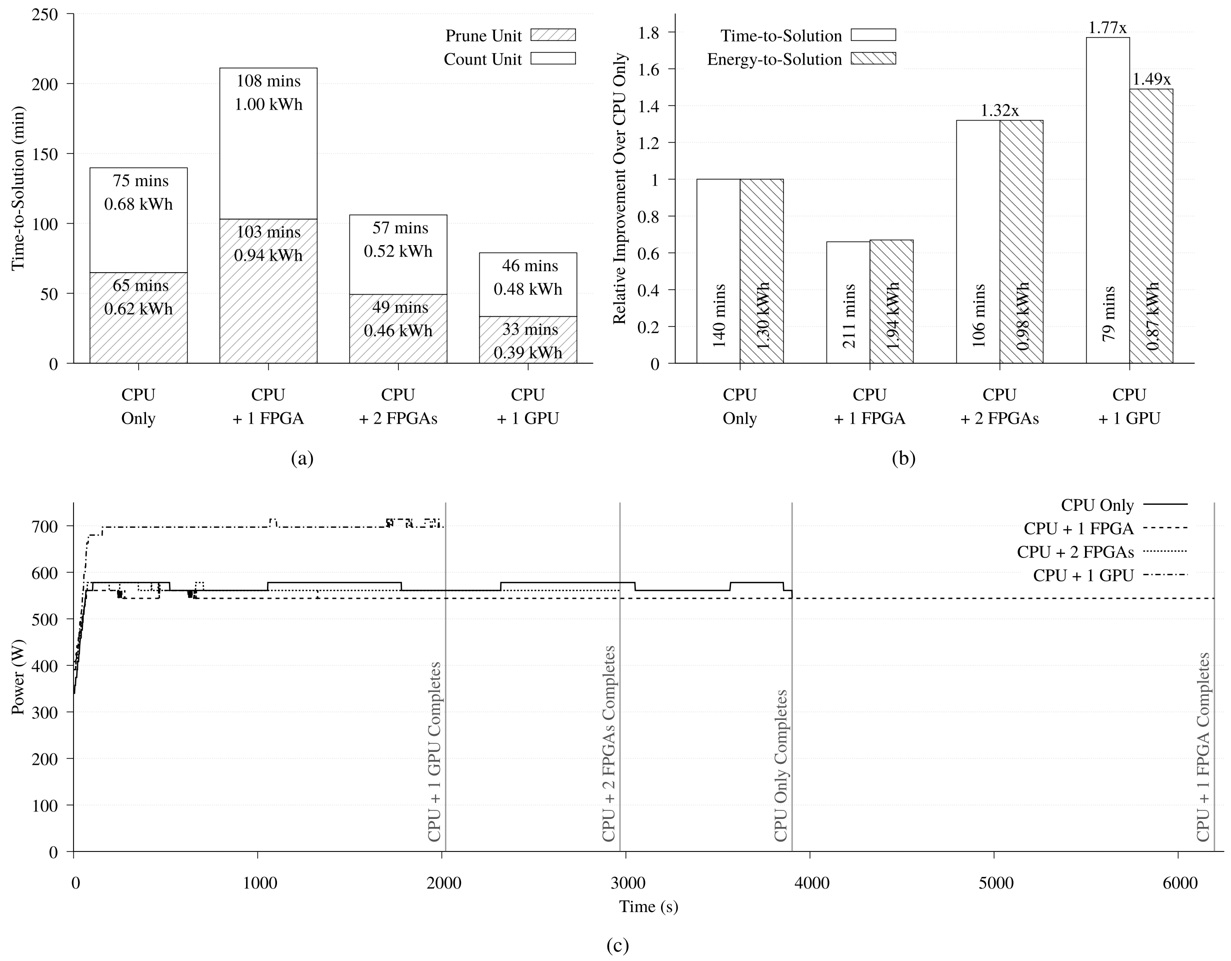


Fig. 5. Time-to-solution and energy-to-solution (a), relative improvements (b), and power samples (c) of SMUFIN's k-mer frequency counting without acceleration, with one FPGA, with two FPGAs, and with one GPU. Power samples relative to the Count unit are omitted for similarity and space.

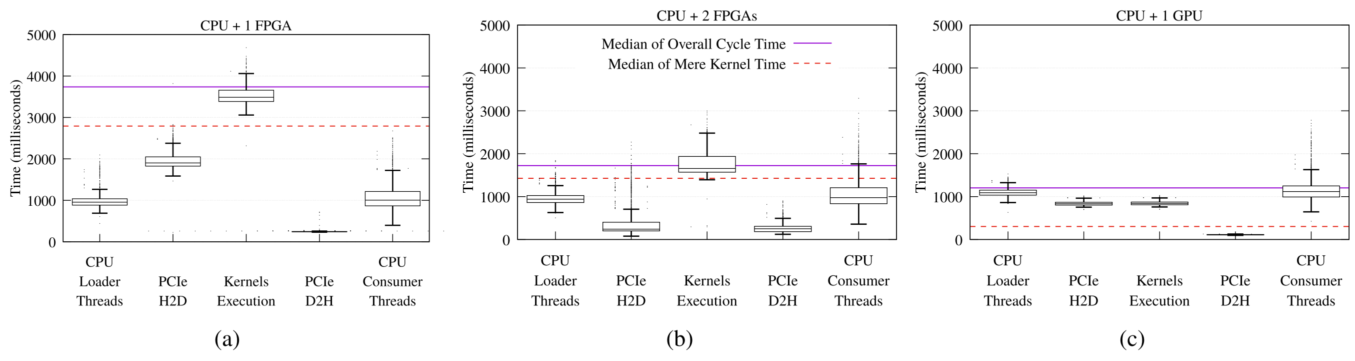


Fig. 6. Distribution of execution time of each individual step of the software pipeline using one FPGA (a), two FPGAs (b), and one GPU (c) to execute the Prune unit for the whole input.

each cycle. In Count unit instead, this configuration allowed to load the Bloom filters to the two FPGAs at the very beginning of the execution and, because during this unit the filters are not altered by the CPU consumer threads, there is no need to update them constantly.

For all the hardware configurations we evaluated the time-to-solution and energy efficiency to count k-mer frequency of a personalized genome based on the Hg19 reference. The input genome was characterized by randomly chosen germline and somatic variants as described in [1]; including SNPs, SNVs (more than 100 bp apart), translocations, and with random insertions, deletions and inversions, all ranging from 1 to 100 Mbp. In silico sequencing was simulated using ART Illumina21. The total size of the final normal and tumoral samples is of 312 GB of gzip compressed FASTQ files — around 638 GB once uncompressed. With accelerators, the software double buffering pipeline was set

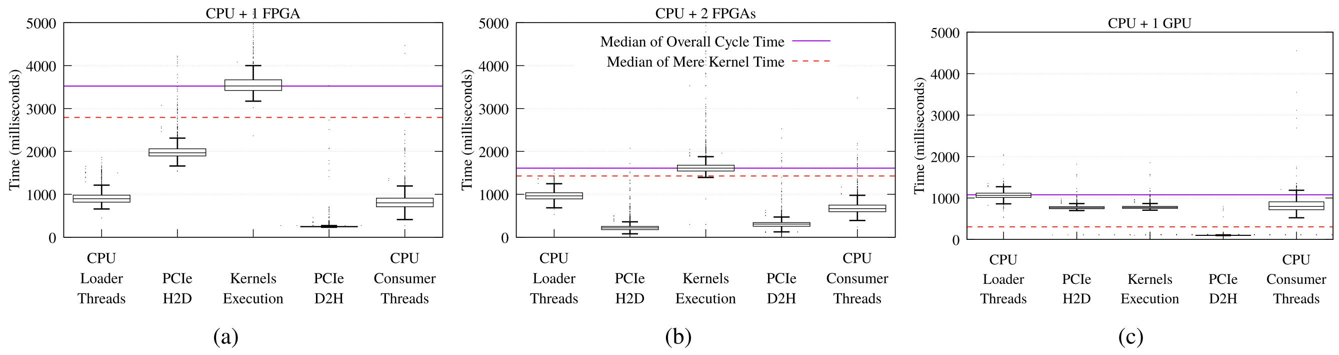


Fig. 7. Distribution of execution time of each individual step of the software pipeline using one FPGA (a), two FPGAs (b), and one GPU (c) to execute the Count unit for the whole input.

to work with input chunks of 400 MB; requiring around 1630 cycles to process the whole input.

Energy measurements and power samples were collected using a metered-by-outlet PDU (Power Distribution Unit) and also IPMI (Intelligent Platform Management Interface) to validate the results. Lastly, to prevent accounting the power consumption of idle accelerators, the PCIe slot of unused device was disabled from the BIOS.

5.3. *Performance and scalability of OpenCL kernels*

As we outlined in Section 4.2.2, one possible strategy to increase the performance of FPGAs is to increase the parallelism by replicating kernels. We achieved this using a single-producer/multiple-consumers design were the internal parallelism of the produced kernel, obtained with loop unrolling, equals the number of the consumer kernels.

Fig. 4a shows the execution time of the kernels with a breakdown for the Encode plus Shuffle kernels and the Bloom filter kernels in one and two FPGAs. For comparison, execution time of equivalent kernels on the GPU are also reported. With multiple devices and kernels, that are executed concurrently, the plot reports the execution time of the slower kernel among all devices. Also, the time related to the Bloom filters kernels is the aggregated time for all partitions, which are executed one after the other. That is, 48 with one FPGA and 24 with two FPGAs. The plot clearly shows how the execution time of the Encode and Shuffle kernels scales with the number of Consumers kernels but it does not with the number of FPGAs. Intuitively, this is because when scaling to multiple devices, the Encode and Shuffle kernels of each device still process the same data, like if only one device was to be used. With the only difference being in the shuffle kernels that discard k-mers belonging to partitions assigned to another device. As a result, when increasing the number of devices, the only difference is in the amount of data written by the Shuffle kernels in each device. For instance, if with one device the output is of 1 GB; with two devices instead, the amount of output is spread evenly for a 512 MB on each FPGA. More in detail, Fig. 4b illustrates that the kernel execution time of only the Encode and Shuffle kernels scales linearly to up to four replicas; but, with eight replicas the improvement starts to be sub-linear. With the rationale being the lower frequency achieved by the FPGA design, noted in Table 2, and the pressure on DRAM subsystem. The former can be either a direct consequence of higher FPGA resourced used by the design, a different seed used in the place-en-route algorithm during compilation, or a combination of the two. Regarding the pressure on DRAM instead, one can deduct this by looking at how, with eight replicas, the execution time with two FPGAs is slightly better than with one.

On the Bloom filter kernels side instead, Figs. 4b and 4c illustrate how these kernels do not scale with the number of parallel kernels. Intuitively, this is due to the random memory accesses of the Bloom Filter due to the many random requests. On the other hand, as we split the Bloom filter kernels between devices, they do scale linearly with the number of FPGAs. Besides, Figs. 4c details how the execution of the Bloom filters dominates the execution time and how this hinders performance scalability. However, because this is due to the low bandwidth offered by the DDR4 DIMMs, next-generation FPGA board equipped with faster memory technology are expected to be competitive like studied in [18].

Finally, even if the best FPGA configuration – eight Shuffle kernels and two FPGAs – is approximately 4.9 times slower than the GPU, one should not look only at the execution time of kernels but also at the total execution time of the entire application. In fact, to evaluate one board and one acceleration technique, one should always keep into account other factors like: host-device synchronization, PCIe performance, how well PCIe transfers and kernels execution are overlapped on each board, and how is the overall software impacted.

5.4. *Considerations on time-, energy-to-solution, and power consumption*

Figs. 5a and 5b display the time- and energy-to-solution of all four hardware configurations. Results show how only one FPGA is not sufficient and becomes the bottlenecks of the application considerably increasing the execution time. Both configurations with two FPGAs and one GPU outperform the non-accelerated configuration in terms of time and energy. Moreover, as depicted in Figs. 5b, while with two FPGAs there is an improvement in time- and energy-to-solution of 1.32x for both measurements, with one GPU instead, the improvement is of 1.77x in time-to-solution but of a lower 1.49x in energy-to-solution. This difference between the improvement in time and energy of the configuration with one GPU is to attribute to the higher power consumption of GPU itself. In facts, like Fig. 5c illustrates, with one GPU there is steady higher power consumption than with any other configuration. This figure shows how, not only is the power consumption with FPGAs similar to the CPU Only configuration, but, in some cases, it's even lower; demonstrating how the FPGAs are more power-efficient than the CPUs.

Lastly, the difference in the power consumption between two FPGAs and one GPU suggests that: if it were possible to shorten the execution time of the configuration with two FPGAs of some minutes, this configuration could become the most energy-efficient even if still slower than the configuration using one GPU.

5.5. Final considerations on GPU and FPGAs performance

Considering the different FPGAs and GPU components, it is not easy to define the real reason of why the setup with two FPGAs falls behind the one with one GPU. Fig. 4 suggests that the FPGA kernels as main bottleneck. However, when it comes to evaluate one board for an application, where PCIe transfers and kernels execution are overlapped, one must look beyond the mere kernel execution time taking into account the cycle time of each component of the software pipeline. A consideration that is even more important when there are dependencies between different steps. For instance, a kernel execution that, before to start, must wait for PCIe transfer to conclude. A visual aid that helps in this direction comes from the plots in Figs. 6 and 7. These plots show the distribution of the execution time of all five steps of the software pipeline used to overlap CPU threads, PCIe transfers, and kernel execution on the accelerators. Intuitively, the longest step consist in the main bottleneck of the software pipeline. Plots 6a, 6b, 7a, and 7b confirms that the kernels execution is the bottleneck for the configurations using FPGAs. However, as discussed in 5.3, the random memory accesses of the Bloom filter kernels together with the low bandwidth offered by the on-board DDR4 are to be blamed. Moreover, these plots show the effect of Bloom filter kernels waiting for the PCIe transfer of the relative Bloom filter to complete before to start; thus, increasing the cycle time for kernel execution. One can observe the effect of these dependencies in all plots in Figs. 6 and 7, where the median of the box plots relative to the kernel execution is significantly higher than the median of the mere kernel execution time. The figures also show how, for configurations with one device, the difference between the two medians is even more significant. This because, in these configurations, the application virtualizes devices memory in the host DRAM, copying the Bloom filters buffer to the device at every cycle of the software pipeline. In the configuration with two FPGAs instead, where the overall memory is enough to store the second level Bloom filters, the Bloom filters in the accelerators are updated only once every eight cycle in the Prune unit and only at the beginning of the execution in the Count. Which results in a much lower PCIe host to device time and much closer, yet distinct, medians of mere kernel time and kernel execution step of the pipeline. Lastly, and to be completely fair, these plots also prove that in the configuration with one GPU, the GPU itself is not always busy, and how the CPU threads become the bottleneck. However, because SMUFIN makes use of the whole 11 GB of memory that the GeForce GTX 1080 Ti offers, when deploying it at scale it's not possible to share the GPU with other instances of SMUFIN or other workloads.

Another aspect that must be taken into consideration is the inferior PCIe capabilities of the FPGAs when compared to the GPU used. Not only have the FPGAs a narrower and slower connection, but also they lack of a dual copy engine. Meaning that, whereas the GPU is capable of overlapping transfers in both directions, transfers to and from the FPGAs are serialized. As a result, with FPGAs the PCIe time increases in both directions.

A third factor to bear in mind is how well a board can overlap independent PCIe transfers and kernel execution. Although it is a common belief that independent tasks (i.e., a PCIe transfer and a kernel) enqueued on different command queues could be carried out whenever the resource (i.e., bus or chip) is free, this is not always true. In fact, even if the FPGA board we used can start one PCIe transfer and many kernels at the same time, for some yet not clear reasons, it is not able to start a kernel execution if a PCIe transfer is in process. Consequently, stalling the FPGA chip in executing following kernels in each cycle of the software pipeline, and the facto, exacerbating the kernel execution time; thus, the cycle time and time-to-solution of the entire software. Therefore, justifying the difference between the box plot of the kernels execution cycle time and the median of the mere kernel execution in Fig. 7b where there is no dependencies between PCIe transfers and kernels execution and all steps of the software pipeline could, in principle, be fully overlapped.

In conclusion, we can affirm that, for this k-mer counting algorithm, the two FPGAs are slower than the GPU because of the slower on-board memory and due to the inferior PCIe capabilities. Also, we can deduce that future FPGAs board equipped with faster memory and adequate PCIe subsystem, including a dual memory engine to allow full overlapping of transfers and kernels, could come closer to the performance of the GPUat a lower power envelop. Making FPGAs an asset for future energy-efficient genomics workloads.

6. Conclusions

In this paper, we presented the result of porting and optimizing a k-mer frequency counting workload from GPUs to FPGAs boards. Even if the code is written in OpenCL, due to the fundamental differences between the two hardware architectures the work comprised different programming techniques such as kernel replication, inter-kernel communication using Intel's OpenCL channels, and loop unrolling. Additionally, we described how data partitioning could be used to expand similar designs to support an arbitrary number of FPGAs without the need to recompile the design. We presented a detailed scalability analysis of the kernel replication technique that showed how performance scales linearly until the off-chip DDR4 memory of the FPGAs become the bottleneck. For the same reason, this same technique proved very poor improvement for Bloom Filter kernels where the filters where stored off-chip.

Overall results showed that one FPGA is not enough to improve the baseline CPU-only implementation but that two FPGAs yield an improvement of 1.32x in both time- and energy-to-solution. Results also showed how the GPU outperforms the two FPGAs with an improvement of 1.77x in time-to-solution but, due to higher power consumption, of a lower 1.49x in energy-to-solution. We carried out a comprehensive analysis that took in consideration PCIe capabilities and the on-board memory of both FPGA and GPU boards to uncover that the FPGA chip itself is not the main bottleneck. With power samples collected throughout the executions we revealed how the offloading to FPGAs didn't increase the power envelope but instead, in some moments, even lowered the power consumption of the entire node of few Watts. Based on these findings, we outlined how next-generation FPGA boards constitute an asset for energy-efficient genomics workloads.

We are currently working on methods to improve data locality for the Bloom filters and that could make the FPGAs more competitive. Besides, we are exploring how NVMe over Fabrics [19] and pooled accelerators can reduce the total cost of ownership without overly penalizing the execution time.

Acknowledgment

We are grateful to Intel who provided the Optane drive and to SandDisk for lending the HGST card.

Funding

This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement s No 639595); the Ministry of Economy of Spain under contract TIN2015-65316-P and Generalitat de Catalunya, Spain under contract 2014SGR1051; the ICREA, Spain Academia program; and the BSC-CNS Severo Ochoa, Spain program (SEV-2015-0493).

Appendix. OpenCL kernels

Algorithm 1 GPU Encode Kernel

```
Input:  in (DNA input read and quality markers).
Output: mid (Buffer of k-mers generated from the input DNA),
        histo (Number of k-mers in each partition),
        wg_offsets_mid (Work-groups offsets in the middle buffer),
        wg_offsets_out (Work-groups offsets in the output buffer).
procedure ENCODE
    // Cohesively read all the input for this work-group to local memory
    l_in = cohesive_read_input (in, get_group_id(0), get_local_id(0))
    barrier(CLK_LOCAL_MEM_FENCE)

    // Generate all k-mers and build a private histogram
    for i = 0, KMERS_PER_READ do
        p_kmers[i] = generate_kmer(l_in, get_local_id(0), i)
        if p_kmers[i] == INVALID then continue
        end if
        p_histo[get_partition_id(p_kmers[i])]++
    end for

    // Merge private histograms into a local histogram and store
    // the result to be used as offset by this thread to shuffle k-mers locally
    for i = 0, NUM_PARTITIONS do
        p_offsets[i] = atomic_add(l_histo[i], p_histo[i])
    end for
    barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE)

    if get_local_id(0) == 0 then
        // Merge local histograms into a global histogram and store
        // the result to be used as offset by this work-group in the mid buffer
        for i = 0, NUM_PARTITIONS do
            wg_offsets_out[i] = atom_add(histo[i], l_histo[i])
        end for

        // Perform an exclusive sum-prefix on the local histograms applying
        // an offset and store the result to be used by this work-group
        // in the mid buffer
        wg_offsets_mid[0] = get_group_id(0) * get_local_size(0) *
                            KMERS_PER_READ
        for i = 0, NUM_PARTITIONS do
            wg_offsets_mid[i+1] = wg_offsets_mid[i] + l_histo[i]
        end for
    end if
    barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE)

    // Write generated k-mers shuffling them locally - within a work-group
    for i = 0, KMERS_PER_READ do
        if p_kmers[i] == INVALID then continue
        end if
        offset = wg_offsets_mid[i] + p_offsets[i]
        mid[offset] = p_kmers[i]
        p_offsets[i]++
    end for
end procedure
```

Algorithm 2 GPU Shuffle Kernel

```
Input:  mid (Buffer of k-mers generated in the encode kernel),
        offsets (Consumer threads offsets in the output buffer),
        wg_offsets_mid (Work-groups offsets in the middle buffer),
        wg_offsets_out (Work-groups offsets in the output buffer).
Output: out (Buffer of k-mers shuffled by partitions).
procedure SHUFFLE
    for i = 0, NUM_PARTITIONS do
        // Cohesively read all k-mers from this work-group to write
        // and shuffle them globally
        num_items = wg_offsets_mid[i+1] - wg_offsets_mid[i]
        wg_offset_out = offsets[i] + wg_offsets_out[i]
        for j = get_local_id(0), num_items, j += get_local_size(0) do
            out[wg_offset_out+j] = mid[wg_offsets_mid[i]+j]
        end for
    end for
end procedure
```

Algorithm 3 FPGA Encode Kernel (Producer)

```
Input:  in (DNA input read and quality markers),
        num_reads (Number of DNA reads in input).
procedure ENCODE
    for chunk_id = 0, num_reads/READS_PER_SUBCHUNK do
        // Load READS_PER_SUBCHUNK DNA reads in bulk
        // from the global (off-chip) to the on-chip memory
        read_input (in, chunk_id, l_in)

        // Generate all k-mers from the DNA reads in the subchunk in parallel
        #pragma unroll READS_PER_SUBCHUNK
        for r_id = 0, READS_PER_SUBCHUNK do
            for k_id = 0, KMERS_PER_READ do
                kmers[r_id][k_id] = generate_kmer(l_in, r_id, k_id)
            end for
        end for

        // Send one k-mer to each shuffle kernel at every cycle
        for k_id = 0, KMERS_PER_READ do
            #pragma unroll READS_PER_SUBCHUNK
            for r_id = 0, READS_PER_SUBCHUNK do
                write_channel_intel(channel[r_id], kmers[r_id][k_id]);
            end for
        end for
    end for
end procedure
```

Algorithm 4 FPGA Shuffle Kernels (Consumers)

```
Input:  num_reads (Number of DNA reads in input),
        pid_min (Minimum pid for this device),
        pid_max (Maximum pid for this device).
Output: out (Buffer of k-mers shuffled by partitions),
        histo (Number of k-mers in each partitions).
procedure SHUFFLE #N
    histo_cache[NUM_PARTITIONS] = {0}
    kmers_cache[NUM_PARTITIONS][8] = {{INVALID}}
    num_kmers = num_reads * KMERS_PER_READ /
                READS_PER_SUBCHUNK

    // Digest one k-mer at each cycle
    for kmer_id = 0, num_kmers do
        kmer = read_channel_intel(channel[N])
        pid = get_partition_id(kmer)

        // Store the k-mer in the on-chip memory and count only valid ones
        kmers_cache[pid][histo_cache[pid] & 0x7] = kmer
        if kmer != INVALID && pid_min <= pid && pid <= pid_max then
            histo_cache[pid]++

            // Flush k-mers from the on-chip memory to global memory
            // in bulk using 512-bit wide memory transactions
            if (histo_cache[pid] & 0x7) == 0x0 then
                out[(histo_cache[pid]) >> 3] = kmers_cache[pid]
            end if
        end if
    end for

    // Flush on-chip caches to the global memory
    for pid = pid_min, pid_max do
        out[(histo_cache[pid]) >> 3] = kmers_cache[pid]
        histo[pid] = histo_cache[pid]
    end for
end procedure
```

References

[1] 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 Biotechnol. 32 (11) (2014) 1106–1112.

[2] N. Cadenelli, J. Polo, D. Carrera, Accelerating K-mer frequency counting with GPU and non-volatile memory, in: 2017 IEEE 19th International Conference on High Performance Computing (HPCC), 2017, pp. 434–441, http://dx.doi.org/10.1109/HPCC-SmartCity-DSS.2017.57.

[3] 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 Res. 20 (2) (2010) 265–272.

[4] D.R. Kelley, M.C. Schatz, S.L. Salzberg, Quake: quality-aware detection and correction of sequencing errors, Genome Biol. 11 (11) (2010) R116.

[5] IntelFPGA Channelizer Design Example. URL https://www.altera.com/support/support-resources/design-examples/design-software/opencl/channelizer.html.
[6] Z. Wang, B. He, W. Zhang, A study of data partitioning on OpenCL-based FPGAs, in: 2015 25th International Conference on Field Programmable Logic and Applications (FPL), 2015, pp. 1–8, http://dx.doi.org/10.1109/FPL.2015.7293941.
[7] S. Sridharan, P. Durante, C. Faerber, N. Neufeld, Accelerating particle identification for high-speed data-filtering using OpenCL on FPGAs and other architectures, in: 2016 26th International Conference on Field Programmable Logic and Applications (FPL), 2016, pp. 1–7, http://dx.doi.org/10.1109/FPL.2016.7577351.
[8] O.J. Arndt, F.D. Trger, T. Mo, H. Blume, Portable implementation of advanced driver-assistance algorithms on heterogeneous architectures, in: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017, pp. 6–17, http://dx.doi.org/10.1109/IPDPSW.2017.100.
[9] Accelerating Genomics Research with OpenCL and FPGAs. URL https://www.intel.com/content/www/us/en/healthcare-it/solutions/documents/genomics-research-with-opencl-and-fpgas-paper.html.
[10] F.B. Muslim, L. Ma, M. Roozmeh, L. Lavagno, Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis, IEEE Access 5 (2017) 2747–2762, http://dx.doi.org/10.1109/ACCESS.2017.2671881.
[11] H.R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, S. Matsuoka, Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs, in: SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 409–420, http://dx.doi.org/10.1109/SC.2016.34.
[12] S.O. Settle, High-performance Dynamic Programming on FPGAs with OpenCL, 2013.
[13] E. Rucci, C. Garcia, G. Botella, A.D. Giusti, M. Naiouf, M. Prieto-Matias, Smith-Waterman protein search with OpenCL on an FPGA, in: 2015 IEEE Trustcom/BigDataSE/ISPA, Vol. 3, 2015, pp. 208–213, http://dx.doi.org/10.1109/Trustcom.2015.634.
[14] A. Sirasao, E. Delaye, R. Sunkavalli, S. Neuendorffer, FPGA Based OpenCL Acceleration of Genome Sequencing Software, 2015.
[15] L.D. Tucci, K. O'Brien, M. Blott, M.D. Santambrogio, Architectural optimizations for high performance and energy efficient Smith-Waterman implementation on FPGAs using OpenCL, in: Design, Automation Test in Europe Conference Exhibition (DATE), 2017, 2017, pp. 716–721, http://dx.doi.org/10.23919/DATE.2017.7927082.
[16] E. Houtgast, V.M. Sima, Z. Al-Ars, High performance streaming Smith-Waterman implementation with implicit synchronization on intel FPGA using OpenCL, in: 2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE), 2017, pp. 492–496, http://dx.doi.org/10.1109/BIBE.2017.000-6.
[17] C. Poirier, B. Gosselin, P. Fortier, DNA assembly with de bruijn graphs on FPGA, in: 2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), 2015, pp. 6489–6492, http://dx.doi.org/10.1109/EMBC.2015.7319879.
[18] N. Mcvicar, C.C. Lin, 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), 2017, pp. 203–210, http://dx.doi.org/10.1109/FCCM.2017.23.
[19] A. Call, J. Polo, D. Carrera, F. Guim, S. Sen, Disaggregating non-volatile memory for throughput-oriented genomics workloads, in: Workshop on Advances in High-Performance Bioinformatics, Systems Biology Co-Located with the 24th International European Conference on Parallel and Distributed Computing, 2018.

Nicola Cadenelli received the MS degree at the Università degli Studi di Brescia (UniBS), Italy in 2014. During his master studies, he spent one year as a visiting student at the University of Applied Sciences of Leipzig, Germany in 2012, and one semester at the Jülich Supercomputing Center, Germany in 2014. Currently, he is a Ph.D. Student at the Technical University of Catalonia (UPC), Spain and part of the "DataCentric Computing" research group at the Barcelona Supercomputing Center (BSC), Spain. In 2018, he was a summer visiting student at the Massachusetts Institute of Technology (MIT), USA. His research revolve around the scalability, both vertical and horizontal, of real-world data-intensive workloads.

Zoran Jakšić is a postdoctoral researcher in Barcelona Supercomputing Center (BSC). His primary research interest is the acceleration of compute-intensive workloads with FPGAs and GPUs. Before joining BSC, he was with Broadcom Networks where he worked as an RTL verification engineer for a year. He obtained a Ph.D. from Universitat Politecnica de Catalunya in 2015, and for that research, he was awarded by Intel E.U. Doctoral Student Honour Programme.

Dr. Jordà Polo received his bachelor's degree in Computer Science from Universitat Politècnica de Catalunya in 2009. He then started his graduate work with Professors David Carrera and Yolanda Becerra at the Barcelona Supercomputing Center (BSC), completing his Ph.D. in 2014. His research focused on how to manage and model the performance of data-intensive workloads. He is currently working as a Postdoc in the same institution, leading the research in software-defined infrastructures and data-centric architectures for genomics workloads.

David Carrera received the MS degree at the Technical University of Catalonia (UPC) in 2002 and his PhD from the same university in 2008. He is an associate professor at the Computer Architecture Department of the UPC. He is also the Head of the "DataCentric Computing" research group at the Barcelona Supercomputing Center (BSC). His research interests are focused on the performance management of data center workloads.

In 2015 he was awarded an ERC Starting Grant for the project HiEST (1.5Mâ, 2015–2020), and ICREA Academia award (2015–2020) and an ERC Proof of Concept grant ('Hi-OMICS') in 2017 to explore the commercialization of an SDI orchestrator for genomics workloads. He has participated in several EU-funded projects and has led the team at BSC that has developed the Aloja project (aloja.bsc.es) and the servIoTicy platform (servioticy.com). He is the PI for several industrial projects and collaborations with IBM, Microsoft and Cisco among others.

He was a summer intern at IBM Watson (Hawthorne, NY) in 2006, and a Visiting Research Scholar at IBM Watson (Yorktown, NY) in 2012.

He received an IBM Faculty Award in 2010. He is an IEEE and ACM member.