HARDWARE/SOFTWARE CO-DESIGN FOR DATA-INTENSIVE GENOMICS WORKLOADS

NICOLA CADENELLI

Dissertation submitted in partial fulfillment of the requirements for the Doctoral Degree in Computer Architecture

Universitat Politècnica de Catalunya - BarcelonaTECH

October 2019

Nicola Cadenelli: *Hardware/Software Co-design for Data-Intensive Genomics Workloads*.
Dissertation submitted in partial fulfillment of the requirements for the Doctoral Degree in Computer Architecture, 13th October 2019.

SUPERVISORS:
David Carrera
Jordà Polo

AFFILIATION:
Department of Computer Architecture
Universitat Politècnica de Catalunya - BarcelonaTECH

LOCATION:
Barcelona, Spain

COLOPHON

The image in the title page is a derivative of "Hospital de Sant Pau" by Nicky Boogaard, used under CC BY 2.0. This image is licensed under CC BY 2.0 by Nicola Cadenelli.

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*.

Se ghè de na, ghè de na!

Where we love is home,
home that our feet may leave, but not our hearts.

— Oliver Wendell Holmes, Sr.

ABSTRACT

Since the last decade, the main components of computer systems have been evolving, diversifying, to overcome their physical limits and to minimize their energy footprint. Hardware specialization and heterogeneity have become key to design more efficient systems and tackle ever-important problems with ever-larger volumes of data. However, to fully take advantage of the new hardware, a tighter integration between hardware and software, called hardware/software co-design, is also needed. Hardware/software co-design is a time-consuming process that poses its challenges, such as code and performance portability. Despite its challenges and considerable costs, it is an effort that is crucial for data-intensive applications that run at scale. Such applications span across different fields, such as engineering, chemistry, life sciences, astronomy, high energy physics, earth sciences, et cetera.

Another scientific field where hardware/software co-design is fundamental is genomics. Here, modern DNA sequencing technologies reduced the sequencing time and made its cost orders of magnitude cheaper than it was just a few years ago. This breakthrough, together with novel genomics methods, will eventually enable the long-awaited personalized medicine. Personalized medicine selects appropriate and optimal therapies based on the context of a patient's genome, and it has the potential to change medical treatments as we know them today. However, the broad adoption of genomics methods is limited by their capital and operational costs. In fact, genomics pipelines consist of complex algorithms with execution times of many hours per each patient and vast intermediate data structures stored in main memory for good performance. To satisfy the main memory requirement genomics applications are usually scaled-out to multiple compute nodes. Therefore, these workloads require infrastructures of enterprise-class servers, with entry and running costs that that most labs, clinics, and hospitals cannot afford. Due to these reasons, co-designing genomics workloads to lower their total cost of ownership is essential and worth investigating.

This thesis demonstrates that hardware/software co-design allows migrating data-intensive genomics applications to inexpensive desktop-class machines to reduce the total cost of ownership when compared to traditional cluster deployments. Firstly, the thesis examines algorithmic improvements to ease co-design and to reduce workload footprint, using NVMs as a memory extension, and so to be able to run in one single node. Secondly, it investigates how data-intensive algorithms can offload computation to programmable accelerators (i.e., GPUs and FPGAs) to reduce the execution time and the energy-to-solution. Thirdly, it explores and proposes techniques to substantially reduce the memory footprint through the adoption of flash memory to the point that genomics methods can run on one affordable desktop-class machine.

To demonstrate this thesis, we do the exercise to co-design SMUFIN, a state-of-the-art real-world genomics method that was originally deployed on 16 nodes MareNostrum 3, where, per each patient, it needed around 10 hours and 56 kWh to complete its execution. Thanks to algorithmic improvements, an NVM used as main memory extension, and accelerators, we made it possible to execute SMUFIN on one single enterprise-node with 512 GB of main memory in 9

hours and as few as 4.3 kWh, a 13.1x improvement. However, we were able to run SMUFIN on a desktop-class machine only thanks to the adoption of NVMe as an alternative to main memory. In this affordable node with a 6-core i7 and only 32 GB of main memory, SMUFIN suffers a considerable slow-down, requiring 22.4 hours, but it consumes only 2.4 KWh, a 23.3x improvement compared to the original deployment. Compared to the single enterprise-node, this desktop machine costs only 1/4 as much, and requires only approximately 1/2 of energy per patient. As a result, a cluster of multiple desktop-class machines costs half as much compared to a cluster of servers and consumes half as much energy while maintaining a similar throughput. These results prove that hardware/software co-design allows significant reductions in the total cost of ownership of data-intensive genomics methods, easing their adoption on large repositories of genomes and also on the field.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

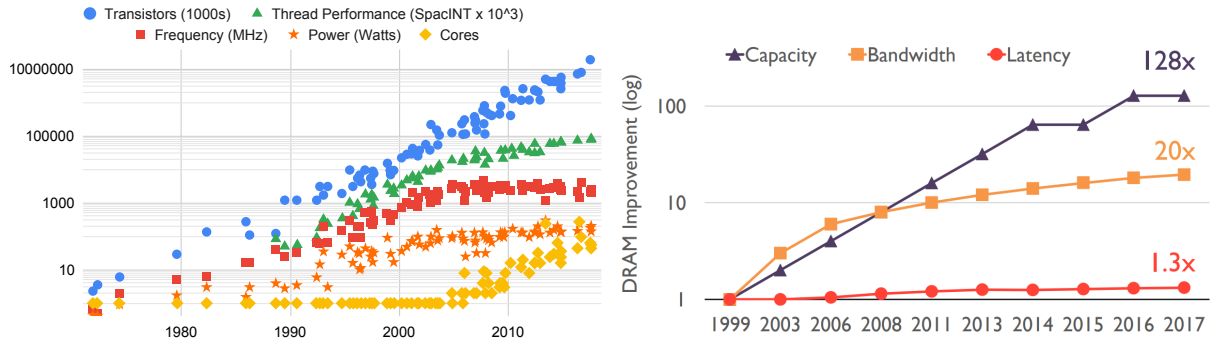
INTRODUCTION

1.1 THESIS CONTEXT

Computer systems are in continuous evolution, and they have come a long way. In the last decade, the main components of computer systems, such as processing units and memory, reached physical limits and have been evolving, diversifying, to overcome these limits, but also to minimize their energy footprint and the humongous costs related to run power-hungry systems at scale.

One example of these physical limits is the so-called "power wall", or breakdown of Dennard scaling or MOSFET scaling, reached in the middle of the last decade [16, 144]. Due to this limit, the more we scale the lithography process of transistors, the more impractical it is to keep increasing the clock frequency of microprocessors because they would dissipate too much heat that commodity cooling cannot manage. As Figure 1a shows, this forced a dramatic change in the design of microprocessors. In particular, the impracticality of higher frequency led manufacturers, always seeking for better performance, to move from single-core to multi-core chips [110]. At the same time, many-core chips, such as GPUs, also started to be used for general-purpose computing, and they now offer chips with thousands of cores. In pursuit of even more power-efficient computing, the interest also expanded to more specialized chips, such as FPGAs, and also to application-specific chips ASIC (e.g., TPU).

A second physical limit is the ending of DRAM scaling; which makes DRAM capacity, cost, and power consumption difficult to scale further. Furthermore, as Figure 1a shows, DRAM latency is mildly reducing; augmenting the speed gap between processors and main memory. Manufacturers are still striving to find alternatives to DDRx SDRAM that offer lower latency, higher capacity, and lower power at the same time. In the meantime, new kinds of memory technology have



(a) Microprocessor [110]

(b) DRAM [25, 26, 121, 17, 101]

Figure 1: (§1) Microprocessor and DRAM trends in the last decades.

emerged, each with its trade-off. An example is Reduced Latency DRAM (RLDRAMx) which offers 4x lower latency than the DDRx DRAM chip, but its cost/bit is significantly higher, up to 39x [25]. A second example is the high bandwidth memories, such as GDDR and HBM, that were created to deliver high bandwidth to feed the thousands of compute units of accelerators. Compared to DDRx, these memories offer lower power consumption but worse latency, which is not entirely an issue given the parallel nature of accelerators. Another example is the non-volatile memories, such as NAND memory and Phase-Change Memory (PCM), that were created to provide much lower power consumption and affordable price/GB ration than DDRx. Compared to DDRx, NVM memories offer better power consumption and prices, but orders of magnitude higher latency and much lower bandwidth. For this reason, they are rapidly being adopted as a new memory tier between main memory and storage to provide significant capacity with low power consumption and low cost.

All these major shifts in the hardware landscape are allowing humankind to advance scientific computing, and they are enabling us to tackle ever-important problems with ever-larger volumes of data. At the same time, these shifts demand radical changes also on the software side. For instance, to improve the software performance at lower clock frequency, traditionally sequential code needed to be adapted to take advantage of multiple cores. To harness accelerators and their forms of parallelism, new programming models and algorithms have been developing. To take advantage of the new non-volatile memory, a new protocol and changes in user and kernel code have been carrying out. Therefore, while hardware heterogeneity has become key to design more efficient systems, it requires tighter integration between hardware and software. This integration is what we call hardware/software co-design. From an appliance standpoint, hardware/software co-design is the cooperative design of the hardware and software aimed to meet some requirements by exploiting possible synergies between hardware and software. A non-exhaustive list of possible requirements includes: make the execution time shorter, reduce the energy-to-solution, or do not exceed a set power consumption limit. While the former requirement is the most common, application executed at a large scale might be more interested in reducing the total energy consumed, even if this might imply longer execution times.

While hardware specialization and diversification are fundamental for more efficient systems, they introduce new challenges. First, it is difficult to identify the best hardware for a given workload. This challenge requires a broad view and understanding of all the different hardware. Plus, it takes a good knowledge of the application and its external requirements (e.g., need to execute the application on-premises). Second, if the software uses proprietary programming models or libraries, the portability of the application to similar hardware from different vendors is restricted. Third, there are no guarantees on performance portability when moving one application from one platform to another. Finally, co-design is an iterative and time-consuming process that needs to evolve together with new and emerging technologies that might change the hardware landscape. Although these challenges, hardware/software co-design is particularly crucial for data-intensive applications that run at scale, such as climate models, cosmological simulations, data compression, molecular dynamics, et cetera.

Another field where hardware/software co-design is fundamental is genomics. Here, modern DNA sequencing technologies greatly reduced the sequencing time and, like Figure 2 shows, made its cost orders of magnitude cheaper than it was just a few years ago. The rate of progress

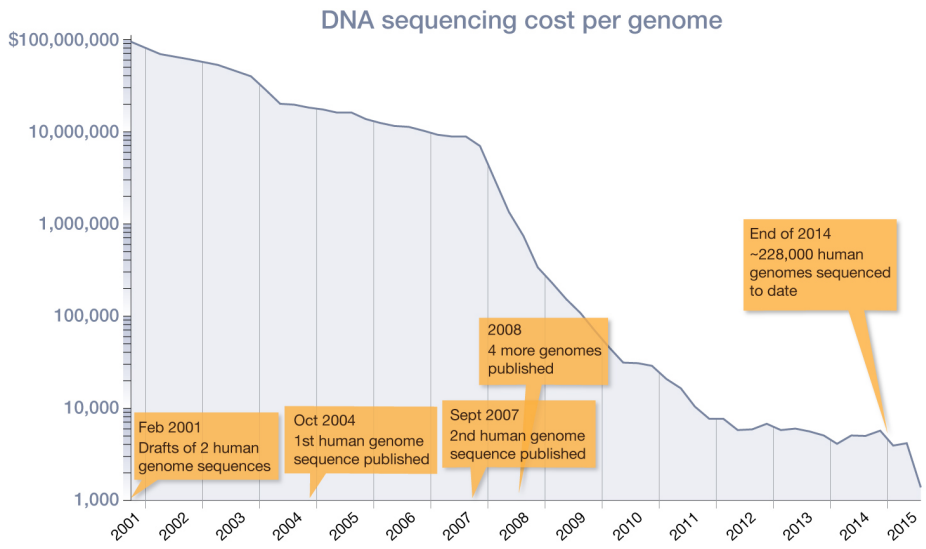


Figure 2: (§1) As DNA sequencing costs have plummeted, the number of sequenced genomes has increased dramatically. Cost data from NHGRI Genome Sequencing Program [50], information about genome sequences published before 2011 from [45], and chart from Genetic Science Learning Center [24].

in this field is unparalleled and, as DNA sequencing becomes more affordable and available, more and more human genomes are sequenced. This breakthrough, together with novel genomics methods, will eventually enable the long-awaited personalized medicine, which selects appropriate and optimal therapies based on the context of a patient's genome; potentially changing medical treatments as we know them today. However, the broad adoption of such methods is limited by their capital and operational costs. In fact, genomics pipelines consist of complex algorithms with execution times of many hours per each patient and vast intermediate data structures stored in DRAM for good performance; requiring to scaled-out to multiple compute nodes to satisfy the main memory requirement. Therefore, these workloads require infrastructures of enterprise-class servers, with entry and running costs that quickly become unmanageable with many patients and that most labs, clinics, and hospitals cannot afford. This poses two challenges. First, the processing of large repositories of genomes becomes expensive, and it requires a considerable amount of resources, even for a supercomputing facility. Second, the adoption of such methods in the field by physicians and researchers is limited by energy and hardware costs. Due to these reasons, we believe that co-designing genomics workload to lower their total cost of ownership is essential and worth investigating.

This thesis wants to demonstrate that **hardware/software co-design allows migrating data-intensive genomics applications to inexpensive desktop-class machines to reduce the total cost of ownership when compared to traditional cluster deployments**.

To prove this statement, we do the exercise to co-design SMUFIN [95], a state-of-the-art reference-free genomics algorithm. SMUFIN is a leading method for detecting mutations acquired by an organism after conception instead of inherited from a parent, called somatic mutations. The

peculiarity of the SMUFIN method is that it does not require a full genome reconstruction or alignment against a reference; like other methods do. The SMUFIN method is supposed to run at scale on thousands of patients to identify relations between DNA mutations and cancerous tumors. These relations are key to discover the origin of tumors and must be understood to predict the types of cancer a patient might develop; which is fundamental to enable effective and personalized treatments. Nevertheless, the software implementation of this method takes multiple hours on multiple enterprise-class nodes consuming tens of KWh per every single patient. As an example, we estimated that to process the over 100'000 genomes of the European Genome-phenome Archive (EGA) [74] the original deployment of SMUFIN in 16 nodes of `MareNostrum 3` would require over 256 millions CPU core hours and over 5.6 GWh. Numbers that directly translate to excessive costs that limit the broad adoption of the method.

Like other data-intensive applications, the software implementation of the SMUFIN method has characteristics and requirements that differ from those of more traditional high-performance applications. Some of the differences include but are not limited to: the application is data-intensive, and the main memory requirements dictate the number of compute nodes required; there is minimal communication between compute nodes; the amount of I/O increases with the number of nodes used.

1.2 THESIS CONTRIBUTIONS

This thesis produces three contributions that all revolve around the same main topic: **hardware/software co-design of data-intensive workloads**. Each contribution delves into a separate co-design; investigating distinct challenges and proposing solutions to different problems. The contributions can be summarized as follows:

C1: Examine algorithmic improvements to ease co-design and to reduce workload footprint using NVMs as a memory extension to be able to run in one single node.

C2: Investigate how data-intensive algorithms can offload computation to programmable accelerators (i.e., GPUs and FPGAs) to reduce the execution time and the energy-to-solution.

C3: Explore and propose techniques to substantially reduce the memory footprint through the adoption of flash memory to the point that data-intensive workload can run on one single desktop-class machine that uses only consumer-grade hardware.

Even these three contributions are independent of each other, in the scope of this thesis they are part of one incremental work and, as Figure 3 illustrates, they all follow the same trajectory toward a lower total cost of ownership (TCO). This is accomplished by reducing the capital expenditures of the hardware, lowering the operating expenses of the electric bill, or both at the same time. The first contribution lowers both costs at the same time; reducing the workload footprint to just one node to minimize the waste of resources and to reduce the energy-to-solution. The second contribution reduces the operating costs; by slightly increasing the capital cost to buy additional hardware accelerators that can be more power-efficient. Finally, the third contribution lowers both costs; targeting inexpensive and low-power desktop-class machines rather than enterprise nodes. Here, the much lower computational capability translates to longer execution

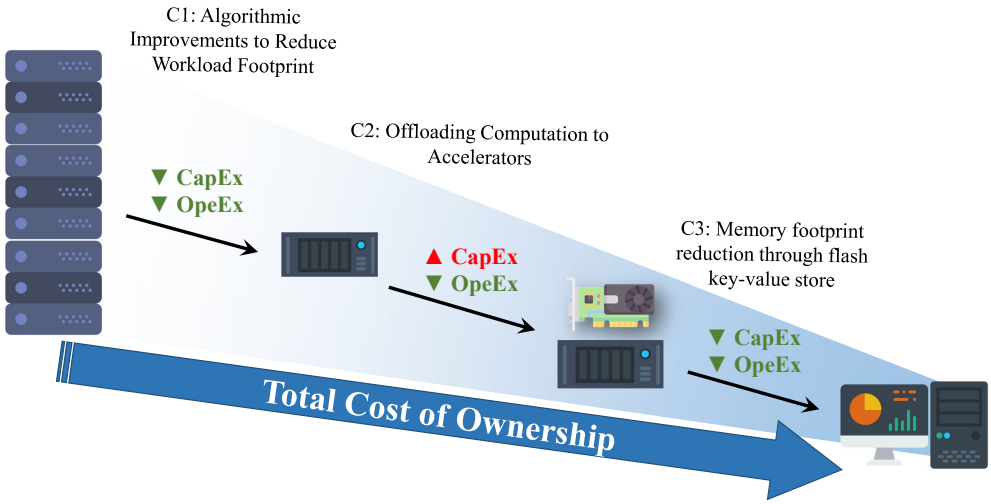


Figure 3: (§1.2) Graphic representation of the three contributions of this thesis, all following a trajectory towards a lower Total Cost of Ownership (TCO).

times. However, the power consumption is reduced to the point that the energy-to-solution is also reduced despite the longer time-to-solution. Besides, this class of machine does not need an air-cooled server room; further reducing the total cost of ownership. This third contribution is the key to ease the adoption of the SMUFIN method – and potentially other methods – by researchers in their labs and enables genomics analysis on-premises by physicians.

1.2.1 *Algorithmic Improvements to Reduce Workload Footprint*

The **first contribution** of this thesis is a set of algorithmic improvements that aim to reduce workload footprint and ease the co-design of data-intensive applications. This contribution is motivated by the observation that applications have different stages, each one with its footprint and requirements. However, monolithic applications have to request enough resources to meet the requirement for the execution of all stages; wasting resources. This problem is particularly relevant for data-intensive applications with a memory footprint that exceeds the amount of memory in a compute node. In fact, in such cases, the only way to meet the memory requirement is to scale-out the application to multiple nodes to obtain enough aggregated memory; possibly requesting more compute resources than needed; wasting CPU time and energy. This first contribution of this thesis aims to reduce the footprint of workloads by minimizing this waste of resource; thus, reducing the operational costs. Besides, it suggests techniques to lower the main memory requirements to enable genomics applications in one single node; thus, reducing the capital expenses.

The algorithmic improvements of this first contribution are three. First, to minimize the waste of resources, this thesis proposes to redesign the application using a modular structure made of multiple stages, called *units*. This structure allows the sizing of the workload footprint based on the resource demands of each distinct unit. To fit the execution into one single compute node with less memory than required this structure also admits breaking most demanding units into multiple *partitions*, to be executed sequentially one after another. Second, to reduce DRAM

requirement of the most demanding units, this thesis suggests a manual swapping mechanism to flush data structures from the main memory to non-volatile memory. Third, to reduce the memory requirement even further, this thesis analyses alternative and memory-efficient data structures. While the first two improvements are general, and applications from other domains, beyond genomics, could benefit from it, the third improvement is more specific to genomics workloads.

Results showed that we were able to reduce the workload footprint of SMUFIN to the point that it can be executed in just one node with 512 GB of main memory and one non-volatile memory of 1.6 TB (used as memory extension) rather than the usual from 16 nodes, each with 128 GB of main memory, at a similar time-to-solution and more than 10x lower aggregate node time and energy-to-solution. A considerable part of the algorithm proved to be CPU bound; suggesting that accelerators could be leveraged to offload some of the compute-intensive work.

The work performed in this area resulted in the following publications:

> [20] Nicola Cadenelli, Jordà Polo, and David Carrera. Accelerating K-mer Frequency Counting with GPU and Non-Volatile Memory. In *2017 IEEE 19th International Conference on High-Performance Computing (HPCC)*, pages 434-441, December 2017.
>
> [22] David Carrera Perez, Jordà Polo, Nicola Cadenelli, David Torrents Arenales, and Mercè Planas. A Computer-Implemented and Reference-Free Method for Identifying Variants in Nucleic Acid Sequences, January 2018. Patent No.: WO/2018/007034.

1.2.2 *Offloading Computation to Accelerators*

The **second contribution** of this thesis is a study that aims to investigate the offloading of computation to programmable accelerators like GPUs and FPGAs for data-intensive applications. This contribution is motivated by the observation that data-intensive applications that process large volumes of data can easily become CPU bound, particularly when being executed in one single node. Therefore, programmable accelerators that offer a high degree of parallelism can be efficiently used to offload some of the computation. However, their adoption requires an arduous exploration to determine the best kind of accelerator and a careful co-design that aims to overcome possible bottlenecks. This second contribution targets accelerators that have significant costs and, usually, increase the power consumption of a compute node. However, their adoption is justifiable if it yields a reduction of the execute time, thus lowering the energy-to-solution. Motivation that is particularly true for applications meant to run at scale, where the energy costs are more relevant than the capital expenditures.

The topics of the study of this second contribution are many. First, the study focuses on offloading CPU intensive operations. Then, to overcome memory constraints of accelerators, it proposes and prototypes a method to virtualize accelerators' memory in host DRAM. Next, it delves into portability issues that arise when porting the code from one kind of accelerator to another. In this scope, the study also shows how single components of accelerator (i.e., off-chip memory and PCIe connection) can hinder performance, particularly for data-intensive applications. Finally, the study analyses and discusses power and energy consumption with and without accelerators.

Thanks to the modular structure given to the application in the first contribution, the study is isolated to only a few units of the SMUFIN pipeline.

Results showed that offloading to GPUs and FPGAs can reduce both time- and energy-to-solution with improvements up to 1.8x for both metrics. A thorough analysis of the application uncovered that in the faster execution, with one GPU, the main bottleneck is the main memory latency of the CPU that must be paid to randomly access the principal data structures stored in main memory; suggesting that the algorithm could benefit from building the main hash table in a different manner that minimizes the random updates.

The work carried out in this field produced the following publications:

> [20] Nicola Cadenelli, Jordà Polo, and David Carrera. Accelerating K-mer Frequency Counting with GPU and Non-Volatile Memory. In *2017 IEEE 19th International Conference on High Performance Computing (HPCC)*, pages 434-441, December 2017.
>
> [21] Nicola Cadenelli, Zoran Jaksić, Jordà Polo, and David Carrera. Considerations in using OpenCL on GPUs and FPGAs for throughput-oriented genomics workloads. *Future Generation Computer Systems*, 94:148-159, 2019. ISSN 0167-739X.

1.2.3 *Memory footprint reduction through flash key-value store*

The **third contribution** of this thesis is the exploration of co-design aimed to reduce the memory footprint of data-intensive applications through a flash key-value store. This key-value store, optimized for NAND-flash storage, provides an alternative way to store the TB-size intermediate data structures that are usually kept in main memory. This contribution is motivated by two main observations. First, new flash NVM technologies are getting better and cheaper. Second, as of today, NVMs constitute a valid alternative to DDR SDRAM, which instead is a stable technology whose prices are not decreasing. However, replacing main memory with flash storage does not come for free and has its challenges. In particular, one needs to overcome the four orders of magnitude longer latency and two orders of magnitude larger access granularity of flash storage over DRAM.

This third contribution targets commodity PCIe NVMe and SATA-III SSD drives that can be placed into desktop-class machines. Compared to enterprise machines with hundreds of GB of main memory and high-end CPUs, target desktop-class machines consume less power at the cost of considerably lower compute capabilities that could lead to noticeable longer execution times. However, these target machines are profitable if, through co-design efforts, it is possible to limit the performance penalty on the selected workload so that the energy-to-solution is still lower than on the faster enterprise machine. Furthermore, desktop-class machines considerably reduce the capital expenditures of the hardware, and since these machines are deployed in regular offices, without the need for air-conditioned server rooms, their use further cuts capital and operational costs.

The exploration of this third contribution has different outcomes. First, it presents a method to generate and access large intermediate data structures in flash storage as opposed to DRAM memory. Second, it proposes a novel key-value store implementation that uses domain-specific information to perform efficient in-memory caching. Next, it investigates and proves how mul-

tiple CPU threads and asynchronous direct I/O with deep queues are required to exploit the high level of parallelism offered by flash storage. Finally, it provides scalability and cost analyses of a solution based on a desktop-class machine furbished with NVM drives against an enterprise-class machine with hundreds of GB of main memory.

Results showed that we reduced the system requirements for the entire SMUFIN method to the point that it can run to completion on an affordable desktop with a 6-core i7 and 32 GB of memory instead than on an enterprise machine with four times as many cores and 512 GB of DRAM. A cluster of multiple desktop-class machines costs half as much compared to a cluster of servers and consumes half as much energy while maintaining a similar throughput.

The work carried out in this third contribution assembled the following pending publication:

> [Under Revision] Nicola Cadenelli, Sang-Woo Jun, Jordà Polo, Andy Wright, David Carrera, and Arvind. Enabling genomics pipelines in commodity machines with flash storage.

This thesis is organized as follows. Chapter 2 introduces some background knowledge on general-purpose accelerators, non-volatile memories, genomics applications, and the SMUFIN method. Chapter 3 describes and evaluates the first contribution explaining the algorithmic improvements used to reduce workload footprint to the point that we can run the SMUFIN method in one single node with node. Chapter 4 presents and evaluates the second contribution describing how we offloaded part of the k-mer counting algorithm to general-purpose accelerators, such as GPUs and FPGAs, to reduce time- and energy-to-solution. Chapter 5 details and evaluates the third contribution presenting how flash storage can be leveraged to enable data-intensive genomics on commodity desktop-class machines. Finally, Chapter 6 presents conclusions, shows overall results that stretch from the first to the third contribution, and it discusses possible future works.

2

BACKGROUND

2.1 GENERAL PURPOSE ACCELERATORS

The so-called "power wall", or breakdown of Dennard scaling reached in the middle of the last decade, forced a dramatic change in the design of microprocessors. In pursuit of even more power-efficient computing, this inflection point in microprocessor design rose the interest in accelerators. As of today, accelerators can have different forms, such as Graphics Processing Units (GPUs), Application Specific Integrated Chips (ASICs), Field Programmable Gate Arrays (FPGAs), and even System-on-Chip (SoC). Each with its advantages and design trade-offs.

In this work, we focus on programmable accelerators like GPUs and FPGAs. Thanks to the parallelism offered by both kinds of these accelerators, they have been key to process the large volumes of data of today's applications: particularly for high-throughput computations that exhibit data-parallelism. 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 higher efficiency in parallel processing. Besides, they allow adding other interfaces (e.g., networking), which might be critical for some use cases and that GPUs usually do not have.

Accelerators in the form of GPUs turn the massive computational power of modern graphics chipset into general-purpose computing power. Applications requiring a tremendous number of vector operations can achieve several orders of magnitude higher performance than a conventional CPU. In order to leverage the potential of a GPU, an application needs to: be parallelizable using thousands of threads all doing the same instruction at the same time; have low as possible branch divergence; perform coalesced accesses to global memory; and make use of on-chip memory which offers higher bandwidth and lower latency than the other memories, local and global, in the GPU.

Field Programmable Gate Arrays (FPGAs) are integrated circuits that allow programming customized digital logic in the field even after manufacturing. What makes FPGAs different is that they allow the creation of sets of instructions that could be executed in a single clock cycle, while CPU or GPU would require multiple cycles. Moreover, FPGAs accomplish parallelism by duplicating the logic that each algorithm needs. In contrast, other architectures achieve parallelism replicating the entire generic computation hardware multiple times, potentially duplicating unnecessary components for a given problem. Another difference is that in FPGAs the logic of each instruction is connected to others creating complex vines of logic in a pipeline fashion. This means that after a ramp-up phase, the pipeline sustains a constant throughput that well suits stream processing.

Today, both GPUs and FPGAs can be programmed with 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 different hardware platforms is not guaranteed 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, GPU boards are typically equipped with a high-performance onboard memory (e.g., GDDR5X or HBM2) and a high-speed full-duplex interconnection (e.g., 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). While these differences seem of secondary importance, they can be critical for the data-intensive application. In fact, interconnection technologies between CPUs and accelerators are very important for data-intensive workloads, where huge volumes of data must be continuously moved back and forth in order to be processed. Here, when the interconnection bandwidth is lower than the throughput offered by computational units, the interconnection becomes the bottleneck of the application. Besides, to allow simultaneous copies in both directions and the overlapping of data transfers with computation devices must be equipped with two independent copy engines.

2.2 NON-VOLATILE MEMORIES

Disk access time can be one of the critical parts for every-day applications, and it becomes even more crucial when dealing with huge volumes of data. Non-volatile memory (NVM) storage came into play to help to provide lower access time and even energy gains over electromechanical disks. NVMs are taking the place of old rotational disks in many systems. In large-scale enterprise systems, they are being used as staging areas to overcome the I/O burstiness that characterize some applications and the huge requirement of central memory [44]. Moreover, this allow a post-execution reuse of data for in-situ data analysis [43, 126, 85, 127]. Besides being used as traditional disks with conventional file systems, NVM storage can be used as persistent memory by memory-mapping files, or even the entire device using it as a block device. In this way, data can be accessed using processor load and store instructions – as volatile memory is accessed – but would not be lost across power loss. Because of these characteristics, new non-volatile memory storage are changing the traditional memory hierarchy and are considered a third memory tier along with memory and storage [76].

In order to fully exploit the potential of NVM, software changes are required from the kernel to the application level. These changes are needed to minimize the storage software overhead caused context switches, data copies, interrupts, resource synchronization, and et cetera [118]. To this end, the Non-Volatile Memory Express (NVMe) specification was created. NVMe was architected from the ground-up to capitalize on the low latency and internal parallelism of NVM storage devices, mirroring the parallelism of contemporary CPUs and applications. As a result, NVMe reduces I/O overhead, power consumption, and latency while, at the same, increases the possible level of parallelism. Compared to previous logical interfaces, NVMe is designed to have

up to 64 thousand queues, each of those queues can have up to 64 thousand commands at the same time.

Most of today's NVM drives are made using NAND technology, which offers an endurance that could not be enough with data-intensive workloads. Emerging Phase-Change Memories (PCM) technologies can provide a better lifetime (in terms of write cycles) and better performance [23] than NAND drives. Thanks to the recent 3DXPoint, PCM is reaching the market as NVMe drives but also as NVDIMMs [136, 54]. This technology might complement, or even replace, NAND drives and DRAM. In the meantime, hybrid solutions like PCM/SSD hybrid solid-state storage [43] and PCM/DRAM hybrid main memory [140] have been proposed.

2.2.1 *Flash Devices Architecture and Practical Knowledge*

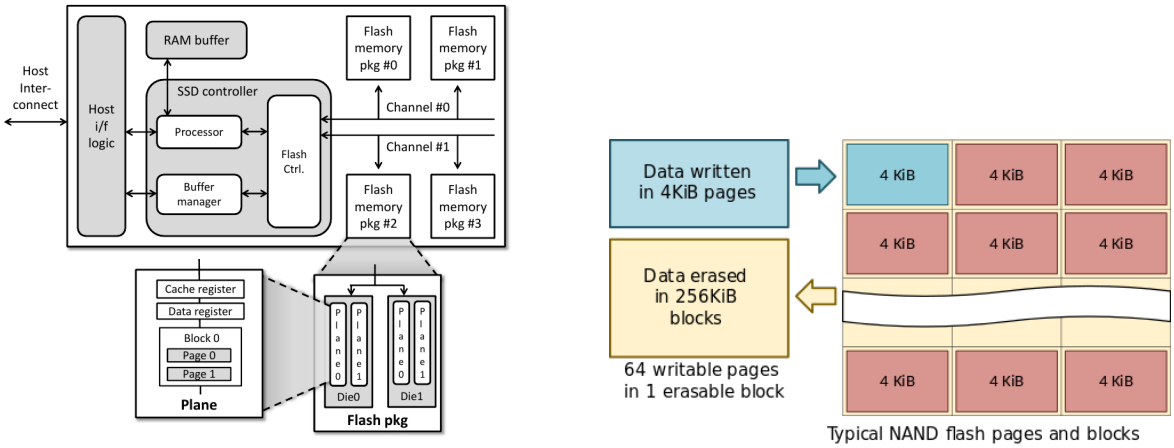


Figure 4: (§2.2.1) Internal SSD architecture [6] (left) and NAND flash memory writes data in 4 KiB pages and erases data in 256 KiB blocks [135] (right).

At first blush, one could think that getting a flash device and a way to access it is enough to start benefit from it. However, this is far from the truth, and programmers must understand the internal structure. SSD controllers receive I/O requests from the host via an interface connection (i.e., PCIe) and uses the Flash Translation Layer (FTL) to translate logical pages of incoming requests to physical pages. Memory cells are grouped into grids, called blocks, which in turn are arranged into planes. Block and page sizes vary from drive to drive, but typical flash page sizes include 4 KiB, 8 KiB, or 16 KiB, while a block can contain tens or thousands of pages. Each page can have a few bytes of metadata region to store fields like: bad block indicator, Error-Detection Code (EDC), Error-Correction Code (ECC), Block Sequence number, et cetera [12, 67]. Figure 4 shows the internals of SSD internal architecture.

Aside from understanding the internal architecture, there is some practical knowledge should be borne in mind:

- Reads and writes are aligned on page size, and it is not possible to read or write less than one page at once. Even if a read or write operation affects a few bytes, a full page is read or written, amplifying the I/O size. Therefore, to maximize throughput small writes should

be buffered to be written performing a single large write to batch all the small writes only when the buffer is full.

- A NAND-flash page can only be written if it is in the *free* state. When data is modified, the content of the page is copied into an internal register, updated, and the new version is stored in a different *free*, which means that the data is not updated in place. Once the data is written to the destination page, the original page is marked as *stale* until it is erased.
- Erases are performed at a block-granularity. Thus, it is not possible to erase individual pages, and it is only possible to erase whole blocks at once. Besides, flash memory must be erased before it can be rewritten. The erase command is triggered automatically by the garbage collection process in the drive controller when it needs to reclaim *stale* pages to make free space.
- Flash devices suffer from an undesirable phenomenon called write amplification (WA), where the actual amount of physical information written is a multiple of the logical amount intended to be written. This effect is due to the way flash devices work. In particular, to the garbage collector trying to move pages so to free up an entire block to be erased. This phenomenon increases the number of physical writes required, shortening the drives lifetime. The extra writes also consume bandwidth to the flash memory, which reduces random write performance. This phenomenon can be mitigated using techniques like over-provisioning, data compression, and implementing *TRIM* command. However, to diminish the write amplification effects, writes should be aligned with the page size and should use a granularity that is a multiple of a flash page [48].

2.3 GENOMICS

The typical input of a genomics application consists of hundreds of GB of sequenced DNA samples. Such samples are stored in a compressed form and include short sequenced strings of DNA nucleobases called *reads*. Each sequenced human genome 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 10s to 100s of bases that are represented by the four character alphabet {`A`, `C`, `G`, `T`}. Along with each base in a sequenced sample, there is also an associated score that measures its quality, doubling the amount of data. There are continuous technological improvements in sequencing, which allow longer reads, and this is likely to increase the amount of data available for further genomics analysis.

Methods to find mutations typically align reads from a sequenced sample to a reference genome. Somatic mutations are particularly challenging because they usually involve comparing normal and tumoral samples from the same patient, and reads carrying variations are harder to align [39]. Current reference-based approaches tend to be very specialized and use different algorithms to target a particular kind of variant [36]. For instance, Mutect [32] and Platypus [107] are designed for single nucleotide variants (SNV), and Pindel [142], DELLY [125], and BreakDancer [28] are designed for structural variants (SV) with different characteristics. Hence, defining a

complete catalog of variation generally requires complex pipelines with combinations of multiple methods.

Emerging reference-free methods have the potential to provide more accurate results, but they also require significant computational and memory resources. Methods to detect single nucleotide polymorphisms (SNP) based on De Bruijn graphs [113] like Cortex [53], Bubbleparse [80], and NIKS [98] easily exceed the memory of a server with 512 GB of DRAM; with human datasets, processing a single chromosome can take as much as 105 GB [130], and memory for processing whole human genomes is expected to be much higher. Other methods like DiscoSnp use a cascade of Bloom filters to represent De Bruijn graphs [31, 33] and manage to keep a significantly lower memory footprint [130]. These variant calling methods are limited in scope and target a particular kind of non-somatic mutation.

SMUFIN, the user-case for this thesis, differs from the other methods because it is a comprehensive reference-free method that targets somatic mutations, and all kinds of variants, from SNVs to large SVs. Like many genomics applications, SMUFIN relies on 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 of 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 an example read `ACGGCAGCTG` has the following 8-mers: `ACGGCAGC`, `CGGCAGCT`, and `GGCAGCTG`. Counting the frequencies of k-mers is an important algorithm in bioinformatics that is used by many methods for different kinds of studies. The main applications for k-mer frequency counting are: sequence assembly, to reconstruct the original DNA sequence [133, 55]; sequence comparison, to extract knowledge of biodiversity between different environmental conditions [90, 11, 129, 99]; sequence clustering, to discover the feature of some sequence [143, 60, 19]; sequence alignment and variant calling [82, 65]; to predict secondary structures of RNAs [89, 30]; et cetera [139]. What makes SMUFIN k-mer frequency counting different from this method is that it operates on a couple of whole human genomes, normal and tumoral samples, of the same patient, making the memory footprint even higher in the order of few TBs.

2.4 THE SMUFIN METHOD

SMUFIN [95, 20] is a leading algorithm and implementation for detecting somatic mutations that detects both point and structural somatic mutations without full genome reconstruction or alignment against a reference. Since SMUFIN is *reference-free*, it avoids costly alignment, which often takes 70% of the genome analysis pipeline [137]. Additionally, by not aligning against a reference genome, SMUFIN can detect complex structural variations more effectively [95], and it could work on species where reference genomes may be incomplete or unavailable [79].

The end goal of SMUFIN is to reveal the exact mutations found in a tumorous sample, therefore allowing doctors to produce personalized medicine for those specific mutations. In its current state, SMUFIN is a research tool used by scientists, and as a result, it is desirable for SMUFIN to produce and keep more data to create a better picture of the potential mutations seen in the genome. SMUFIN has built-in parametrization to allow scientists to specify what k-mer combinations and reads are *interesting*, and this tuning can have a significant impact on the sizes of data in the later phases in SMUFIN. This work assumes settings that biologists have used for explor-

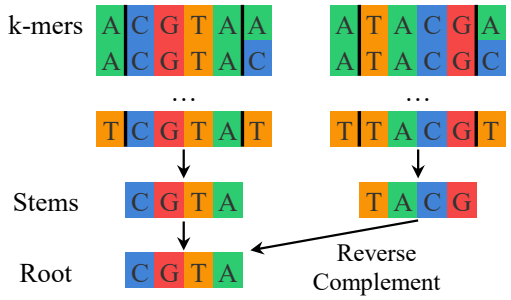


Figure 5: (§2.4.1) Hierarchy of k-mers, stems, and roots showing all the k-mers that map to the root `CGTA`.

atory research so far. In the future, different settings may be used for clinicians using SMUFIN as a tool to produce personalized medicine.

2.4.1 *SMUFIN Overview*

SMUFIN assumes no prior knowledge of the structure or the location of mutations. Instead, it uses k-mer counts for the normal and tumor sample to determine possible mutations. In the simplest case, a k-mer that is common in tumoral reads but is not seen in normal reads is expected to be part of a mutation. This is useful for detecting the presence of mutations, but it is not sufficient to align the normal and tumoral reads together to reconstruct the mutation.

To find k-mers that are useful for reconstructing mutations, one can look for k-mers that match in the middle k-2 bases, called the *stem* of a k-mer, but do not match in the first or last base. The simplest case is when of two k-mers with matching stem, one is only found in the normal sample, and the other only in the tumoral sample. In this case, these k-mers are likely to be the start or end of a mutation. Thus, they can be used to align DNA reads from both samples together to reveal the structure of the mutation.

Since DNA sequencing produces reads from both strands of the DNA, and mutations affect both strands equally, it is also useful to be able to refer to the frequency counts of the k-mers as they would appear in the other strand of the DNA, i.e., its reverse complement, to further improve the quality of the results. SMUFIN groups k-mers of the same stem with their reverse complement using their *root*, a canonicalized stem obtained by taking the first of the stem and its reverse complement in alphabetical order. The hierarchy of k-mers, stems, and roots can be seen in Figure 5. The software implementation of SMUFIN produces a histogram indexed by root, such that each time one searches for a k-mer in the histogram, it returns the counts for all the 64 k-mers with the same root, $(4 \cdot 4 \cdot 2 =)$ 32 for the normal data set and as many for the tumoral data set.

To decide what reads and k-mers are interesting for further mutation reconstruction and analysis, biologists define a set of criteria that reads and k-mers must meet. The set of criteria comprises characteristics of each single reads (e.g., contains a particular sequence of bases, quality markers of the bases in the reads) but also on the k-mers of a read (e.g., frequency counts of the k-mers and other k-mers from the same root). These criteria are used to construct the *interesting reads and k-mers database* which can be thought of as a set of two tables: one containing interesting reads, and the other containing interesting k-mers. Each read in the reads table contains a

Table 1: Number of different roots in a typical input data set with a breakdown of unique and different non-unique roots when using 30-mers. The number of different roots represent the number of entries in the histogram.

	Numbers in Billions	Percentage
Different roots	78	100 %
Roots of unique 30-mers	68	87.2 %
Roots of different non-unique 30-mers	10	12.8 %

bit vector denoting which k-mers in it are interesting, implicitly creating pointers to the k-mer table. Each k-mer in the k-mers table contains the IDs of the reads where this k-mer occurs, implicitly creating a pointer to the reads table. This database structure is efficient for constructing groups of reads containing a particular k-mer. Reads in such a group can be aligned for mutation reconstruction and further analysis.

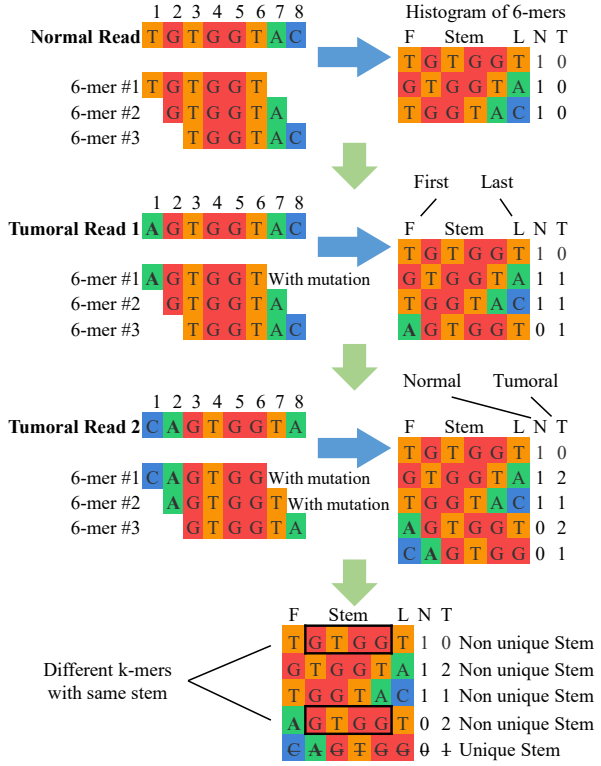
2.4.2 *SMUFIN Algorithm*

Conceptually, the SMUFIN algorithm is organized into three phases: k-mer counting, labeling, and grouping.

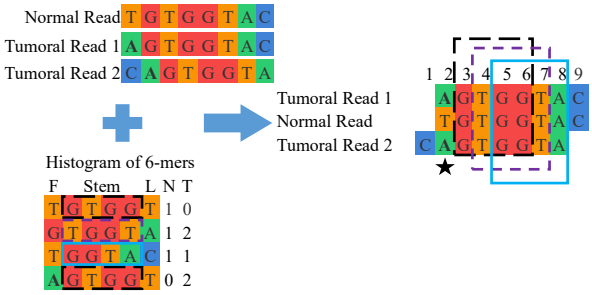
K-mer counting: The k-mer counting phase takes in sets of reads from two data sets: one normal and one tumoral. Depending on the sequencing machine, each read can be on the order of 80 to 150 bases long, and each base in the read has a corresponding quality score representing the confidence of the sequencing machine on the correctness of the base. In a typical sequenced genome sample, each part of the DNA appears in tens of different reads, resulting in a sample data set with billion of reads [49]. This input is around 310 GB of compressed FASTQ files and around 740 GB when uncompressed. These reads are then divided into k-mers. SMUFIN uses k values in the range of $24 < k < 32$. According to the domain experts involved in the original algorithm construction, this range of k-mers is unique enough to align to genomes accurately, and at the same time, general enough to accurately pinpoint mutations. For values outside this range results might become either too general (for $k <= 24$) or too selective (for $k >= 32$) producing results with poor sensitivity and specificity.

For each k-mer in each read, the corresponding entry in the histogram is incremented. Figure 6a shows a simplified example of k-mer counting. The full size of this histogram is about 78 billion roots, which would require almost 10 TB of memory. However, like Table1 reports, many of these roots can be regarded as sequencing errors [18] in the input data, and they only appear in a single read. To identify and remove these unique items, we use a list of all items seen in the input. The application throws out all the unique roots reducing the size of the histogram down to around 1 TB and the memory footprint of the application to around 2 TB. This histogram, once built, is not updated further, but is read 100s of billions of times by the labeling phase.

Labeling: The labeling phase constructs the *interesting reads and k-mers database*, described earlier in Section 2.4.1, making one additional pass through all the input data set. This step requires a histogram lookup of each k-mer of each input read to determine if the read and k-



(a) SMUFIN candidate break-point **detection** with 3 sample reads of length 8 and a histogram of 6-mers. In this example the stems GTGG and TGGT are potential point of divergence, and their neighboring bases are candidate break-point for a mutations.



(b) SMUFIN candidate break-point **reconstruction** where the "Tumoral read 1", containing a candidate stems, is aligned with all other reads containing at least one of the stems in "Tumoral read 1". In this example, the second base of the aligned block ($\star$) is a candidate break-point for a single point mutation.

Figure 6: (§2.4.2) Simplified example of SMUFIN candidate break-point detection and reconstruction. In this example, the candidate break-point is for a single point mutation, where a single nucleotide base is changed. In reality, a candidate break-point might be the beginning or end of a larger mutation, such as a structural mutation or a virus.

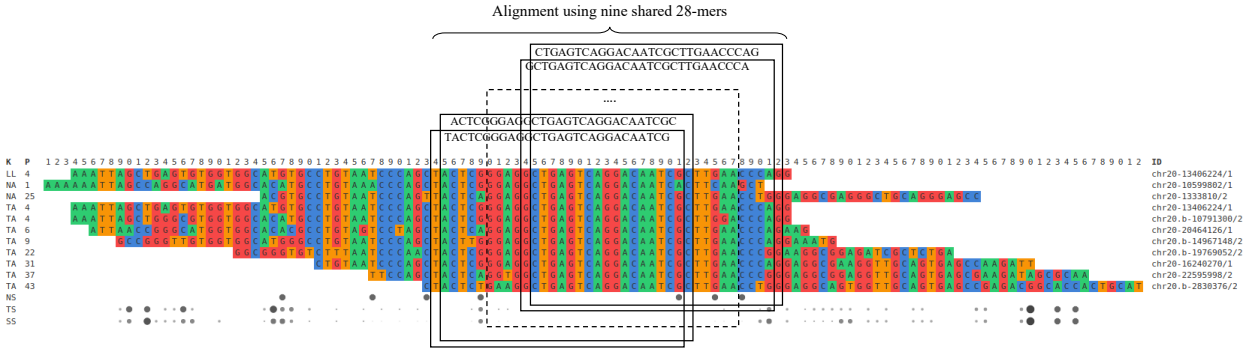


Figure 7: (§2.4.2) Example result of SMUFIN showing a candidate mutation.

mers are interesting. The root-indexed histogram offers data locality in accessing all the k-mer counters of the same root, which is necessary to determine if a read and k-mer are interesting for candidate break-point reconstruction. The database created in this step is implemented using RocksDB [123]. The read table in the database can be easily constructed sequentially by scanning through all the input reads, but the construction of the k-mer table requires random-access insertions. Depending on the set of criteria defined by the biologists, the size of the resulting database can vary significantly; for standard executions, it contains up to 150 million reads and 110 million k-mers, and its size is around 200 GB.

Grouping: The final phase of SMUFIN is grouping reads with the same interesting k-mers together to reconstruct the mutations. This is done by looking in the interesting reads and k-mers database and clustering reads with the same interesting k-mers. Once the groups of reads are assembled, they are aligned with each other according to where the matching k-mers were found. If the position of matching k-mers do not agree between normal and tumoral reads, then there may be a mutation, either insertion or deletion, causing the alignment to be off. Figure 6b shows a simplified example of grouping, and Figure 7 shows an example result for one candidate mutation.

3

ALGORITHMIC IMPROVEMENTS TO REDUCE WORKLOAD FOOTPRINT

This chapter describes the first contribution of this thesis. This contribution aims to reduce the footprint of workloads to target fewer nodes, if not just one. This contribution is motivated by the observation that monolithic data-intensive applications might have different requirements during the execution, and this might lead to wastes of resources, particularly when running on multiple nodes. The idea is to explore techniques to limit the waste of resources and to reduce the workload footprint; thus, lowering both capital and operational costs.

The main parts of this contribution are various. First, in Section 3.1 we present a modular software structure that is driven by the resource requirement to reduce the waste of resources and target one single machine. Second, in Section 3.2 we explore alternative and more memory-efficient data structures. Third, in Section 3.3 we suggest a manual swapping mechanism to non-volatile memory that allows extending the main memory of a system with NVM storage to reduce the main memory requirement. Next, in Section 3.4 we evaluate the different techniques and discuss results. Here we compare the legacy code of the application running on multiple nodes with a new version that adopts the solutions proposed in this contribution, and that can run in one customized single node with 512 GB of DRAM and an NVM drive used as memory extension.

3.1 REFACTORING MONOLITHIC SOFTWARE FOR MODULARITY

Read-world applications usually have different stages, each one with its own footprint and requirements. However, monolithic applications have to request enough resources to meet the requirement for the execution of all stages, wasting resources when executing less demanding stages. This waste of resources is particularly relevant for data-intensive applications with some stages characterized by memory footprint that exceeds the amount of memory in a compute node. In such cases, the usual way to meet the memory requirement is to scale-out the entire application to multiple nodes to obtain enough aggregated memory to meet the requirement of just one part of it. Besides, scaling to multiple nodes to meet the main memory requirement usually implies to over-allocate the compute resources, wasting even more CPU time and energy.

To minimize the waste of resources this thesis proposes to redesign the application using a modular structure made of multiple stages, called *units*. This structure allows the sizing of the workload footprint based on the resource demands of each distinct unit. To fit the execution into one single compute node with less memory than required this structure also admits breaking most demanding units into multiple *partitions*, to be executed sequentially one after another.

3.1.1 *Design Applications as a Sequence of Smaller Units*

To reduce the workload footprint, we propose a structure based on the resources demands of each algorithmic step of an application. In this structure, each step is considered a unit and the application consists of a sequence of units. When executing the application, each unit is executed sequentially, scaling to the number of nodes that better suit the unit itself.

Although simple, this structure offers other advantages. The structure fits well with scientific applications that run for hours because each change between two units is the natural moment to perform check-pointing. With check-points between units, this structure allows taking advantage of the heterogeneity offered by the computing facility of today. In fact, it is straightforward to execute a unit in the kind of node that better meets the requirement of that particular unit. For instance, if the performance of a unit scales well with the number of CPU threads, then that unit could be executed in nodes with high-end nodes. Instead, if another unit requires a considerable amount of main memory, then it should be run in "big memory" nodes that offer TB of DRAM at the cost of lower compute power. This structure allows reducing the workload footprint closer to its real resource demand, limiting the waste of resources, and lowering the operational costs.

For SMUFIN, we mapped the three phases of its algorithm, described in Section 2.4.2, to a structure with as many units:

Count: Unit that performs the k-mer frequency counting step. It reads the input genomes in a streaming fashion and builds the histogram in DRAM. Once the entire input data set is processed, it dumps to storage the *root histogram* containing all interesting, non-unique, roots. The work in this unit is parallelized using two different kinds of CPU threads: *Loader* and *Consumer*. The Loader threads are committed to (i) load the compressed input files from storage to memory, (ii) perform quality checks on the reads, (iii) generate and encode all the k-mers in their 64-bit form, and (iv) send the k-mers to the Consumer threads. The Consumer threads, instead, read incoming k-mers from the Loader threads and insert them in the list of all roots seen and, only if the root is already seen, it gets into the root histogram. Communication between Loader and Consumer threads happens via dedicated lock-free single-producer single-consumer queues. The Count unit has a DRAM footprint of approximately 2 TB to fit the root histogram and the list of all roots seen in DRAM. This substantial memory footprint dictates the workload footprint.

Label: Unit that performs the labeling step. It reads the input genomes in a streaming fashion once more and consults the root histogram, loaded in main memory, to identify interesting reads and k-mers for mutation reconstruction. This unit adds interesting items to the *interesting reads and k-mers database*. This unit has a high DRAM footprint (more than 1 TB), needed to store the histogram, and it benefits from CPU with many CPU threads.

Group: Unit that performs the grouping step. It reads the *interesting reads and k-mers database* in a streaming fashion to group reads with the same interesting k-mers that are stored into a human-readable output. The unit has a relatively low memory footprint. It needs no more than a hundred GB, but it can also run in machines with few tens GB at the cost of lower execution time.

Due to their high main memory requirements, Count and Label units are likely to run in multiple nodes or either in "big memory" nodes with plenty of main memory. Differently, the Group unit can now run in just one node, saving resources.

3.1.2 *Splitting Units into Partitions*

Like SMUFIN, it is reasonable to imagine applications with DRAM footprints that exceed the main memory of one single node, and that for this reason requires to scale-out to multiple nodes. Running on multiple and different nodes might not be a problem when running a workload on data centers and cloud computing services. Nevertheless, when targeting in-house computing, this has a direct impact on capital and operational costs.

To target one single compute node, we propose to split the most demanding units into multiple partitions, each for a disjoint part of the main data set domain. Each partition can then be executed as needed: sequentially in a single machine, concurrently in multiple nodes, or a combination of the two. Obviously, this partitioning depends from application to application, and it is not always possible; for instance, due to data dependencies between partitions. However, it matches well with genomics workloads. In fact, as described in Section 2.3, many genomics applications usually work with relatively small sub-strings, whose domain can easily be split using the first, for example, n bases to split the sub-strings domain, generates 4^n parts [1]. This allows splitting the domain easily. However, due to the natural distribution of DNA strings that is not proportional, and it is likely to lead to very unbalanced partitions where some partitions have many more items than others. For example, given a DNA sample, the k-mers that begin with `AAA` or `TTT` are many times more numerous than those starting with `CGA` or `CGC`.

For SMUFIN, we created balanced static partitions so that the memory requirement for each partition is similar. This required to collect frequency data for representative input data set for values of n up to 5. Then, we executed many bin packing problems with different numbers of bins. For each problem, the partitions were the bins, the five bases the items, and the frequency of the five bases were the weights. With this method, we were able to create balanced partitions schemes that are distributed with the binary. Furthermore, we decided to use the same partitioning scheme to divide each partition into many sub-partitions further to spread the work between the different Consumer threads. In this way, each thread can now work on dedicated data structures; thus, minimizing the need for synchronization between threads.

A partitioning, like the one we propose, that splits the data domain into disjoint partitions that can be executed independently has two minor drawbacks. First, every partition needs to read the entire input data set. This is not a problem when running each partition sequentially, but it might create a considerable amount of I/O between compute nodes and storage when executing the application with many partitions in parallel, each on a different node. Second, if two consecutive units are likely to be executed with a different number of partitions, then, the outputs generated in the first unit might need to be reorganized to the number of partitions used in the second unit; either merging it into fewer partitions or splitting it into more partitions.

For SMUFIN, where the Count and Label units have similar workload footprint but the Group unit instead is candidate to be executed with one partition, we added a new optional fourth unit, named Merge, in between the Label and the Group unit. Like its name suggests, the job of this new unit is to merge the output of the different partitions of the Label unit into one database. If the Label unit is executed with just one partition than, the Merge unit is not needed. Figure 8

1 4^n because each DNA base can assume four different values, one for each possible nucleobases – Adenine (A), Cytosine (C), Guanine (G), Thymine (T).

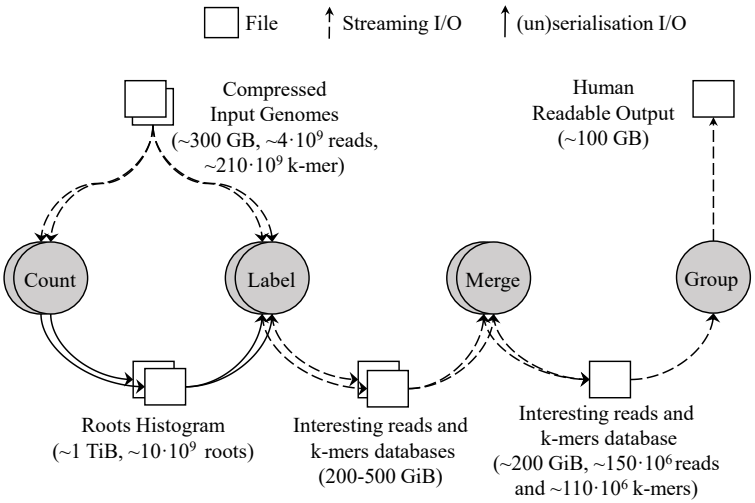


Figure 8: (§3.1) SMUFIN modular structure using partitions and units. In this example the application is executed using two partitions; thus, Count and Label units are executed twice reading the whole input genomes two times for each unit. The Merge unit takes care to fuse the output of the Label partitions into one so that the Group Unit needs to be executed only once.

shows an example of the modular structure of SMUFIN, using the four units – Count, Label, Merge, and Group.

Thanks to this modular structure, applications can be executed in just one single node using multiple sequential partitions. The drawback is that, compared to a scale-out solution to multiple nodes, the time-to-solution is likely to increase. However, thanks to the modularity of the proposed solution, the CPU time does not necessarily increase. Instead, since we minimized the waste of resources, it is expected to decrease.

For SMUFIN, the number of partitions required to meet the memory requirement of the Count and Label unit directly impacts the time-to-solution. Thus, solutions that relax this memory requirement could effectively reduce the execution time.

3.2 CHAINS OF BLOOM FILTERS TO IDENTIFY NON-UNIQUE K-MERS

Genomics applications that work on big data sets, like whole-human genomes, usually want to focus on parts of the DNA that are repeated many times in the DNA sample. The simplest way to do this is to build a list of all items seen. However, with billions of different items, this list can get as big as hundreds of GB. SMUFIN builds this table on the fly during the Count unit, and it keeps track of all items seen. This table is consulted per each item, but only those items already in the list are added to the roots histogram. With 87 billions of different items, the size of this histogram can be as big as 1 TB and is approximately half of the memory footprint of the Count unit.

As a more memory efficient alternative, we propose an approach based on Bloom filters. A Bloom filter is a space-efficient data structure that is used to tell if an item is in a given set or not. The only drawback of Bloom filters is that they accept false positives – items that are not in

the set, but that are thought to be. However, this probability of false positives can be reduced at the cost of a higher memory footprint, more hashing, and more memory accesses. To distinguish from unique and non-unique items, we propose a chain of two Bloom filters. In the chain, all items are added to the first Bloom filters, and they get added to the second filter only if they are already present in the first. At the end of the process, the second filter can tell which items are seen more than once and which are not. In this case, the false positives are those items though to be seen multiple times, but that they are seen only once. For whole-human genome analyses with billions of items, this structure requires only tens of GB and offers a much more memory efficient alternative to a list with all items, that instead requires hundreds of GB.

For SMUFIN, where there can be as many as 78 billion different items and 68 billion unique items (see Table 1), this Chain of Bloom filter is considerably smaller than the list of all items. For instance, with a false positive of 1%, the size of the filters is approximately 84 GB and 11 GB for the first and the second level, meaning that the entire input can be processed in just one partition. Besides, once fully built, the first filter can be discarded, and only the second and smaller filter needs to be used in the following Count unit. We integrated this method by adding an additional unit called Prune. This unit is executed before the Count unit. Its goal is to add all roots to the chain of Bloom filter so that the second level can, then, be used in the Count unit to drop unique items. This is an optional unit that requires to read the entire input data set one extra time, but that helps to lower the DRAM requirement of the next Count unit, thus lowering the number of partitions needed.

3.3 MANUAL SWAPPING TO NON-VOLATILE MEMORY

Data-intensive applications that build vast data structure resident in main memory need to run either on multiple nodes or in one single node, using a partitioning scheme like the one presented in Section 3.1. However, even if the second option is possible, the main memory requirement might lead to many partitions, which, in turn, increase the time-to-solution. One obvious way to reduce DRAM footprint of data-intensive applications is to store the data structure on storage, for example using a key-value store, rather than in main memory. This solution comes with a clear performance penalty, but new NVM drives with few GB/s of bandwidth might offer a valid comprise[2].

To explore and measure the performance penalty of using NVM as an alternative to main memory, we ran a benchmark on a key-value store to emulate a load of a generic k-mer counting algorithm. The benchmark ran a load of randomized 100% percent updates. For the experiments, we used a Fusion ioMemory SX350-3200 drive, formatted using ext3, and RocksDB. Even if the drive has a nominal read and write bandwidth of 2.2 and 2.7 GB/s, results showed a drastic degradation of the performance with a maximum key-value store bandwidth of 80 MB/s. This exploratory result highlighted how the key-value store was unable to take advantage of the parallelism offered by NVM drives. Besides, it became clear that the random nature of the updates to the root histogram is not a good fit for NVM drives. Particularly, if we consider that each of the hundreds of billions of updates to the histogram will trigger a write of an entire 4-KB flash

2 This work dates back to 2015. At the time, NVMe was not so popular as it is today, and our work focused on general NVM drives, not necessarily NVMe. However, most of this contribution still applies to NVMe drives of today.

page, degrading performance, and wearing out the flash drive. In fact, with so many updates, the execution would write hundreds of PB per each patient. Owing to these reasons, we discarded the option to use NVM drives to build the histogram using a general-purpose key-value store. We explore and discuss the idea of creating the histogram on NVM storage in Section 5. Here we use a different approach to build and consult the data. Besides, we also offer results in an evaluation of more recent RocksDB version using more recent NVMe drives.

To overcome the poor performance of a key-value store, we implemented a custom swapping mechanism to flush to NVM data structures that gets too big to fit in memory. This custom swapping mechanism substitutes the OS swapping, which is disabled to prevent performance deterioration due to kernel-jittering. To avoid the overhead of a file system software stack and to have a byte-addressable memory, we access the NVM drive as a block device which gets memory mapped, using `mmap(2)`, and addressed as normal memory. This requires to have an NVM drive fully dedicate to the application. When this is not possible, the mapping target for the `mmap(2)` call can use a normal file. Additionally, we also developed a ledger system to store all metadata required to identify and retrieve each swapped data structure. Such data comprise, but is not limited to, information about: the partitions to which a table belongs, if a table is *stale* and can be removed or not, the offset in bytes from the beginning of the NVM drive, and the size of the data structure. Metadata is kept in DRAM, and it is written to a reserved area at the beginning of the memory extension right before the end of the executing. To hide swapping latency and overlap I/O with computation, the mechanism uses multiple threads to enable an asynchronous swapping. Once a data structure gets too big, it is passed to the main thread that keeps the ledger of the metadata. This main thread updates the ledger and spawns a new thread that performs a user-defined function to perform the swapping of the data itself, at the offset determined by the main thread, and dies. The user-defined function allows using this mechanism with any data structure. While the swapping thread is at work writing to storage, the calling thread can keep working using either a new empty data structure or a previously allocated one that works as *hot spare*. Hot spare data structures are useful when the initialization of the table is a costly operation. Besides, this mechanism adopts some of the flash-specific optimizations discusses in Section 2.2. In detail, it enforces writes to be aligned to flash pages, and it buffers the writes so to write multiple flash pages at once and take advantage of the internal parallelism of flash drives [27]. Finally, the swapping mechanism can be tuned to use only a few working threads to reduce the number of context switches when the CPU is already saturated. Similarly, the mechanism also admits a forced swapping where it spawns as many worker threads as needed. This is useful at the end of the execution when the application needs to dump everything as fast as possible, and there is no other load on the CPU.

For SMUFIN, we easily integrated the swapping with a user-defined function that iterates on every item of Google's Hash tables used to build the histogram. Besides, we added yet another unit called Unify. This unit that is executed after the Count unit to unify the swapped tables and convert their format into the format required by the next Label unit. This extra Unify unit allows us to adopt different data layouts from Count to Label unit to address the needs of both units. In detail, in the original implementation, the data layout was chosen to provide maximum data locality in the Label unit, where for each k-mer in the input samples, all counts of all k-mers belonging to the same root must be considered. For this reason, counts are grouped per

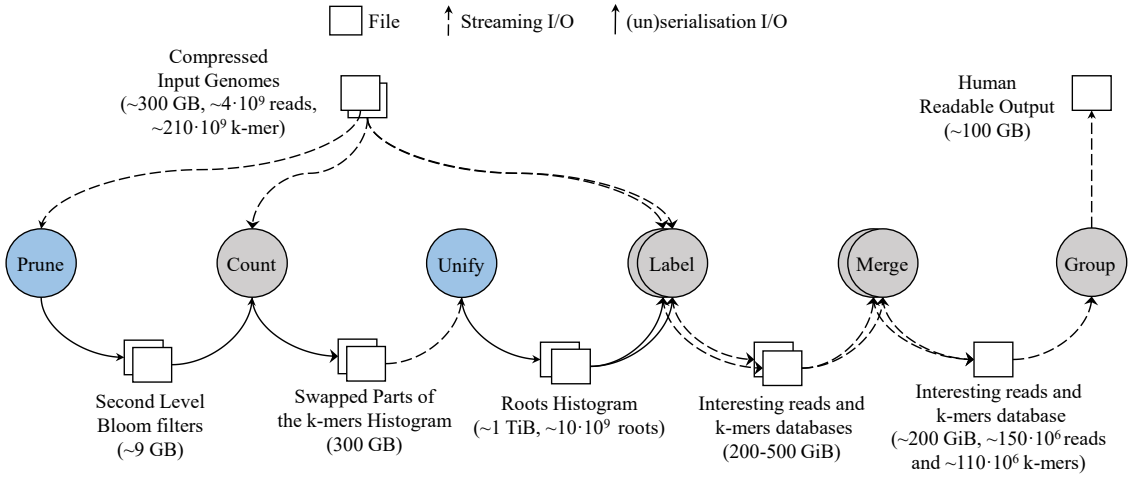


Figure 9: (§3.2-3.3) SMUFIN extended structure using Prune and Unify units to lower the number of partitions required to execute the k-mer counting algorithm. In this example the application is executed using two partitions only for the Label and Merge units.

root, creating items of 136 Bytes – 8 for the root and 2 for each of the 64 counters. While this format is good for the performance of the Label unit, it is an inefficient format that contains many counters with a value equal to zero. Thus, the root format wastes precious space during the Count unit. A better layout for the Count unit is to store counts per k-mer. Using the k-mer format each entry of the histogram is 16 Bytes – 8 for the k-mer, 4 for the two counts, and 4 of padding added by the *std::pair<key,item>* used internally by the implementation of Google's Sparse hash table. A layout that, given the many zeros in the root form, reduces the memory footprint of the overall tables. Besides, when swapping a table, we drop the extra padding added by the *std::pair* preventing to write hundreds of GB of useless data on the memory extension. Finally, now this swapping and the Prune unit allow us to target only one partition for the Count unit; it is worth considering faster Google's Dense hash table to build the root histogram. The Dense implementation is known to be considerably faster than the Sparse, but it comes with a higher memory footprint, which, for SMUFIN, translates to more tables swapped. We explore and evaluate this option in Section 3.4.

The work discussed in Sections 3.2 and 3.3 transformed the k-mer counting algorithm of SMUFIN to reduce its memory footprint to target just one partition. This is achieved by doing a preliminary full pass on the entire input data (Prune unit) and some extra work to transform and merge the swapped tables (Unify unit). Figure 9 shows the new structure of SMUFIN with its new units summarized below:

Prune: Unit that adds all k-mers in the input files to a chain of two Bloom filters where only those already in the first (i.e., seen before), plus false positives, are propagated to the second one. At processing all the input, the second Bloom filter effectively constitutes a structure that can tell whether a k-mer has been observed more than once. This second filter is used in the Count unit to discard unique k-mers that are not relevant for the algorithm, reducing memory footprint and execution time.

Table 2: (§3.4) Experimental Setup

	`MareNostrum 3`	`FatNode`
CPU	Intel SandyBridge-EP 2x 8-core E5-2670 @2.6GHz	Intel Xeon 2x 12-core E5-2680v3 @ 2.50GHz
#CPU Threads	2x 16	2x 24
Main Memory	128 GB 8x 16-GB DDR3-1600 DIMMs	512 GB 16x 32-GB DDR4-2133 DIMMs
Storage	1.9 PB of GPFS disk storage	1x 3.2 TB PCIe NVM FusionIO SX350-3200 (2.7 GB/s read, 2.2 GB/s write), plus rotational disks for the OS
Storage as Memory Extension	N/A	1x 1.6 TB PCIe NVM FusionIO SX350-1600 (2.7 GB/s read, 1.7 GB/s write)
Operating System	SuSe 11 SP3	Ubuntu 16.10 4.4.0-72-generic kernel

Count: This new version uses the Bloom filter created in the Prune unit to drop unique items without the need to build a list of all roots seen. This new version also uses a different more compact memory layout to keep a lower memory footprint. Finally, the custom swapping mechanism stores full tables to an NVM drive so as not to run out of memory.

Unify: Unit that combines swapped frequency tables in the new version of the Count unit and changes the memory layout to its expected form required by the following Label unit. False positives given by the Bloom filter are also removed at this point.

The rest of the application was left unaltered and the proposed changes are entirely compatible with the original implementation of the other units.

3.4 EVALUATION AND RESULTS

In order to evaluate the impact of the presented work, we ran SMUFIN in two different environments. First, we use `MareNostrum 3`, a supercomputer with many distributed nodes and a high-speed interconnect where SMUFIN is usually deployed in production. Second, a single customized node, called `FatNode`, with 512 GB of main memory and NVM drives to evaluate the vertical scalability. Table 2 reports the specification of both these nodes. We evaluate the aggregate node time, time-to-solution, and energy-to-solution of the entire SMUFIN application running in each node. Besides, we compare the results with those of the legacy code also running in the `MareNostrum 3`. We conclude with a characterization of the new implementation of the k-mer counting algorithm running in `FatNode` to understand its bottleneck and limitations.

3.4.1 *Evaluation Methodology*

To evaluate the work presented, we first compare the legacy and the new SMUFIN, both running in 16 nodes of `MareNostrum 3`. Here, each of the 16 nodes takes care of one different data partition. Then, we compare the proposed scale-up version on `FatNode` using both Sparse and Dense hash table implementation. Here, thanks to the Prune and the manual swapping we proposed in Section 3.3, data partitioning is not required for counting k-mers – Count unit. However, the roots histogram still occupies around 600 GB and does not fit in DRAM entirely. To overcome this, we take advantage of the partitioning scheme, and we use two partitions only for the Label unit. When more than partitions are used, to compare the node times, we aggregate the execution time of each partition into the aggregate node time metric.

In each machine, the number of CPU threads used by the application changes accordingly so to match the hyper-thread threads of each machine. That is, in each node of `MareNostrum 3` we used 8 loaders and 8 Consumers, whereas in the `FatNode`, we increased the number of Consumer threads to 48. In most of the cases, we collected power and energy consumption collected using IPMI (Intelligent Platform Management Interface), and, for `FatNode`, we confirmed with the read of the Intelligent Power Distribution Unit (iPDU) used. For the legacy code on `MareNostrum 3`, we instead extrapolated the energy-to-solution using the mean power consumption of the nodes when executed the newer version of SMUFIN.

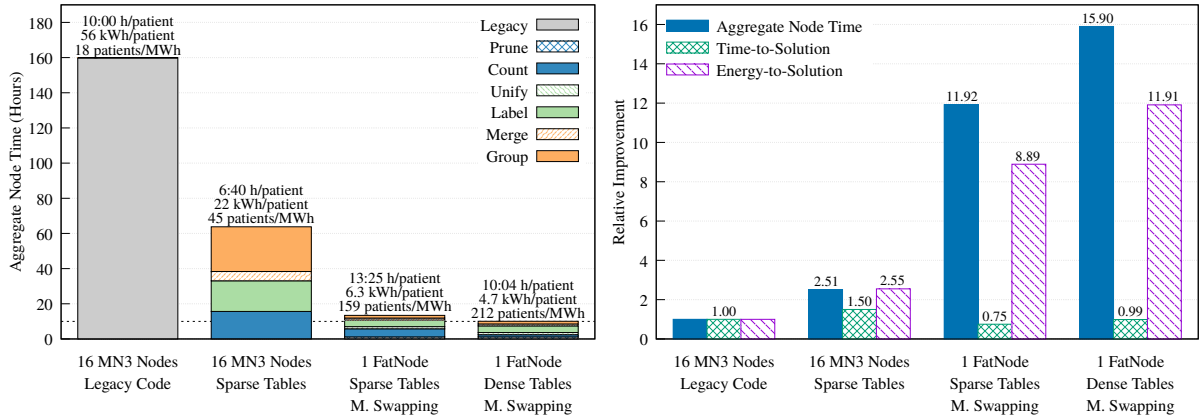
Before every execution, the OS caches were cleared, and to avoid any performance penalty due to OS noise, we disabled the OS swapping. This is particularly important on `FatNode` to avoid any conflict with the manual swapping.

In each experiment, we processed the same personalized genome based on the Hg19 reference, with randomly chosen germline and somatic variants as described in [95], including SNPs, SNVs (more than 100 bp apart), translocations, and random insertions, deletions and inversions, all ranging from 1 to 100Mbp. In silico sequencing was simulated using ART Illumina21. The total size of both normal and tumoral samples is 312GB of gzip-compressed FASTQ files, approximately 740 once uncompressed. We use this same input data set for all the experiments of this thesis.

3.4.2 *Performance and Energy Consumption*

Figure 10a shows the aggregate node time, time-to-solution and energy-to-solution for all four configurations. Figure 10b offers instead a view on the relative improvement normalized to the legacy code on the 16 `MareNostrum 3` nodes. The figures show how, even when running on the same nodes, the new code outperforms the legacy monolithic application under all the metrics used. To note how the aggregated node time and energy-to-solution are both reduced of 2.5x while the time-to-solution of an inferior 1.5x. This result is due to the fact that the Group unit is executed only in one node rather than keeping all 16 nodes busy, further reducing node time and energy-to-solution.

Figures 10a also show how all three main units – Count, Label, and Group – have significantly better performance on our single customized node compared to the distributed environment. Such difference is not only due to the diverse CPU generations between systems, but it is mainly due to the reduced number of data partitions and to the use of local NVM, which helps I/O



(a) Absolute results

(b) Relative improvement normalized to the legacy code

Figure 10: (§3.4.2) Absolute (left) and Relative (right) aggregate node time, time-to-solution and energy-to-solution of SMUFIN running in 16 `MareNostrum 3` nodes and in 1 Xeon-based node with NVM drives.

bound units like Merge and Group. The results show that even though both scale-up configurations have higher time-to-solution than the 16 `MareNostrum 4` nodes, they are clearly better in terms of node time and energy-to-solution. The difference between the Count unit in the configuration with Sparse and Dense tables shows how this second implementation outperforms the former. However, Dense tables come with a higher memory footprint, and they can be used only in the `FatNode` together with the manual swapping mechanism. In fact, on `MareNostrum 3` the dense implementation would lead to even more partitions, thus more nodes and more pressure on the I/O – each partition is reading the whole input data set – that not necessarily means lower node time.

Finally, the dashed horizontal line in Figure 10a marks the time-to-solution of the legacy code using 16 `MareNostrum 3` nodes. This line shows how the proposed solution on `FatNode` with dense tables and manual swapping matches the performance of the legacy code while consuming 15.90x less node time and 11.91x less energy.

3.4.2.1 *Targeting Even Less Main Memory*

Thanks to the work of this contribution, we can run the k-mer counting in one single node with 512 GB of DRAM and with an NVM drive where it writes around 312 GB of data. However, since the root histogram is still too big for the Label unit, we had to split this unit into multiple partitions executed sequentially at the cost of higher execution time. To understand the performance of hypothetical systems like `FatNode` but with 256 and 128 GB of DRAM, we accordingly increased the number of partitions for the Label unit and instructed the application to start swapping not to exceed the memory quota. With 256 GB of DRAM, the application took approximately 22 hours swapping 534 GB of data during the Count unit and using 4 partitions for the Label unit. Instead, with 128 GB of DRAM the application took approximately 40 hours swapping 591 GB of data and using 8 partitions for the Label unit. During these executions, the Count unit maintained a similar execution time than when targeting 512 GB of memory, proving that the

manual swapping mechanism can sustain more load. Differently, since we used more partitions than before, the Label unit is the unit that was most penalized among all units, and it dominated the time-to-solution.

3.4.3 *Characterization of the K-mer Counting Algorithm*

To understand the bottleneck and limitations of the K-mer Counting Algorithm, we characterized its behavior using profiling tools to collect traces of the entire system. This algorithm is of particular interest because it is a genomics algorithm also used in other methods. Figure 11 shows the collected traces.

The trace relative to the CPU usage shows a steady 100% utilization of the CPU for both Prune and Count units. The only time when the CPU usage lowers is when the Manual Swapping mechanism starts to operate in the middle and end of the Count unit. At these points in time, the application is swapping k-mer tables to the NVM resulting in the I/O bursts visible in the storage traffic plot. Although for the first group of I/O bursts, there is a feeble reduction in the CPU usage. For the second group, instead, the CPU usage drops drastically with simultaneous peaks of Wait time. This difference between the two groups of burst lies in the way the manual swapping mechanism works when called passively rather than forcedly. The first group of bursts is during the execution, where the manual swapping mechanism works on the background, trying to minimize its noise on the rest of the application. The second group of bursts, instead, comes at the end of the Count unit, where there is nothing more to process; thus, we do not mind to flood the NVM with request and all the 48 consumer threads used require to swap in a forced manner. The lower CPU usage in the Unify unit is mainly due to the I/O and memory-intensive nature of this unit. Because of it, in this unit, we only use a dozen CPU threads that are already able to put enough pressure on the I/O subsystem. Besides, empirical tests showed that a higher number of CPU threads is counterproductive and rapidly exacerbates the CPU wait time. The traces also show the ability of the NVM drives to absorb short bursts of I/O at a higher bandwidth than their nominal write bandwidth. This happens on multiple occasions during the execution, and it is achieved thanks to the on-board DRAM that these drives have. Finally, the traces suggest that both Prune and Count units are limited by the CPU, and their execution time could be reduced with CPUs with higher thread count or by offloading some of the computation to accelerators.

3.5 RELATED WORK

3.5.1 *K-mer Counting Using Storage*

Jellyfish [83] and KAnalyze [9] write sorted subsets of data to disk, and subsequently accumulate counts from each subset. DSK [108], KMC [40], and MSPKmerCounter [91] split the k-mer space into partitions, or disk buckets, so that each partition can be loaded into memory and processed individually. These general algorithms for k-mer counting process a single sample and do not optimize their output for random access. SMUFIN compares and performs operations consider-

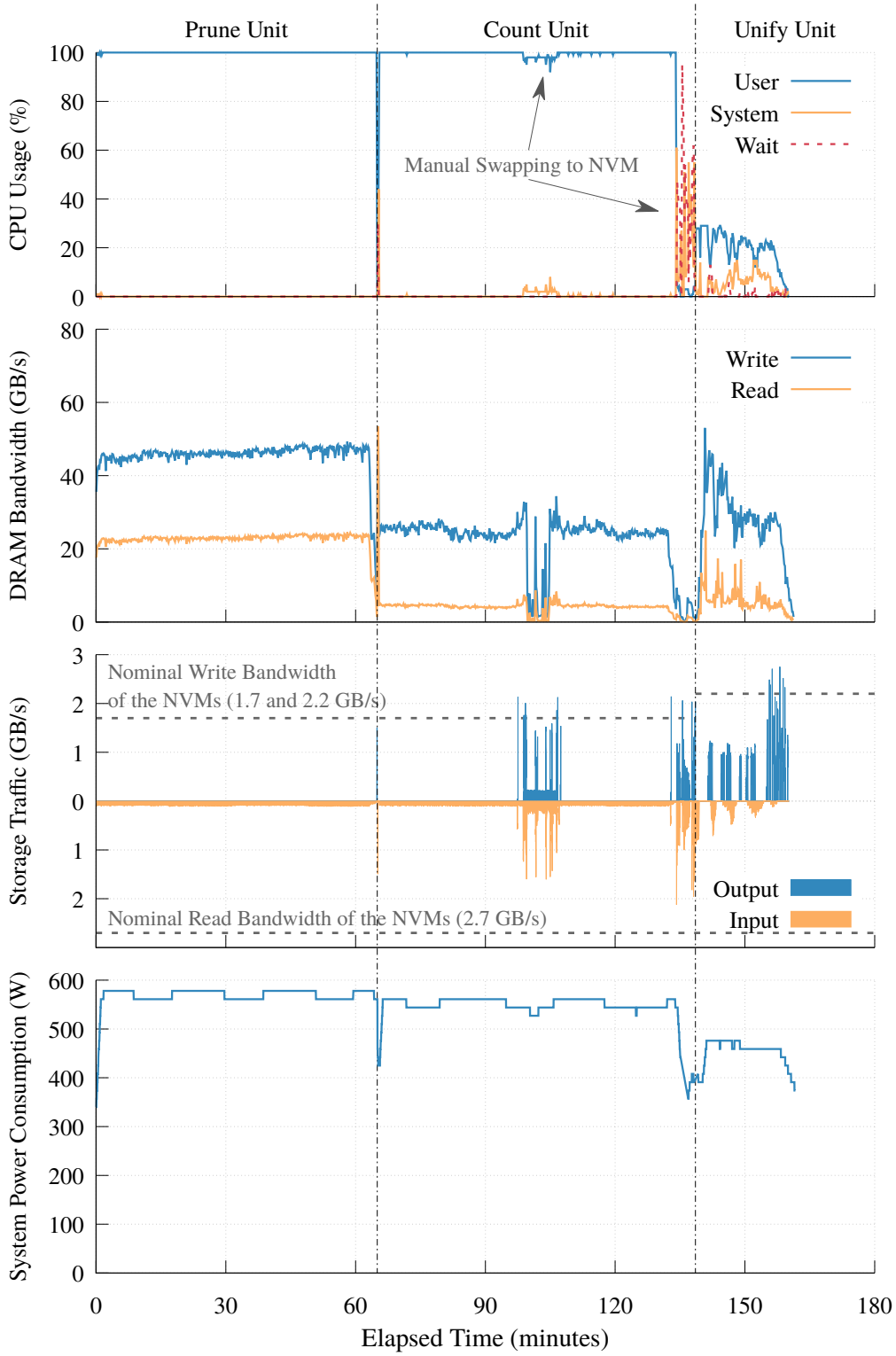


Figure 11: (§3.4.3) System traces of SMUFIN k-mer Counting algorithm – Prune, Count, and Unify unit.

ing k-mer counters of normal and tumoral samples of the same patient, potentially making the memory footprint even higher. Besides, our implementation was designed to take advantage of the internal parallelism of NVM while most, if not all, other methods were developed and used on rotational disks.

Like other implementations, Google's hash map implementations allow the serialization of a table to a stream, which we could have used to swap and load tables back to memory. However, while this constitutes a neat way, it has two issues that increase the amount of data required to store a serialized table. First, its implementation also stores metadata to ensure that once a table is loaded back to memory, it will have each item at the same position of the same bucket as in the original table, which is something we do not need. The second issue instead, is that it copies each *std::pair<key,item>* as it is, including eventual padding used for data alignment within the *std::pair*. If for small volumes, these two issues might be irrelevant. For vast data structures like the tables of k-mers using in the Count unit of SMUFIN, this extra padding is 25% of the raw data footprint that amounts to more than 100 GB per patient, and it should be prevented.

3.5.2 *K-mer Counting Using Bloom filters*

Counting k-mer frequencies is a widely studied problem in the literature, and different kinds of data structures have been used to solve it [71]. Similar to SMUFIN, BFCounter [93] uses a combination of hash tables and Bloom filters. In [86], instead, Counting Bloom filters are used to remove k-mers seen few times. However, this kind of filter has a memory footprint that is 3 to 4 times higher than the normal filter; thus, it is not a good fit for the SMUFIN method. Moreover, some of such works also include specific methods to discard the least frequent k-mers similar to the chain of Bloom filter we implemented for SMUFIN.

Differently from SMUFIN, most k-mer counters focus on either short k-mers or smaller input sizes, which reduces the size of the Bloom filter drastically. For instance, both [70, 88] use Bloom filters smaller than 1 MB while our implementation uses much bigger Bloom filters (around 90 GB) to deal with two human DNA samples.

3.6 FINAL CONSIDERATIONS

In this chapter, we explored an initial hardware-software co-design to reduce the workload footprint of data-intensive applications to target fewer compute nodes, if not just one. We presented a modular structure that splits the application into different units. This structure can be used to prevent waste of resources and that divide the data domain into partitions to target just one node where partitions can be executed sequentially. Besides, we explored how a Chain of Bloom filters and a manual swapping technique to NVMes allow reducing the main memory requirement to lower the number of partitions that impacts the time-to-solution.

Results showed that we were able to reduce the workload footprint of SMUFIN to the point that it can be executed in just one node with 512 GB of main memory and one non-volatile memory (used as memory extension) rather than the usual from 16 nodes, each with 128 GB of main memory, while maintaining a similar time-to-solution. This allowed reducing the aggregate

node time and the energy-to-solution of 15.9x and 11.91x, respectively. A considerable part of the algorithm proved to be CPU bound, suggesting that accelerators could be leveraged to offload some of the compute-intensive work.

4

OFFLOADING COMPUTATION TO ACCELERATORS

This chapter describes the second contribution of this thesis. This contribution explores the offloading of computation to programmable accelerators (i.e., GPUs and FPGAs) to reduce the time- and energy-to-solution. This contribution is motivated by the observation that data-intensive applications that process large volumes of data can easily become CPU bound. When this is the case, accelerators could be a more energy-efficient alternative. In the case of SMUFIN, system traces in Section 3.4 suggested that the CPU is fully utilized for a considerable part of the k-mer counting algorithm, algorithm that could also be used in other genomics methods. Owing to these reasons, in this contribution, we explore how this algorithm can be accelerated. Thanks to the modular structure given to the application in the first contribution, the study is isolated to only a few units of the SMUFIN pipeline.

The topics of this second contribution are many. First, we focus on offloading CPU intensive operations to GPUs. Then, to overcome the memory constraints of accelerators, we propose and prototype a method to virtualize accelerators' memory in host DRAM. Next, we port the GPU code to FPGAs. Here, we delve into portability issues that arise when porting the code from one kind of accelerator to another. Finally, we discuss results with a thorough analysis of bottlenecks and limitations. In this scope, we discuss power and energy consumption with and without accelerators, and we show how single components of accelerator (i.e., off-chip memory and PCIe connection) can hinder performance, particularly for data-intensive applications. To be able to target both GPUs and FPGAs, we implemented the code using OpenCL. Given the long compilation times for FPGAs, we started prototyping for GPUs, and we ported the code to FPGAs in a second moment.

4.1 IDENTIFYING CANDIDATES FOR ACCELERATOR OFFLOADING

This section presents a description of how the k-mer counting algorithm is adapted to offload some computation to GPUs. First, we describe how DNA data encoding can be offloaded and how to reduce inter-thread communication between CPU threads. Next, we present how to shuffle data on the GPU to minimize data movement on the CPU side. Finally, we outline how CPU and GPU can cooperate to build the chain of Bloom filters.

4.1.1 *Offloading CPU Intensive Operations and Minimizing Inter-Thread Communications*

Like described in Section 3.1.1, the work in the Count and Prune unit is parallelized using two different kinds of CPU threads: *Loader* and *Consumer*. While the Loader threads read and encode all k-mers in the input data set, the Consumer threads, instead, read incoming k-mers from the

Loader threads and insert them in the chain of Bloom filters in the Prune unit, or the roots histogram in the Count unit. This design allows taking advantage of CPU with many cores. However, its vertical scalability, to even more CPU threads, might be limited by the communication between the two kinds of threads. In fact, each Loader thread has a dedicated lock-free single-producer single-consumer queue with each of the Consumer threads. This means that with 8 Loaders and 48 Consumers, like in a node similar to `FatNode`, there are $8 \cdot 48 = 384$ queues that generate plenty of traffic. By offloading to accelerators, we also want to remove the need for these queues to minimize inter-thread communication.

We start with a naive offloading of the quality checks of the input DNA reads and encoding of all the k-mers to their 64-bit representation. These operations are a good fit for accelerators like GPUs and FPGAs because of they are compute-intensive bit-wise operations that expose the possibility of SIMD parallelism. Moreover, because each DNA read can be processed in parallel, there is no data dependency between the data. To efficiently processes the whole data set, that once uncompressed is around 600 GB, into smaller chunks. The size of each chunk involved depends on the available GPUs and must be carefully chosen to fully exploit the high degree of parallelism offered by high-end GPUs. In particular, because each GPU thread (work-item) processes one distinct DNA read, the number of reads in one input chunk has to be enough to keep the processing elements of the accelerator busy. Thus, the degree of parallelism that the accelerator exhibits dictates the size of each chunk. To keep the thousands of threads in modern GPUs busy, the input chunks size must in the order of hundreds of MB.

To overlap the communication between and computation on CPU and GPU, a double buffering pipeline is used to stream data chunks to and from the accelerator. This pipeline comprises of five stages that after a ramp-up phase concurrently work on different data chunks at every cycle and where the output of one stage is the input of the following stage in the next cycle. The stages can be summarized as: (i) CPU Loader threads fill a host-side input chunk with DNA reads and quality markers; (ii) transfer an input chunk from host to accelerator memory; (iii) the accelerator consumes an input chunk generating all k-mers in an output chunk; (iv) transfer an output chunk from the accelerator to host memory; and (v) CPU Consumer threads process the k-mers directly from the output chunk. The synchronization of the entire double buffering pipeline is taken care of by a new kind of CPU thread called *Orchestrator* thread. Figure 12 offers a representation of the pipeline. To have better control of data transfers and allow simultaneous data transfers, one in each direction, the code takes advantage of pinned memory buffers. However, simultaneous data transfers are not always possible. In fact, only accelerators equipped with two, or more, copy engines provide this option. When this is not possible, the code is not going to change, but the driver of the accelerator will force sequential transfers.

During this work, we also assessed the possibility to compute the hashes of each k-mer in the GPU. Hashing multiple items in parallel is an ideal work for a GPU, and it would be used by the Consumer threads to add the items in the Bloom filters and in the Hash tables used to build the histogram. This path was discarded because we realized that output buffers from the GPU of twice the current size would increase the PCIe transfer time, becoming the main bottleneck.

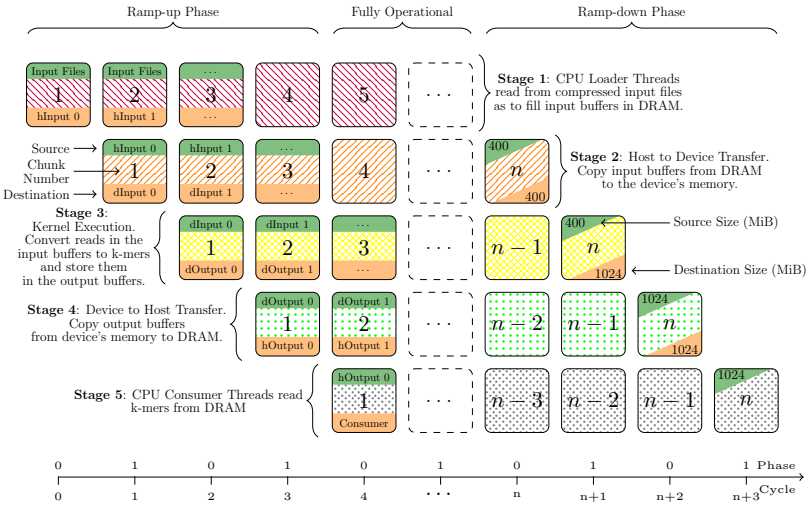


Figure 12: (§4.1) Representation of the double buffering pipeline used to offload computation to accelerators. The pipeline allows overlapping simultaneous PCIe transfers, in both directions, and computation, on the host CPU and on the accelerator.

4.1.2 *Shuffling Data on GPU to Minimize Data Movement on CPU*

With the pipeline described in the previous section, one of the challenges is to minimize data movement. Given a single output chunk coming from the GPU, In fact, with the naive offloading each CPU Consumer threads needs to read the whole output chunk from the GPU, even if only a subset of the items in the chunk belongs to each Consumer. This approach is simpler and removes the inter-thread communication. However, it increases data movement, which also constrains the scalability of the application.

To minimize data movement and to address this limitation, we designed an algorithm that shuffles k-mers according to the Consumer thread they belong to. In this approach, the output chunk of the GPU is split into disjoint sub-chunks, one per each Consumer thread. The sizes and offset of each sub-chunks are now added to the output of the GPU. These values are used by each Consumer thread to know where its sub-chunk starts and ends, effectively minimizing data movement, on the CPU side, by the number of Consumer threads. For instance, with a typical output buffer of one GB and 48 Consumer threads, like in the `FatNode` used in Section 3.4, this shuffling reduces the data movement from 48 GB to just 1 GB.

To shuffle the k-mers, we implemented an algorithm that resembles a parallel Counting Sort, but that does not sort the values within the buckets. This shuffling kernel uses the data partitioning logic of SMUFIN to know to which CPU Consumer thread each k-mer belongs. The algorithm for shuffling was blended with the naive kernel that generates the k-mers, and the resulting algorithm is composed of the following four kernels, whose times are summarized in Table 3:

- *Zero-out kernel*: Resets all device-side data shared among different kernels from the previous cycle of the pipeline.

Table 3: (§4.1.2) GPU Kernels times on a TESLA K40c.

Kernel	Average Time (µs)	Percentage Deviation (%)	Weight (%)
Zero-out	29.56	70.19	0.006
Encode	420,064.06	1.10	91.936
Prefix-sum	11.06	7.89	0.002
Shuffle	36,806.56	1.10	8.056
Sum	456,911.22	-	100

- *Encode kernel*: Each GPU thread (work-item) generates all k-mers from a distinct DNA input read and count how many k-mers belong to each Consumer thread creating a histogram with as many bins as many Consumer threads. GPU threads cooperate (using atomic add operation) to build, first, a local histogram (at work-group scope) and then a global histogram stored in an intermediate GPU-side buffer. Algorithm 1 describes this kernel in more detail.
- *Prefix-sum kernel*: Performs an exclusive prefix-sum on the global histogram. Note that the result of the prefix-sum is the list of offsets that tells each Consumer thread how many k-mers and where to start reading them in the output chunk.
- *Shuffle kernel*[3]: Per each Consumer thread, each work-group copies the, already locally shuffled, k-mers to the output buffer applying the offsets obtained from the previous prefix-sum, shuffling all k-mers by the Consumer threads. Algorithm 2 describes this kernel in more detail.

3 In the original publication [20], this *Shuffle* kernel was called *Broker* kernel. The logic was left unchanged.

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 for each CPU thread),
wg_offsets_mid (Work-groups offsets in the middle buffer),
wg_offsets_out (Work-groups offsets in the output buffer).

```
1: procedure ENCODE
2:     // Cohesively read all the input for this work-group to local memory
3:     l_in = cohesive_read_input (in, get_group_id(0), get_local_id(0))
4:     barrier(CLK_LOCAL_MEM_FENCE)
5:
6:     // Generate all k-mers and build a private histogram
7:     for i = 0, KMERS_PER_READ do
8:         p_kmers[i] = generate_kmer(l_in, get_local_id(0), i)
9:         if p_kmers[i] == INVALID then continue
10:        end if
11:        p_histo[get_cpu_consumer_thread_id(p_kmers[i])]++
12:    end for
13:
14:    // Merge private histograms into a local histogram and store
15:    // the result to be used as offset by this thread to shuffle k-mers locally
16:    for i = 0, NUM_CPU_CONSUMER_THREADS do
17:        p_offsets[i] = atomic_add(l_histo[i], p_histo[i])
18:    end for
19:    barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE)
20:
21:    if get_local_id(0) == 0 then
22:        // Merge local histograms into a global histogram and store
23:        // the result to be used as offset by this work-group in the mid buffer
24:        for i = 0, NUM_CPU_CONSUMER_THREADS do
25:            wg_offsets_out[i] = atom_add(histo[i], l_histo[i])
26:        end for
27:
28:        // Perform an exclusive sum-prefix on the local histograms applying an
29:        // offset and store the result to be used by this work-group in the mid buffer
30:        wg_offsets_mid[0] = get_group_id(0)*get_local_size(0)*KMERS_PER_READ
31:        for i = 0, NUM_CPU_CONSUMER_THREADS do
32:            wg_offsets_mid[i+1] = wg_offsets_mid[i] + l_histo[i]
33:        end for
34:    end if
35:    barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE)
36:
37:    // Write generated k-mers shuffling them locally - within a work-group
38:    for i = 0, KMERS_PER_READ do
39:        if p_kmers[i] == INVALID then continue
40:        end if
41:        offset = wg_offsets_mid[i] + p_offsets[i]
42:        mid[offset] = p_kmers[i]
43:        p_offsets[i]++
44:    end for
45: 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 CPU thread).
1:  procedure SHUFFLE
2:      for i = 0, NUM_CPU_CONSUMER_THREADS do
3:          // Cohesively read all k-mers from this work-group to write
4:          // and shuffle them globally
5:          num_items = wg_offsets_mid[i+1] - wg_offsets_mid[i]
6:          wg_offset_out = offsets[i] + wg_offsets_out[i]
7:          for j = get_local_id(0), num_items, j += get_local_size(0) do
8:              out[wg_offset_out+j] = mid[wg_offsets_mid[i]+j]
9:          end for
10:     end for
11: end procedure
```

The main reason why we split the algorithm into four kernels is that it is the only way to obtain a global synchronization point among GPU threads. Note also that the kernels use extra GPU-side buffers that are required to share data between kernels and to store k-mers in the Encode kernel. Whereas the former is a common practice of GPU programming and occupies tens of MB, the intermediate buffer for the k-mers is a requirement of the algorithm and must be of the same size as an output chunk. In fact, this shuffling algorithm is not an in-place algorithm, and if we were to use the same buffer, some threads might move some k-mers before other threads even start; thus, overwriting them and producing erroneous results. Such a solution considerably reduces CPU-side data movement, providing more vertical scalability to even more CPU threads at the expense of some extra computation and more memory consumption on the GPU side. Extra computation and memory consumption, that, however, is not proportional to the number of Consumer threads.

During this work, we also evaluated the possibility to sort all the k-mers to find duplicates and merge them into a pair <*key, count*> to minimize the PCIe transfer size for the output and the number of insertions in the histogram. This study was motivated by the observation that even within small buffers of around 1 GB, up to 30% of the items are duplicated. However, we stop this path when it became clear that sorting 1 GB of 64-bit items (2^{27} items) takes around one second, and this would slow down the entire pipeline. We also noticed how the OpenCL implementation of the sorting algorithm we tested – Radix sort and Bitonic sort – were much slower the CUDA version. For instance, the OpenCL implementation of the Radix sort from Nvidia was 2.5x slower (436 ms against 1087 ms) than the same CUDA algorithm on a TESLA K40c. Regarding the Bitonic algorithm, to make it conclude faster, we also explored the possibility of stopping the algorithm before it completes all the steps required to sort all the items and to work with smaller sorted sub-chunk. However, empirical results with real data showed that stopping the algorithm even a few steps before it sorts all the items leads to find considerably fewer duplicates, which make the entire effort fruitless.

4.1.3 *Cooperative CPU-Accelerator Construction of Very Large Bloom filters*

As explained earlier in this section, the second main contribution of this work is part of the Prune unit, which is used to build a chain of two Bloom filters that allow discarding k-mers that are seen only once.

The most intuitive way is to offload the Bloom filter lookups of the Count unit. These lookups to the second level Bloom filter are used to check whether every k-mer is unique or not. Offloading these lookups is simple and unburdens the CPU Consumer threads. Differently, in the Prune unit, we are populating the filters on the CPU side so the same cannot be done. However, when building the chain of filters, if an item is already in the second level adding it again is superfluous. Hence, while building the chain of Bloom filters the second level can be offloaded to the GPU to drop those k-mers already on the list. This opposite behavior from the Count unit reduces the number of k-mers that must be processed by the Consumer threads in the Prune unit. Besides, in the Prune unit, the GPU copy of the second level Bloom filter must continuously be updated at every cycle of the pipeline, increasing the communication from the CPU to the GPU.

4.2 VIRTUALIZING ACCELERATORS MEMORY

Albeit straightforward to implement, offloading the Bloom filters used in this k-mer counting algorithm might require more memory than the selected accelerator. Such filters are, in fact, of around 9 GB, which due to the pinned buffer, might even require twice as much in the GPU memory. Characteristic not common neither in high-end GPUs. However, after the Shuffle kernel, when the k-mers are shuffled per Consumer thread, the lookups to the Bloom filter can be split in multiple Bloom filter kernels, one per each Consumer thread. Each of these kernels accesses a different Bloom filter, the filter relative of one CPU Consumer thread.

To relax the requirement on the device memory, we developed a mechanism to virtualize the memory of accelerators in DRAM. This mechanism uses the Orchestrator thread to allocate, in accelerator memory and host DRAM, as much as possible pinned buffers of the size of the Bloom filters. Plus, in DRAM, it also allocates an additional buffer for each of the Bloom filters. Then, before to execute a Bloom filter kernel, the Orchestrator thread makes sure that its relative Bloom filter has been copied to the pinned buffer in DRAM and then also transferred to the pinned buffer in the accelerator. Once a kernel used a particular Bloom filter, the pinned buffers can be overwritten with another Bloom filter to allow the next kernels to execute.

In principle, the idea is simple as it is similar to what the OS virtual memory does for CPU programs. However, when this mechanism gets implemented with the rest of the double buffering pipeline of the application, the entire solution becomes intricate. Figure 13 show the activity involved in each cycle of the pipeline of a simplified example with 4 CPU Consumer threads, and as many Bloom filters kernel, in an accelerator where only 2 Bloom filters can fit at the same time.

1. Enqueue ① main Zero-out, Encode, Prefix-sum and Shuffle kernels ⓐ
2. Enqueue ① asynchronous data transfers ⓑ, ⓗ to write the first pinned buffers to the accelerator.

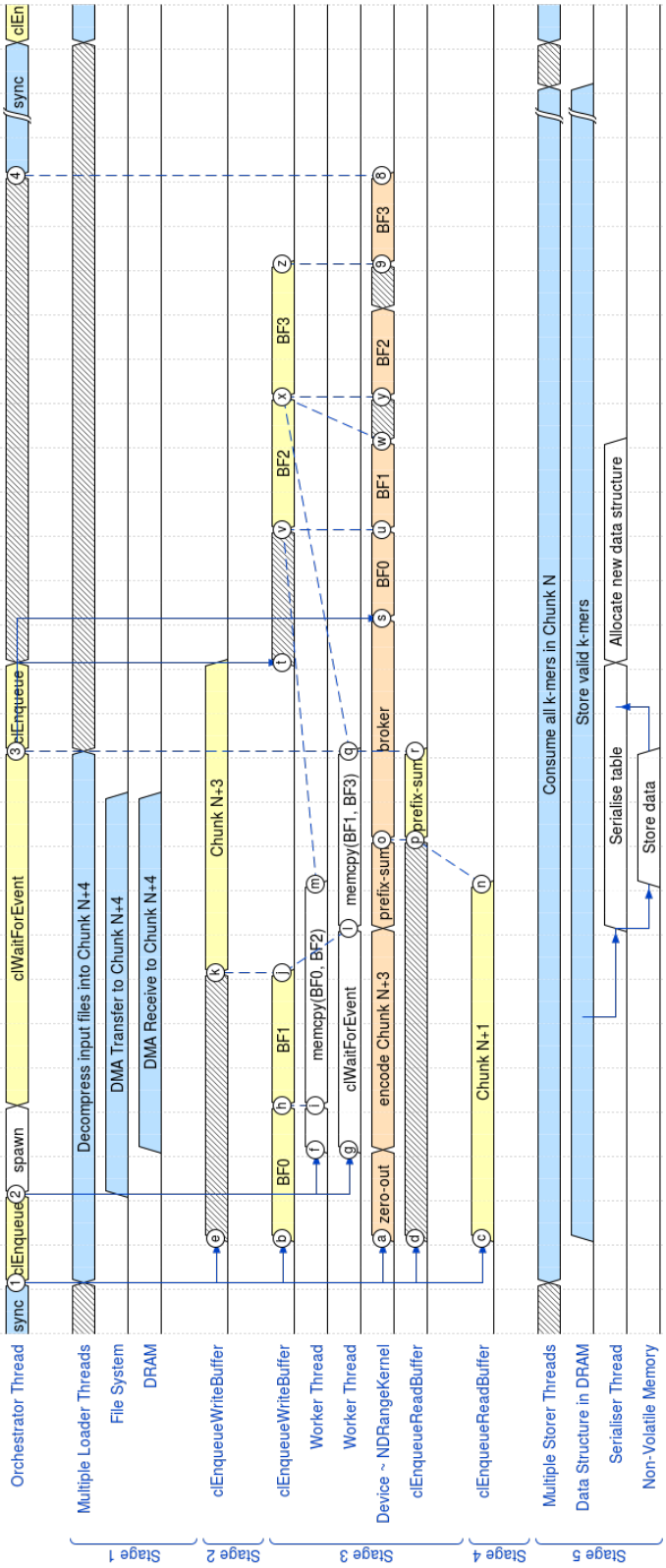


Figure 13: (§4.2) Example of the activity at every cycle of the double buffering pipeline when the hosts virtualizes the accelerator memory.

3. Enqueue (1) asynchronous data transfers (c), (e), and (k) to read and write the output of the previous cycle and the input of the next cycle of the pipeline.

4. Enqueue (1) an asynchronous data transfer (d), (p) to read the global prefix-sum as soon as the Prefix-sum kernel completes (o).

5. Spawn (1) CPU worker threads (f), (g) that wait for the pinned buffer transfer to complete (h), (j) and copy the next Bloom filter to the pinned buffer (i), (l).

6. Wait for the transfer of the prefix-sum (3) and enqueue the first round of Bloom filter Kernels (s), (u).

7. Enqueue asynchronous transfers of next Bloom filters (t), (v), (x). These transfers start only once the relative worker thread finished the copy (m), (q), and the Bloom filter kernel completed (u), (w). At the same time also enqueue the second round of Bloom filter kernels (y), (9). These kernels start only once the respective Bloom filter buffers are transferred in the GPU memory (x), (z).

8. Restart from step #5 until all Bloom filters are copied to the pinned buffer, copied to the accelerator, and its relative kernel executed.

9. As soon last Bloom filters are copied, and while executing last Bloom Filter kernels, spawn worker threads to overwrite the first pinned buffer for the next cycle of the pipeline (not in Figure 13).

Since in the Prune unit, the second level Bloom filters must be updated in any case, this virtualization does not add much of an overhead. On the other hand, in the Count unit, this mechanism required to constantly copy the Bloom filters from the CPU to the GPU at every cycle. This requirement drastically increases the PCIe traffic and might become the bottleneck. Still, without this solution, the program would not be able to run on accelerators with less than 20 GB of on-board memory.

Note that our mechanism is somehow similar to what OpenCL 2.0 Shared Virtual Memory and CUDA Unified Memory offers. However, some GPU vendors do not support OpenCL 2.0. Moreover, even if OpenCL 2.0 Shared Virtual Memory would be available, our mechanism is still valid. In fact, even if it stalls kernels until their buffers are all "swapped in" the accelerator, it prevents costly accelerator-size page faults. Page faults that would instead occur with OpenCL 2.0 Shared Virtual Memory and that, given the random nature of Bloom filters, would substantially penalize the performance of the kernels.

In the scope of this work, we mainly focused on discrete accelerators connected via PCIe, where our virtualization technique makes sense. However, during the work, we also considered solutions like CPU-FPGA hybrid chips where the two chips have a coherent view of the main memory and that are connected via a low-latency interconnect for scalable multiprocessor systems, such as Intel's QPI or the newer UPI. Such chips would remove the need for the virtualization of accelerators' memory. However, an exploratory analysis concluded that this kind of chips is not suitable for our particular use case. The main rationale behind this conclusion is similar to the case of shared virtual memories described above, and it is that our design involves offloading

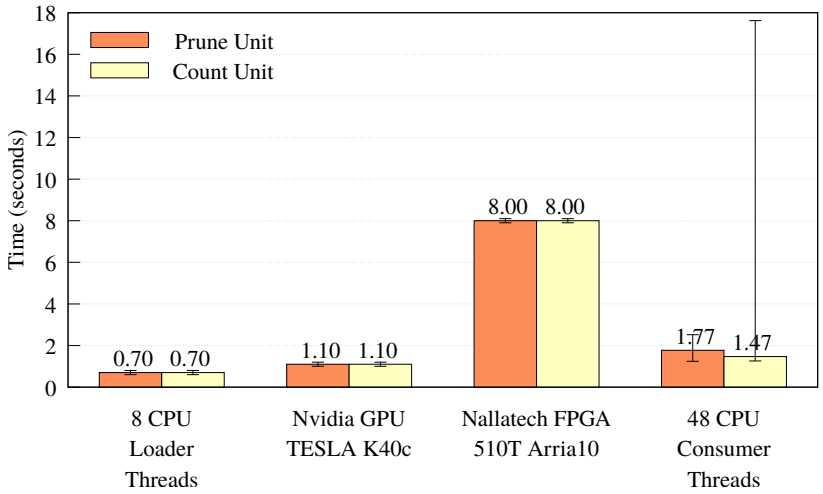


Figure 14: (§3.3) Performance comparison (minimum, maximum and mean time) of GPU-style kernels running on an Nvidia TESLA K40c GPU and a Nallatech 510T FPGA. The comparison also includes the cycle time of Loaders and Consumer threads when running the experiments on the same `FatNode` used in Section 3.4. The maximum cycle time of the 48 CPU Consumer thread is due to stalls during the manual swapping presented in Section 3.3.

lookups to Bloom Filters of the size of some GBs. The size of these data structure is clearly too vast to fit in the on-chip memory of the FPGAs, that is usually around few MBs, and each lookup would trigger many one-bit accesses to system memory, one per each bit that needs to be verified in the Bloom Filters. As a result, this would translate into a huge number of 1-bit requests that dramatically reduce the bandwidth of the interconnect, making this solution not suitable for our kernels.

4.3 PORTING GPUS CODE TO FPGAS

When we started porting the OpenCL GPU code to FPGAs, the very first step was to adapt the kernels and the host code, so to compile it and execute with the target FPGA without major changes. The first exploratory results, depicted in Figure 14, clearly show that the FPGA performance falls far behind all other steps in the pipeline, which makes it the main bottleneck. Compared to the GPU, the FPGA is 7.2x slower (8 seconds against 1.1), suggesting that the GPU parallelism, at least for this code, does not fit on FPGAs and needs to be redesigned.

In this Section we focus, 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 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.

4.3.1 *Reducing Global Memory Usage and Accesses*

As Section 4.1 describes, the GPU algorithm writes to memory all the k-mers two times. 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 is 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 accelerators like 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, that instead 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 CPU Consumer threads. Knowing that the distribution fairly balances how many k-mers belong to each Consumer thread and ensuring that each input chunk is large enough to have tens of millions of k-mers, one can assume the k-mers generated from the DNA reads in each input chunk are evenly distributed among the CPU Consumer threads. 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, to prevent work-items from overwriting each other's results, synchronization among all work-items is required. Besides, to make this solution compatible with the host code used with GPUs, the CPU consumer threads still need the results of the exclusive sum-prefix to know how many k-mers are in the partition of each CPU thread. Obviously, there is always the chance that one Consumer threads get more k-mers than expected. When this occurs, the k-mers of this Consumer thread would overflow, overwriting the k-mers of the next thread and leading to wrong results. To make this unlikely to occur, our solution was complemented by over-provisioning the output buffers to be slightly bigger than required. This expedient wastes a bit of memory and adds more PCIe traffic, but it is a simple and effective solution that proved to work already with an over-provisioning ratio of 1.1. Nonetheless, this does not completely prevent overflows from happening, and it is essential that the application can detect whenever an overflow occurs. For this, the host application ensures that each CPU consumer threads gets no more k-mers than allowed for each output chunk. That is: for CPU thread #i if *histogram[i] > offset[i+1] - offset[i]* then, an overflow occurred and some k-mers of the CPU thread #i were written in the region of memory dedicated to CPU thread #$i + 1$. When this occurs, the host program detects and resolves it rescheduling the kernels on only one half of the input chunk and adding a *bubble* in the first two stages of the double buffering pipeline. In the next cycle of the pipeline, the second half of the chunk is processed, and the first two stages will stall due to the bubble. Since this solution slightly increases the execution time, it is not optimal. However, it proved to be good enough with the over-provisioning of the output buffer, overwrites are very unlikely to happen.

4.3.2 *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. pipelining), and it is a key concept when porting code from one architecture to another. For these reasons, many GPU algorithms require the refactoring of the whole accelerated code to leverage FPGAs fully.

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. 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 35). These instructions make this algorithmic a lousy fit for FPGAs and even unfeasible for those devices that lack some of these functionalities (e.g., 64-bit atomic operations). Motivated by these factors, we redesigned the algorithm to take advantage of FPGA-specific features and optimizations.

4.3.2.1 *Parallel Paradigms*

To better adapt the algorithm to FPGAs, pipelining the OpenCL NDRange kernels used for the GPU were replaced with OpenCL tasks – single work-item kernels using only one thread. With a single work-item, the global histogram can be built directly without the need for atomic operations nor memory barriers. In this way, the code gets extremely simplified, making it a simple build for the FPGA compiler and reducing the number of resources needed. Moreover, since we are now using tasks kernel, rather than NDRange, the Zero-out and Prefix-sum kernels can be removed. In fact, the only reason why these two kernels are separated is that the only way to create a global memory barrier that synchronizes all the work-items within a NDRange is to end the kernel itself. Similarly, also the Bloom filter kernel was changed to a task by adding a loop to process all k-mers one after another.

Even if we changed the parallelism model to a pipeline, loop unrolling could always be exploited to process multiple DNA reads in parallel like Algorithm 3 outlines. In fact, using loop unrolling (line 7 and 16), the Encode kernel can process many DNA reads in parallel, and it generates one k-mer from each of the reads at every cycle.

Algorithm 3 (§4.3.2.1-4.3.2.2) FPGA Encode Kernel (Producer)

```
Input: in (DNA input read and quality markers),
       num_reads (Number of DNA reads in input).
 1: procedure ENCODE
 2:     for chunk_id = 0, num_reads/READS_PER_SUBCHUNK do
 3:         // Load READS_PER_SUBCHUNK DNA reads in bulk to on-chip memory
 4:         read_input (in, chunk_id, l_in)
 5:
 6:         // Generate all k-mers processing the DNA reads in the subchunk in parallel
 7:         #pragma unroll READS_PER_SUBCHUNK
 8:         for read_id = 0, READS_PER_SUBCHUNK do
 9:             for kmer_id = 0, KMERS_PER_READ do
10:                 kmers[read_id][kmer_id] = generate_kmer(l_in, read_id, kmer_id)
11:             end for
12:         end for
13:
14:         // Send one k-mer to each shuffle kernel at every cycle
15:         for kmer_id = 0, KMERS_PER_READ do
16:             #pragma unroll READS_PER_SUBCHUNK
17:             for read_id = 0, READS_PER_SUBCHUNK do
18:                 write_channel_intel(channel[read_id], kmers[read_id][kmer_id]);
19:             end for
20:         end for
21:     end for
22: end procedure
```

4.3.2.2 *Kernels Replication and Channels*

To prevent the need for 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 Figure 15 depicts. In this design, the Encode kernel, described in Algorithm 3, behaves as a producer while the Shuffle kernels, described in Algorithm 4, behave as consumers. Here the Encode kernel processes N DNA reads in parallel, using loop unrolling, and delivers one k-mer to each of the N Shuffle kernels at every FPGA cycle. Instead, each Shuffle kernel reads the k-mers from its dedicated channel, stores them to a dedicated global memory buffer using the predefined offsets to shuffle them, and builds a private histogram. In this way, k-mers are spread equally among the different consumer kernels, each of them receiving the same amount of k-mers. Moreover, now the FPGA design has multiple different output buffers, one per each consumer, we can also replicate the Bloom filter kernels so to filter the k-mers from all, or some, output buffers in parallel.

Algorithm 4 (§4.3.2.2-4.3.3) 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 CPU Consumer thread),
histo (Number of k-mers for each CPU Consumer thread).

```
1:  procedure SHUFFLE #N
2:      histo_cache[NUM_CPU_CONSUMER_THREADS] = {0}
3:      kmers_cache[NUM_CPU_CONSUMER_THREADS][8] = {{INVALID}}
4:      num_kmers = num_reads * KMERS_PER_READ / READS_PER_SUBCHUNK
5:
6:      // Digest one k-mer in each cycle
7:      for kmer_id = 0, num_kmers do
8:          kmer = read_channel_intel(channel[N])
9:          pid = get_cpu_consumer_thread_id(kmer)
10:
11:         // Store the k-mer in the on-chip memory and count only valid ones
12:         kmers_cache[pid][histo_cache[pid] & 0x7] = kmer
13:         if kmer != INVALID AND pid_min <= pid AND pid <= pid_max then
14:             histo_cache[pid]++
15:
16:             // Flush k-mers from the on-chip memory to global memory in bulk
17:             if (histo_cache[pid] & 0x7) == 0x0 then
18:                 out[(histo_cache[pid]) >> 3] = kmers_cache[pid]
19:             end if
20:         end if
21:     end for
22:
23:     // Flush on-chip caches to the global memory
24:     for pid = pid_min, pid_max do
25:         out[(histo_cache[pid]) >> 3] = kmers_cache[pid]
26:         histo[pid] = histo_cache[pid]
27:     end for
28: end procedure
```

This design, based on kernel replication, has different key advantages. First, it removes the need for synchronization. Second, thanks to channels, the data is stream from the producer directly into the consumer kernels without leaving the FPGA chip, eliminating the need to write all k-mers to global memory in the producer kernel and to read them back in the consumer kernels. Third, this approach provides a way to balance 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. Owing to this, not only does the replication factor depend on the FPGAs, but also on the board used and its memory. Section 4.4 describes and discusses this matter with results from our FPGA design.

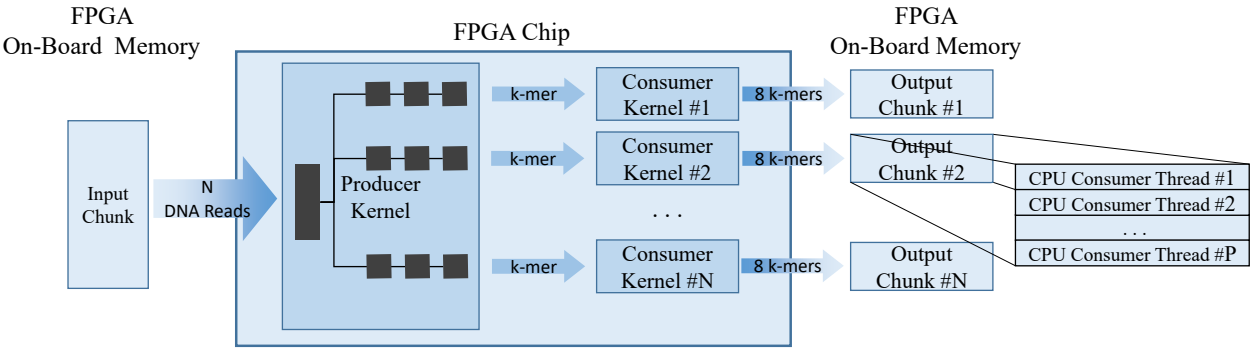


Figure 15: (§4.3.2) Overview of the FPGA Producer-Consumers kernels and data flow. The figure depicts an FPGA design with one producer and N consumers kernels. The Producer kernel reads and processes N DNA reads in parallel using loop unrolling (i.e., #pragma unroll N) and sends one k-mer to each Consumer 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.

4.3.2.3 *On-chip and Global Memory*

The only downside of SMUFIN partitioning scheme is that, since the partitioning criterion only takes into account the k-mer itself, there is no relation between consecutive k-mers of the same DNA read and the partition and CPU Consumer thread to which they belong. 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 use the on-chip memory of the FPGA to create a small cache (*kmers_cache* in Algorithm 4). This cache allows staging up to eight k-mers, per each Consumer thread, so to write them to the global FPGA memory in just one unique 512-bit wide request to use the memory bandwidth at its best.

4.3.3 *Multi-FPGA Support*

In Section 4.2, we described how the virtualization of the memory of accelerators could be used to run the Bloom filter kernels in accelerators with not enough memory. As discusses, the virtualization imposes to update all the Bloom filter (approximately 9 GB) in the accelerator at every cycle of the double buffering pipeline, putting pressure on the PCIe bus. While this might not be a problem with accelerators with PCIe Gen3 x16, it becomes a substantial issue when using devices with a narrower, thus slower, PCIe connection and slower onboard memory technology. When this is the case, the transfers between the host and the accelerator take more time and might become the bottleneck of the entire software pipeline. To overcome these hardware limitations, one possible path is to use multiple accelerators at the same time.

We implemented the support to multiple FPGAs assigning a subset of CPU Consumer threads 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. This kernel was instructed (line 13 of Algorithm 4) to only output those k-mers that belongs to CPU Consumer thread

assigned to the FPGA where the kernel is running (parameter *pid_in* and *pid_in* of Algorithm 4). The advantages of this solution are multiples. First, the Bloom filter kernels are distributed and executed in parallel on multiple FPGAs, decreasing the kernel time. Second, since the kernel parameters are used to tell each FPGA its subset of CPU Consumer threads, this implementation allows reusing the same FPGA design with an arbitrary number of FPGAs without requiring to recompile. Third, spreading the Bloom filters to different accelerators might reduce the memory requirement on each accelerator to the point where the memory of all FPGAs is enough to fit all the buffers. When this happens, there is no more need to virtualize the accelerator memory, which reduces the PCIe transfers. In fact, when this is the case, in the Prune unit, the offloaded Bloom filters can be updated every few cycles of the software pipeline or just a few filters at every cycle. 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 for the entire unit.

4.4 EVALUATION AND RESULTS

In order to evaluate the impact of the presented work, we ran the k-mer counting algorithm of SMUFIN in one node with and without accelerators. For each configuration, we evaluate time-, energy-to-solution, and power consumption. Besides, we analyze the scalability of the single-producer/multiple-consumers FPGA design and discuss on downsides and benefits of the accelerators used. Finally, we profile the fastest configuration to understand its bottlenecks, and we explore how to reduce its energy-to-solution.

4.4.1 *Experimental Setup*

We conducted our experiments on a node similar to the `FatNode` used in Chapter 3, called `FatNode2`. This time, for 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 a memory extension. Both NVMe drives 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 ECC enabled and GPUBoost disabled. On the FPGAs side, we used a Nallatech 510T board equipped with two independent Arria 10 1150 GX FPGAs. Both FPGAs sport a maximum frequency of 450 MHz and come with 16 GB of DDR4-2133 memory DIMMs in a 4-bank configuration. However, at the time of writing, OpenCL is able to use only two of the four banks per FPGA, reducing the available memory to 8 GB. Even if hosted on the same board, the two FPGAs are independent, and, as shown in Figure 16, they only share the PCIe Gen3 x16 bus and an optional onboard 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. We compiled the FPGA design for

Table 4: (§4.4.1) Experimental Setup of `FatNode2`

	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	2x120 W	250 W	112.5 W
PCIe Interface	-	Gen3 x16	Gen3 x8
PCIe Dual Copy Engine	-	Yes	No

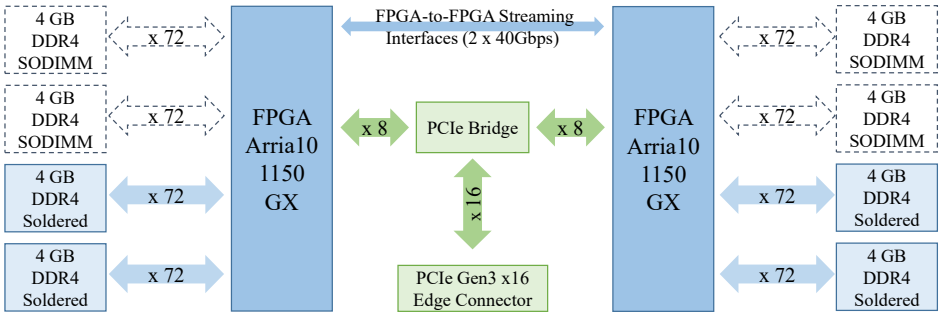


Figure 16: (§4.4.1) Diagram of the Nallatech 510T dual-FPGA board.

a different number of Shuffle kernel replicas, and we use the faster one - the design with eight replicas. Table 4 reports the main specification of the host system and of the accelerators.

4.4.2 *Evaluation Methodology*

We executed using four different hardware configurations: CPU only, CPU plus one FPGA, CPU plus two FPGAs, and CPU plus one GPU. In each configuration, we always used 8 CPU loader threads and 48 CPU consumer threads; thus, 48 different Bloom filters. In each execution, we process the same input genomes used in Chapter 3.4. With accelerators, the double buffering pipeline split the input data set in 1630 input chunks of 400 MB each, requiring as many cycles to process the whole input. Accelerators unburdened both kinds of CPU threads and removed communication from loader to consumer threads, communication that in the CPU only configuration takes place via $8 \cdot 48$ dedicated single-producer single-consumer queues. When only one accelerator is used, device memory was not enough to fit all the 48 Bloom filters; thus, the application virtualized the device memory in host DRAM as outlined in Section 4.2. Meanwhile, with two FPGAs, their aggregated memory was enough not to require the virtualization. This considerably reduced the PCIe traffic in both Prune and Count units. In fact, as discussed in 4.3.3, in the Prune

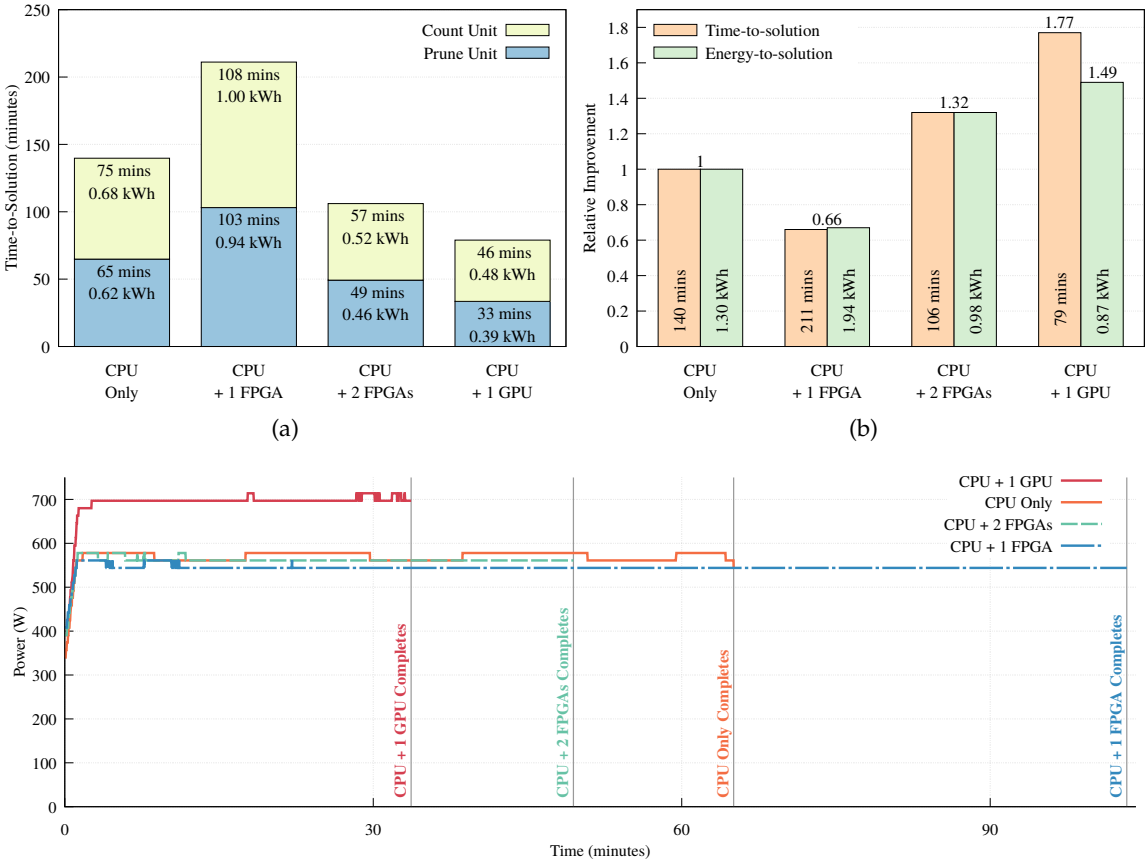


Figure 17: (§4.4.3) Time-to-solution and energy-to-solution (a-b) and Power Consumption (c) of SMUFIN k-mers counting without acceleration, with one FPGA, with two FPGAs, and with one GPU. Plot (c) only shows the power consumption for the Prune unit, data relative to the Count unit was omitted because very similar.

unit, Bloom filters are updated once every eight cycles of the software pipeline instead than at 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, since this unit does not alter the Bloom filter, there is no need to update them constantly.

To collect power consumption samples and energy consumption, we used a metered-by-outlet PDU (Power Distribution Unit) and also IPMI (Intelligent Platform Management Interface) to validate the results. Lastly, to prevent accounting for the power consumption of idle accelerators, we disabled the PCIe slot of the unused accelerator from the BIOS.

4.4.3 *Considerations on Time-, Energy-to-Solution, and Power Consumption*

Figures 17a and 17b 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

Figure 17b depicts, while with two FPGAs there is an improvement of 1.32x in both time- and energy-to-solution, with one GPU the improvement is of 1.77x for the time-to-solution but of a lower 1.49x for the energy-to-solution. This difference between the improvement on the time and energy of the configuration with one GPU is to be attributed to the higher power consumption of the GPU. In fact, as Figure 17c illustrates, with one GPU, the system consumes a considerable amount of extra power. Besides, this Figure also shows how power consumption with one and two FPGAs is similar to the power of the system without accelerators. Unexpectedly, in some cases, the power consumption with FPGAs is even lower than the power with only the CPU. This fact demonstrates how FPGAs can be more power-efficient means than the CPU. In complete fairness, it must be noted that as Table 5 reports, not all the FPGA resources are being used. Lastly, the difference in the power consumption between two FPGAs and one GPU, suggests that if we were able to shorten the execution time of the configuration with two FPGAs of at least 12 minutes, this would become the most energy-efficient setup, even if still slower than the configuration using one GPU.

4.4.4 *Performance and Scalability of OpenCL Kernels*

As we outlined in section 4.3.2.2, 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 where the internal parallelism of the produced kernel, obtained with loop unrolling, equals the number of consumer kernels.

Figure 18a shows the execution time of the kernels on GPU and with one and two FPGAs. The Figure offers a breakdown of the kernel time into "Encode & Shuffle" kernels and the Bloom filter kernels. Whereas in the FPGA, the "Encode & Shuffle" kernels are executed in parallel; in the GPU, the "Encode & Shuffle" refers to the execution time of the four kernels including the Zero-out and Prefix-sum kernels. With multiple devices and kernels that are executed concurrently, the plot reports the execution time of the slower kernel among all devices. Besides, the time related to the Bloom filters kernels is the aggregated time for all kernels that are executed one after the other, which are 48 with one accelerator and 24 with the 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, independently from how many FPGAs are being used. As a result, when increasing the number of devices, the only difference for the Encode and Shuffle kernel is in the amount of data that the Shuffle kernel write 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; thus, 512 MB on each FPGA.

Moreover, Figure 18b illustrates that the kernel execution time of only the Encode and Shuffle kernels scales linearly to up to four replicas. With eight replicas, instead, the improvement starts to be sub-linear. Here, we identify two primary sources for this sub-linear scaling. First, as Table 5 shows, we note that with 8 replicas, the frequency of the FPGA design is the lowest one. However, this frequency drop is not enough to justify alone all the performance lost. Instead, it is more likely that the lower performance is due to the increased amount of resources used within the FPGA chip. Second, we observe that with more replicas, the kernel time gets shorter, but the same

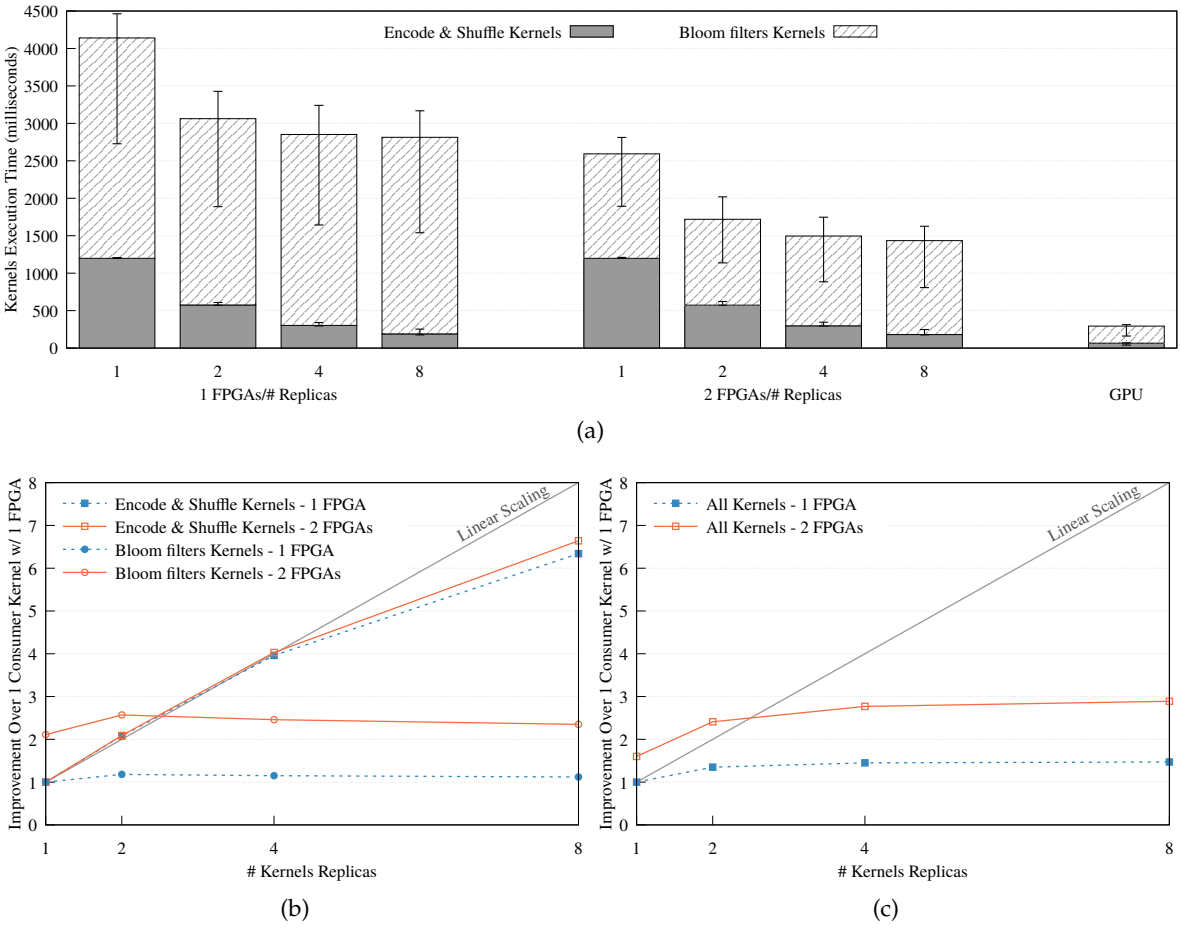


Figure 18: (§4.4.4) Performance comparison of OpenCL kernels running on FPGAs (with different kernel replications) and GPU (a), and scalability of the FPGA kernels (b-c). Data is relative to the 1630 circa cycles of the software pipeline required to process the whole input.

amount of data is written to global memory. Thus, we might suppose that with 8 replicas, the design is already putting enough pressure to saturate the on-board DRAM subsystem. Figure 18b seems to support this thesis. In fact, this Figure shows how, with eight replicas, the execution time with two FPGAs, each writing only half of the output buffer, is slightly better than with one FPGA, that writes the whole output buffer alone.

On the Bloom filter kernels side instead, Figures 18b and 18c illustrate how these kernels do not scale with the number of parallel kernels. However, as we split the Bloom filter kernels between devices, they do scale linearly with the number of FPGAs. Finally, Figure 18c details how the execution of the Bloom filters dominates the kernels time, particularly on FPGAs. Intuitively, the long execution time of these kernels is due to their random memory accesses required to perform lookups in the Bloom filter. This is particularly costly on the FPGAs, equipped with only 2 banks of DDR4 DIMMs. To this end, like [92] suggests, next-generation FPGA boards equipped with faster memory technology like HMC (Hybrid Memory Cube), are expected to be much more competitive.

Table 5: (§ 4.4.4) FPGA Resources used, Clock Frequency, and Compilation Time of the FPGA design. The amount of resources dedicated to the OpenCL environment does not exceed 8% of the available resources.

# 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 minutes
2	19 %	35 %	5 %	15 %	27 %	234.3 MHz	142 minutes
4	27 %	35 %	11 %	22 %	42 %	227.5 MHz	272 minutes
8	44 %	35 %	21 %	39 %	72 %	211.9 MHz	351 minutes

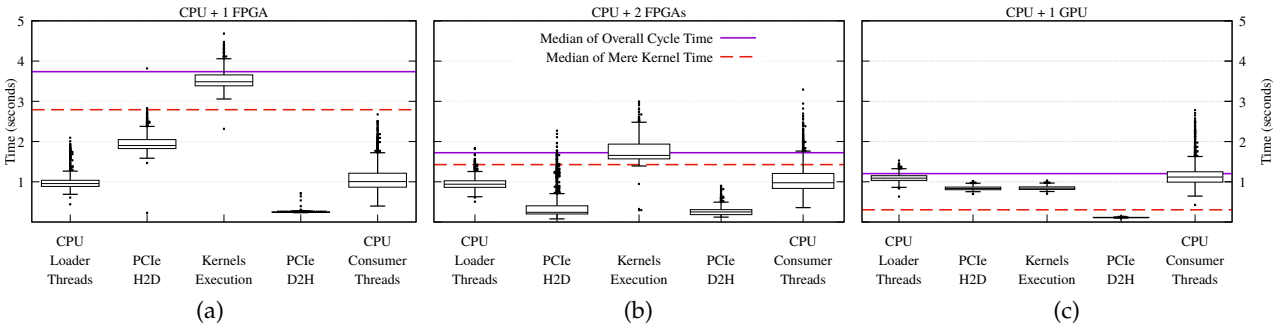


Figure 19: (§4.4.5) Distribution of execution time of each individual stage of the software pipeline using one FPGA (a), two FPGAs (b), and one GPU (c) to execute the Prune unit. The distribution is relative to the 1600 circa cycles needed to process the whole input data set.

Besides, Figure 18a shows that the best FPGA design, the one with eight Shuffle kernels, is actually the design with the lowest frequency, as Table 5 reports, highlighting how the frequency is not the only important metric.

Finally, Figure 18a also shows that with the best FPGA configuration, two FPGAs, and eight kernel replicas, the kernel time is approximately 4.9x slower than on the GPU. This result does not reflect the much smaller difference in the performance of the entire Prune and Count units, showed by Figure 17a, between the acceleration with two FPGAs and with one GPU. This demonstrates that to evaluate one board; one should consider the execution time of the entire application and not only the mere execution time of kernels. As a matter of fact, the overall execution can always be impeded by third factors such as host-device synchronization, poor PCIe performance, and inadequate overlapping of PCIe transfers and kernels execution.

4.4.5 *Final Considerations on GPU and FPGAs Performance*

Even considering the different FPGAs and GPU components, it is not easy to identify the real reason why the setup with two FPGAs falls behind the one with one GPU. Figure 18 suggests that the FPGA kernels as the main bottleneck. However, when it comes to evaluating one board for an application, one must look beyond the mere kernel execution time. Particularly, when

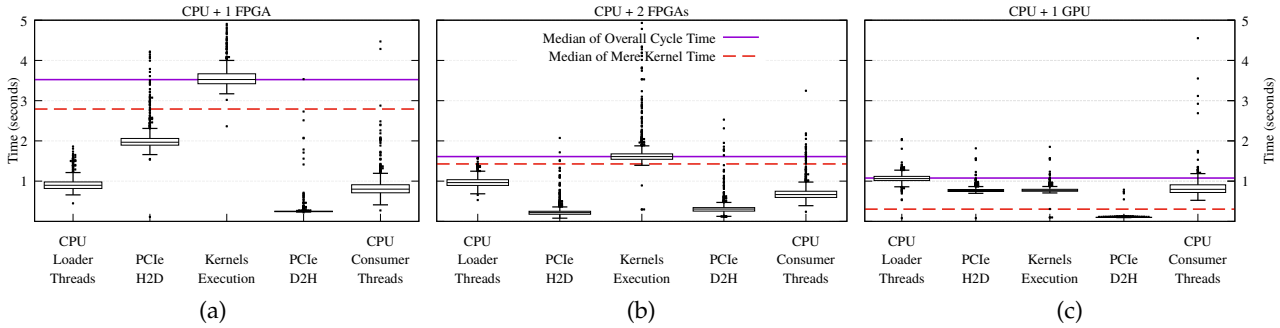


Figure 20: (§4.4.5) Distribution of execution time of each individual stage of the software pipeline using one FPGA (a), two FPGAs (b), and one GPU (c) to execute the Count unit. The distribution is relative to the 1600 circa cycles needed to process the whole input data set.

PCIe transfers and kernels execution are overlapped. In order to better understand the issues and bottlenecks of our solution Figures 19 and 20 show the distribution of the execution time of each stage of the double buffering 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, and Plots 19a, 19b, 20a, and 20b prove that the kernels execution is the bottleneck for the configurations using FPGAs. On the other hand, Plots 19c and 20c prove that the GPU is not always busy and how, in this case, the CPU threads are the main bottleneck; explaining why the difference in the kernel times between GPU and FPGAs is not reflected in the execution time of the application.

One of the first aspects that must be taken into consideration is the PCIe connection. In our case, not only do the FPGAs have a narrower and slower connection, but also they lack of a dual copy engine. As a consequence, whereas the GPU is capable of overlapping transfers in both directions, transfers to and from the FPGAs are serialized. As a result, with one FPGA, the PCIe transfers time, in both directions, is higher than with one GPU.

A second aspect that should also be examined is if there are any dependencies between kernels and PCIe transfers, that is: transfers or kernels that cannot start until another event concluded. Our proposed accelerated k-mer counting algorithm tries to solve these dependencies with the double buffering pipeline. However, the virtualization of accelerator's memory in host DRAM introduces dependencies between Bloom filter kernels and the PCIe transfers needed to copy the Bloom filter in the accelerator memory. One can observe the effect of these dependencies in all Plots with just one accelerator, where the median of the box plots relative to the kernel execution that is significantly higher than the median of the mere kernel execution time, represented by the dashed horizontal line. This difference represents a rough estimation of the performance penalty of the memory virtualization, and it explains why, with two FPGAs, the gap between the two medians is much narrower.

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 already in process. Consequently, this limitation stalls the FPGA chip in executing the following kernels, exacerbating the kernel execution time of the entire software. Furthermore, this also justifies the difference between the box plot of the kernels execution cycle time and the median of the mere kernel execution in Figure 20b. As a matter of fact, in this case, the virtualization of memory is not active, and there are no dependencies between PCIe transfers and kernels; thus, 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. In addition, we can deduce that, in order to be more competitive, future FPGAs board should be equipped with faster memory and adequate PCIe subsystem, including multiple memory engine to allow full overlapping of transfers and kernels. If manufactures will be able to design boards with these characteristics, while also maintaining a similar power budget, then FPGAs could become a real asset for future energy-efficient genomics workloads.

4.4.6 *Characterization of the Accelerated Version*

Figures 20c and 19c prove that in the fastest configuration neither the GPU nor the PCIe transfers are the bottleneck. Instead, the application is slowed down by the 48 CPU Consumer threads that are not able to consume the items fast enough to keep up with the speed of the GPU. To better understand what the real slowing factor is, we collected traces regarding the system utilization. These traces confirmed a CPU utilization close to 100%, and they also showed that, even if fully utilized, the CPU is ineffective and delivers a poor IPC (Instruction Per Cycle) throughout the execution. As a matter of fact, the Xeon CPU used has a maximum IPC of 4, but during the execution of both Prune and Count unit, the CPU never delivers more than 2 instructions per cycles and shows and an average of 1.6. Further profiling showed that around one-third of CPU cycles are wasted in stalls due to L1D cache misses, and that up to half of the CPU cycles are, instead, spent waiting for the memory subsystem. These results suggest that the random updates, to the Bloom filters in the Prune unit and the hash tables in the Count unit, are saturating the memory subsystem and the CPU threads are stalling to wait to be served.

In an effort to mitigate the performance gap between the CPU and the memory, we tried different solutions. First, to try to reduce the gap, we tried to manually prefetch, from memory into cache, the parts of the Bloom filters. This attempt showed no benefits and proved that the compiler and the kernel are doing an excellent job in prefetching. Second, we tried to reduce the performance gap by lowering the running frequency of the CPU. Here the goal was not to reduce the time-to-solution but instead the energy-to-solution. To explore this path, we disabled Intel P-State Driver, and we tested multiple frequencies within the 1.20-3.30 GHz range. Results show that power consumption and execution time started to stabilize at around 2.6 GHz even when forced at a higher frequency. This result proved that the Intel P-State Driver throttles the CPU even if running with the *Performance* power governor and that in our case, even if running at 3 GHz it was doing a good job throttling at around 2.6 GHz. Moreover, the fact that even when we set higher frequency, the CPU was running at lower ones demonstrated that, and we quote, *"For contemporary Intel processors, the frequency is controlled by the processor itself and the P-State exposed to*

software is related to performance levels. The idea that frequency can be set to a single frequency is fictional for Intel Core processors. Even if the scaling driver selects a single P-State, the actual frequency the processor will run at is selected by the processor itself."[66]. The third and last attempt to mitigate the performance gap between memory and CPU was to try to reduce memory latency using all the different QPI (QuickPath Interconnect) snoop protocol possible: *Home Snoop* (default), *Early Snoop* and *COD (Cluster-on-die)*. As shown in [94] each protocol suits a different kind of workloads: Home Snoop increases sustained bandwidth, Early Snoop shortens the latency between sockets, and COD reduces the latency for local memory accesses. Results showed that whereas Early Snoop shortened the time-to-solution of 5%, COD instead increased it of 42%.

In conclusion, the memory latency should be considered the bottleneck, and the poor data locality of the memory accesses to the Bloom filters and hash tables is to be blamed for the steady 100% CPU utilization but poor IPC. For this specific workload, manual pre-fetching of data in CPU cache and manual frequency scaling of the CPU do not yield any improvement, and they can hardly do better than the compiler and than the kernel. For workloads with many random memory accesses, the NUMA snoop protocol can make a substantial difference, and either the default Home Snoop or the Early Snoop protocol are suggested. At last, considering the size of the data structures used, larger CPUs caches would provide hardly any benefit.

4.5 RELATED WORK

4.5.0.1 *Offloading Genomics Workload to Accelerators*

With the broad adoption of heterogeneous computing, many efforts has been done to exploit GPUs and FPGAs in genomics. This includes Smith-Waterman algorithm [116, 109, 119, 128, 46], Burrows-Wheeler Aligner (BWA) [68, 2, 64, 47], and DNA Assembly with De Bruijn Graphs [103, 104].

Many explored the acceleration part of the GATK (Genome Analysis Tool Kit) with FPGAs [52, 102] and with GPUs [105, 106]. In [52] the PairHMM algorithm of GATK was mapped into a 2D systolic array on FPGAs using OpenCL. Results showed 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, once more, the importance of considering the overall impact on the application.

Another example of accelerated genomics tool is BLAST, a local alignment search tool, which has been accelerated adopting FPGAs [34, 70]. In [70], the authors offloaded Bloom filters to discard a large fraction of a genome. The same strategy ported to GPUs was instead implemented in [88]. However, both cases used Bloom filters smaller than 1 MB. Differently, our implementation uses much bigger Bloom filters of approximately 9 GB in total. In [86] instead, counting Bloom filters are used to obtain similar results, but this kind of Bloom filters has a memory footprint that is 3 to 4 times higher than the normal filter; which cannot be afforded with Bloom filters of the size like the one used in SMUFIN (GB-size). The work presented in [92] 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.

To ease the offloading of genomics application to GPUs, some CUDA libraries for DNA read alignment are also available. BarraCUDA [81, 68, 73] implements a read alignment package that is based on backward search with Burrows-Wheeler Transform (BWT), to efficiently align short sequencing reads against a large reference sequence such as the human genome, allowing mismatches and gaps. To overcome the limited amount of on-board memory, BarraCUDA can perform inexact matches [68]. It also supports multiple GPUs [68], but whether this is more cost-effective or more energy-efficient than a node without accelerators has not been discussed. GASAL2 [3] is another library for accelerating DNA/RNA sequence alignment algorithms to GPUs using CUDA. Currently, it supports local, semi-global, global, and tile-based banded alignments. Besides, it allows full overlapping of CPU and GPU execution. Another library for genomics application on GPUs is the recently released Clara Genomics Analysis SDK from NVIDIA [41]. This library provides GPU-accelerated partial and global alignment algorithms. Authors claim accelerator factors over a CPU implementation up to 70x, but no technical analyses of performance and hardware configurations were to be found.

4.5.1 *Performance Portability Between GPUs and FPGAs*

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 from GPUs 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, [51] shows how channels and multiple kernels can be used to spread the work to multiple pipelines (kernels). Similarly, [132] 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 fully-fledged real-world applications. In fact, most of the literature focuses either on synthetic benchmarks like in [96, 145], or only on the kernel time completely disregarding PCIe transfers between host and device, and without showing the overall impact on the application.

Among the works that discuss the performance portability of OpenCL for GPUs and FPGAs on fully-fledged real-world applications we found [52, 124, 8]. In [124] 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. This result proves that it 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 faster dual-GPU implementation.

4.6 FINAL CONSIDERATIONS

In this chapter, we discussed and analyzed a hardware-software co-design to offload some compute-intensive operation of the k-mer counting algorithm to programmable accelerators with the goal to reduce the time- and energy-to-solution. Results showed that offloading to two FPGAs yield an improvement of 1.32x on both time- and energy-to-solution. Results also proved how the GPU outperforms the two FPGAs with an improvement of 1.77x on the energy-to-solution but, due to higher power consumption, of a lower 1.49x on the energy-to-solution. A thorough analysis of the acceleration pipeline uncovered that in the faster execution, with one GPU, the main bottleneck is the main memory latency of the CPU that must be paid to perform the random accesses to the Bloom filters and to the hash tables; suggesting that the algorithm could benefit from building the main hash table in a different manner that minimizes the random updates.

5

MEMORY FOOTPRINT REDUCTION THROUGH FLASH KEY-VALUE STORE

This chapter describes the third and last contribution of this thesis. This contribution has the goal of reducing the memory footprint of data-intensive applications using flash storage. The contribution is motivated by the observation that new flash NVM technologies are getting better and cheaper, and they constitute a valid alternative to DDR SDRAM. DDR memory that, instead, is a stable technology whose prices aren't decreasing. Besides, in the case of SMUFIN k-mer counting algorithm, the results presented in Section 4.4 suggested that even on enterprise machines with tens of CPU cores, the bottleneck is the main memory latency and different ways to build the histogram could be beneficial. Therefore, the general idea is to change the algorithm to remove or minimize the random updates and to move memory-resident data structures from DRAM to commodity PCIe NVMe and SATA SSD drives to target desktop-class machines rather than power-hungry enterprise machines with hundreds of GB of main memory and high-end CPUs. The consequences of running genomics applications on desktop machines are enormous and bring the long-awaited precision medicine closer to reality. In particular, this would allow laboratories, hospitals, and clinics to run the methods in-house without an enterprise infrastructure; thus, reducing both capital and operational costs.

This third contribution has different outcomes. First, in Section 5.1 we present a method to generate and access large intermediate data structures in flash storage as opposed to DRAM memory. Second, in Sections 5.2 and 5.3 we propose a novel key-value store and cache implementation that uses domain-specific information to perform efficient in-memory caching. Next, in Section 5.4 we evaluate the method and discuss results. Here we provide scalability and cost analysis of a solution based on a desktop-class machine furbished with NVM drives against the usual enterprise-class machine with hundreds of GB of main memory. Finally, we compare our key-value store against RocksDB and prove how multiple CPU threads and asynchronous direct I/O with deep queues are required to exploit the high level of parallelism offered by flash storage.

5.1 K-MER COUNTING WITH SORT-REDUCE

Data-intensive applications can be characterized by large (TB-size) intermediate data structures whose random-access requirements imply that they must be stored in the main memory (DRAM) for good performance. One alternative to building vast data structures using flash storage rather than DRAM is Sort-Reduce. Sort-Reduce is an algorithm and library to sequentialize fine-grained random read-modify-writes, or updates, into secondary storage. Because most flash storage devices have coarse, multi-KB, page-level access granularity, updating fine-grained values in

secondary storage incurs a large write amplification. For example, reading and writing an 8 KB page in order to read and update a 4-byte value results in a 2048-fold write amplification, resulting in 2048-fold performance degradation. Instead, Sort-Reduce collects fine-grained updates in a list and sorts them by location to sequentialize them. This works best in systems where each location is the target of multiple updates as in computing histograms or graph analytics. For graph analytics, Sort-Reduce was successfully used in BigSparse [57] and GraFBoost [58], and proved to deliver performances comparable with systems with much larger DRAM capacity.

The general Sort-Reduce algorithm takes a list of key-value pairs, sorts them by the key, and reduces the items with matching keys during the sorting process. Since the list of key-value pairs is expected to be much larger than the total capacity of the DRAM, Sort-Reduce uses a two-phase external sorting technique. In the first phase, Sort-Reduce repeatedly brings in blocks of key-value pairs as big as the available DRAM, sorts them, reduces them, and then stores the sorted block back in external storage. In the second phase, it merges several of these sorted blocks to produce a bigger sorted-and-reduced block. The merging process is repeated until the key-value list is completely sorted. This repeated streaming of data from flash avoids the need to store either the key-value list or the whole histogram in DRAM. Sort-Reduce uses different internal optimizations to exploit the full bandwidth of the flash storage, such as: aligning writes to flash pages, multithreaded asynchronous I/O, and write chunks of the size of multiple flash pages. Besides, the reduce operation performed during the sort significantly reduce the amount of storage I/O needed during the construction of the histogram.

We use Sort-Reduce as an alternative means to build a histogram for k-mer counting in a small amount of DRAM, for instance, 32 GB and terabytes of NAND-flash storage. We explore this technique by adapting SMUFIN Count unit to add k-mers to Sort-Reduce instead of adding them to the in-memory hash table. Besides, since the memory budged of the Prune unit goes beyond our target 32 GB, we also remove it and create a histogram containing all items, also the unique ones. On the other side, SMUFIN-F keeps the size of the histogram to its minimum, preventing to store the many zeroes in the root format. SMUFIN-F accomplishes this by creating a histogram indexed using k-mers, instead of roots. To maintain the locality of k-mers with the same root, the k-mers are sorted using a comparison function that compares the root first. The implementation of this comparison function is done by producing an integer key for each k-mer, where sorting by this key results sorting all k-mers as desired. As Figure 21 shows, this k-mer format reduces the size of the histogram by as much as 90%, and it is particularly required now that we also add unique items to the histograms.

To use Sort-Reduce to construct a k-mer indexed histogram, we use the k-mers as keys and the pair of counters for the k-mer, one for the normal sample and one for the tumoral sample, as the value of each key. As each DNA read is processed, a histogram entry is created for each k-mer in the read and is sent to Sort-Reduce for producing the histogram. Sort-Reduce sorts these entries by k-mer using the comparison function to group k-mers of the same root next to each other. This function merges entries for the same k-mer by adding together the counters of each entry, making sure that no overflow occurs.

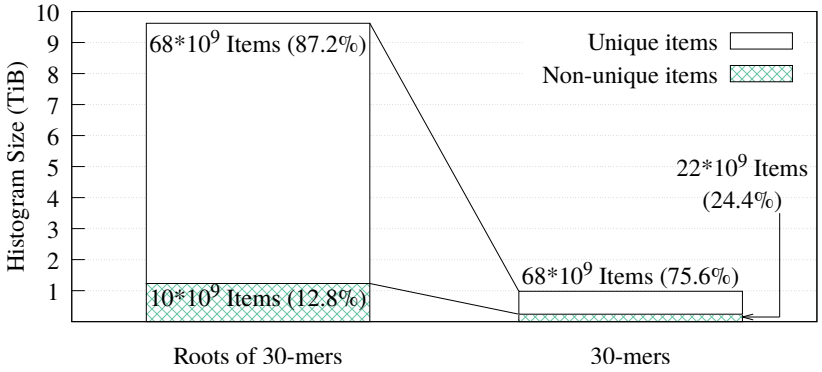


Figure 21: (§5.1) Size of the k-mer counting histogram using the root format (136 Bytes per entry) and the denser k-mer format (12 Bytes per entry).

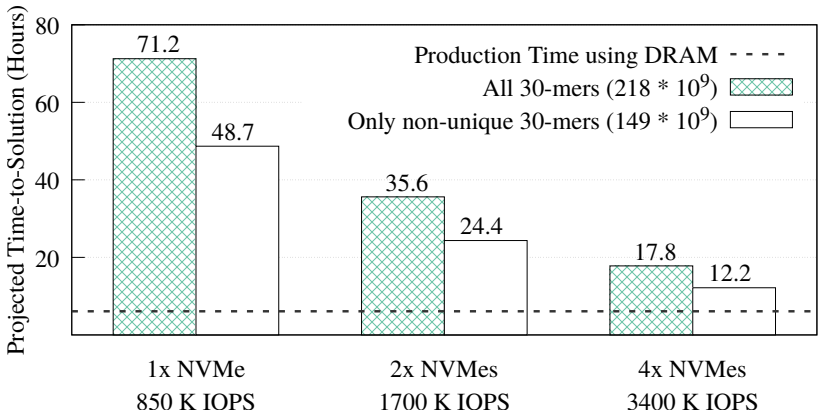


Figure 22: (§5.2) Optimistic projection of the time-to-solution for SMUFIN's Label unit considering only the time to perform the lookups on an hypothetical RAID-0 of NVMe drives, each able to deliver up to 850 K 4KiB random reads. The data series *"All 30-mers"* represents the projected execution time to perform a lookup per each of the 218 billion 30-mers in the input samples. The data series *"Only non-unique 30-mers"* represent the projected execution time if only the positive lookups reach the NVMes, e.g., using a Bloom filter in DRAM to stop the negative lookups, given by unique items, from reaching the NVMes.

5.2 KEY-VALUE STORE FOR HISTOGRAM LOOKUP

Thanks to Sort-Reduce, we are now able to build vast histograms in nodes with a limited DRAM budget. However, naively moving main data structures from DRAM to storage can incur a heavy performance penalty, particularly for applications that would access the histogram constantly. This is the case of SMUFIN Label unit and all those of genomics applications that need to consult the frequency of k-mers. In the particular case of SMUFIN, each k-mer in the input sample must be looked-up to in order to determine whether it is interesting, and so also the DNA read it belongs, or not. This translates to hundreds of billions of lookups per each patient, and, as Figure 22 shows, a naive histogram on flash storage would lead to long execution times even with many high-end NVMe drives. This projection shows that we may get, at best, within a factor of two of DRAM performance. Moreover, this is an optimistic projection based on the nominal SSD throughput; in reality, we would see lower performance.

5.2.1 *Key-Value Store Overview*

To limit the performance penalty of using a histogram on flash storage, we develop a custom key-value store that takes advantage of the following application-specific properties of the histogram and how it is used:

1. Given a histogram as a sorted list of key-value pairs, we first *insert* each key-value pair in the key-value store. During the construction of the key-value store, the application can remove the items with too low frequency, in the case of SMUFIN this is where unique items are removed. After all the insertions have been done, the key-value store becomes immutable because the Label unit only performs lookups to access the pair of counter values associated with a k-mer. Besides, in order to hide the potentially high latency of lookups involving storage access, we implement split-phase or asynchronous lookups that use a scoreboard to keep track of multiple in-flight requests.

2. Since we do not insert unique k-mers in the key-value store, a search for any unique key-value returns nothing. As many as 30% of the lookups fall in this category. We use Bloom filters to expedite negative searches.

3. There is neither spatial nor temporal locality between consecutive lookups to different roots. However, we know a histogram's most commonly accessed entries, because the counters in the histogram represent the number of lookups for each entry. Using this information, we can statically construct a very efficient cache (in DRAM) to store the most frequent entries.

Using these insights, we constructed a key-value store and a key-value cache for SMUFIN Label unit, but that can be used in other genomics applications.

5.2.2 *Histogram as a Multilevel-Index Data Structure*

The root-indexed histogram from the Count unit can be viewed as a sorted list of key-value pairs. A data structure that points to each item individually would be too large to fit in DRAM, so we split the list into an array of consecutive fixed-size chunks, and create an indexing data structure to find the chunk in which a key resides. A good size for the chunks is the size of a flash page, i.e., 4 KB. We can prevent storing pointers to these chunks by creating an array of keys where the indices of the array match the indices of the chunks, and each key in the index is the first key found in the corresponding chunk. For 1 TB of key-value data, this array of non-unique roots is about 10 million elements. Since the keys are sorted, this array can be searched efficiently with a search tree.

Since this array is much larger than the L3 cache of a typical processor, each level of the binary search is likely to cause a cache miss, bringing in an entire cache line typically for just a single value. One way to reduce cache misses is to create a multilevel-index with a fixed fan-out k, e.g., 64. Each level of this k-ary tree uses implicit indices into the next level of the index, i.e., the i^{th} key in one level's array corresponds to the range of keys from $k \cdot i$ to $k \cdot (i+1) - 1$ in the level array below. By using k consecutive keys for the search in each level, a single cache miss brings in the cache multiple keys, useful for the search. This reduces the overall cache misses

observed per lookup and improves the performance. This space-efficient indexing is superior to traditional B+ trees that would require to store additional pointers. For a 1-TB key-value sorted list split into 4-KiB chunks with 8-B keys and a fan-out $k = 64$, the index would require only ($\lfloor \log_k(1 \cdot 10^{12}/4 \cdot 10^3) \rfloor =$) 4 levels and its memory footprint would be around 260 MiB.

5.2.3 *I/O Access Method*

To hide the storage latency of lookups, we use Linux asynchronous I/O. With asynchronous I/O, the application threads schedule I/O operations using the `io_submit(2)` system call. This system call does not block the calling thread, which is free to carry on with other work or schedule more I/O operations, and the I/O operation runs in parallel with the normal thread execution. A distinct system call, `io_getevents(2)`, is used to wait for and collect the results of completed I/O operations. Furthermore, we combine asynchronous I/O with Direct I/O to bypass the kernel's page cache and to obtain an asynchronous direct I/O with no context switches and no extra copies. Linux asynchronous requires every I/O operation to be aligned to the block size of the storage, but this comes for free in our case because the I/O is already designed to be aligned via the index.

Asynchronous direct I/O is able to provide a great performance improvement, but it comes at the cost of software complexity. In fact, its adoption requires two expedients: (i) to align every I/O operation to the block size of the storage and (ii) the application must be written to deal with asynchronous I/O. While, in our case, the first comes for free because the I/O was already aligned with the index. The second expedient, instead, requires to change the key-value store, to use scoreboards that keep track of multiple in-flight requests, and the application, to be able to store all information regarding a key for later processing.

5.2.4 *Unnecessary Database Features*

Like in other domains, general-purpose products offer features to meet the needs of many use-cases. In the case of key-value stores, there are plenty of options to configure depending on the specific workload, and surely not all use-cases need each and every feature. Our insights on the read-only workload make it easy to identify general-purpose database features that are unnecessary for our application. For instance, since we know that the histogram is built in the Count unit and only read in the Label unit, features like garbage collection, compaction, transactions, and snapshots are unnecessary. Similarly, SMUFIN has its own checkpoint mechanism; thus, crash recovery can also be overlooked. Finally, other features like compression and a temporal or spatial cache would add extra latency without benefits.

5.3 SPLITTING THE HISTOGRAM INTO KEY-VALUE CACHE AND STORE

Due to repeated sequences in the input reads – patterns of nucleic acids (DNA or RNA) that occur in multiple copies throughout the genome – there are a few items in the histogram that have very high frequency [35, 72]. Typically 5% of the items account for up to 25 % of all the

positive lookups. To take advantage of this we construct a small, but highly effective in-memory cache, to store these most frequent items and to avoid paying the long latency of accessing items on flash storage. To know which items to place in the cache, we use counters of each item. This cache is constructed by splitting the histogram into two histograms: one in-memory key-value cache, and the other an external key-value store with chunks stored on flash storage.

In our implementation, the external key-value store contains histogram data in the root-indexed format where the keys are roots, and the values are the 64 counters for the given root. Since the values are stored on flash and is smaller than the temporary flash storage needed by Sort-Reduce, the size of this representation does not matter as long as each item can be accessed by reading only a single chunk (the key-value store construction phase adds padding at the ends of chunks to ensure no items span two chunks). Since the data is stored in the exact form necessary for use, this format is actually more efficient for lookups and usage.

On the other hand, the in-memory key-value cache is stored on DRAM, and it needs to be as small as possible. To keep the size to a minimum, the histogram is stored using the same compact k-mer indexed format used by Sort-Reduce in the Count unit of SMUFIN-F, but the keys are the k-mer rotated left by a base so that the root is in the most significant position. This significantly reduces the size of the histogram at the cost of some added lookup complexity. The keys in the multilevel-index remain as roots to simplify the lookup process because, despite the k-mer-based format used for the histogram, lookups are still trying to get all the counters for a given root.

5.3.1 *Preventing Unnecessary Storage Accesses*

Due to the prevalence of sequencing errors in the input read data, we know that a considerable 30 % of the roots are unique and thus lead to negative lookups in the Label unit. We can further improve the performance of the histogram lookups by filtering negative lookups if it can be done efficiently. These filters are constructed per partition when the histogram key-value store and cache are constructed.

The first filter is a *band-pass filter* that removes lookup requests that fall outside the range between the lowest and highest keys seen during batch insertion. The band-pass filter is simple but effective, especially when the root set is partitioned. Since this is such a simple filter, it is beneficial to have a separate band-pass filter before the in-memory key-value cache as well as before the external key-value store. The second filter is a *Bloom filter*, which is populated during the construction of the key-value store. While Bloom filters are commonly used in many key-value stores, it is especially effective for us because the table is read-only, and there is no need to keep track of deleted items nor to rebuild the filter periodically. The Bloom filter does require multiple accesses into DRAM, but these are cheap compared to a lookup in the external key-value store (but expensive compared to the in-memory key-value cache). Therefore, the Bloom filter is only used on the external key-value store to avoid unnecessary flash accesses.

The overall structure of the in-memory key-value cache and the external key-value store can be seen in Figure 23.

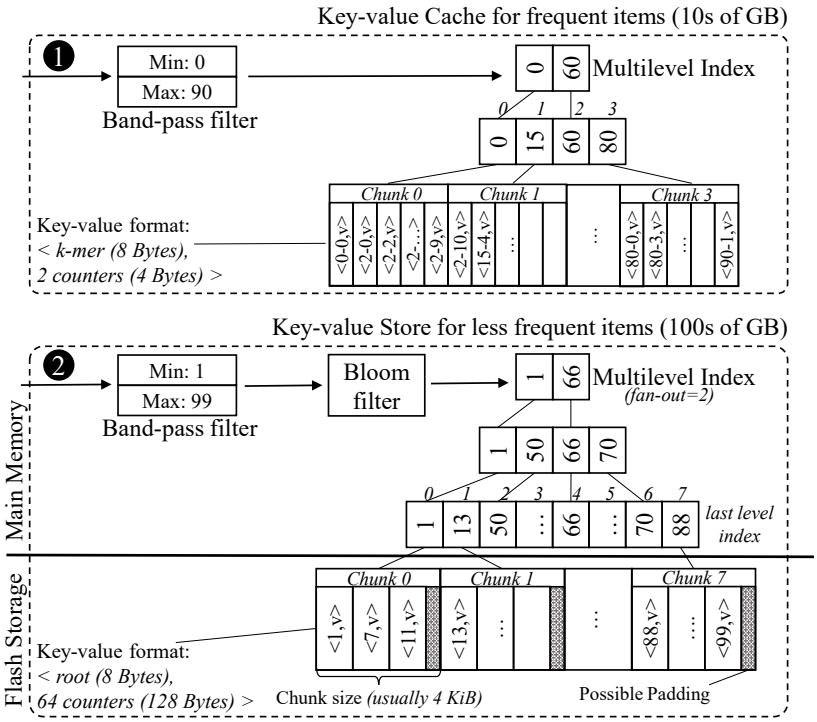


Figure 23: (§5.2-5.3) Example of the proposed key-value cache and store and their internal data structures. Each key is searched first in the key-value cache, and only if not found is searched in the key-value store. Differently from the key-value cache, the store places the items in the flash storage, and it uses a Bloom filter to stop negative lookups without the need to access the storage. The key-value format of the items is also different between the cache and the store. While the cache uses the space-efficient k-mer format, the key-value store uses the root format to limit each lookup to just one chuck.

5.4 EVALUATION AND RESULTS

In this section, we evaluate the performance and power efficiency of SMUFIN-F, and we demonstrate that it offers significant cost- and power-performance benefits against the baseline SMUFIN. We first compare the performance of the baseline version of SMUFIN running on various costly machines with 100s of GB of DRAM against SMUFIN-F on cheaper machines with Flash storage. In this comparison, we evaluate how our work improves energy efficiency, and subsequently, it also reduces the total cost of ownership for the full system. Next, we explore the performance of individual components of SMUFIN-F. We analyze the performance of our application-specific key-value store on PCIe NVMe storage devices and SATA SSDs, and compare it against RocksDB, a popular high-performance key-value store. We also present the effectiveness of our application-optimized in-memory cache.

5.4.1 *Evaluation Setup and Methodology*

For the performance comparison, we use the three systems described in Table 6. In "`MareNostrum 4`", we execute only the baseline version of SMUFIN. This system represents the existing pro-

Table 6: (§5.4) Experimental Setup

	MareNostrum 4	**FatNode3**	**Workstation** & **WorkstationNVMe**
CPU	2x Xeon Platinum 8160 24C @2.1 GHz (approx. 4700 USD)	2x Xeon E5-2680 v3 @2.50 GHz (approx. 3500 USD)	Core i7-8700K @3.70 GHz (approx. 350 USD)
# CPU Threads	2x 24	2x 24	12
DRAM	384 GB 12x 32 GB DDR4-2667 (approx. 4200 USD)	512 GB 16x 32 GB DDR4-2133 (approx. 3300 USD)	32 GB 4x 8 GB DDR4-2133 (approx. 200 USD)
Storage	GPFS	4x 1.5 TB Intel DC P3608 PCIe NVMe SSDs 4x 850 K random 4 KiB IOPS (approx. 2400 USD)	**Baseline Workstation** 4x 1 TB Samsung 860 EVO SATA-III SSDs 4x 98 K random 4 KiB IOPS (approx. 600 USD) **Optional for WorkstationNVMe** 2x 1TB Samsung 970 EVO Plus M.2 PCIe NVMe SSDs 2x 620 K random 4 KiB IOPS (approx. 500 USD)
Estimated Cost	8900 USD (Without storage)	7000 USD	1150 USD **or** 1650 USD

duction environment. In "FatNode3" we execute the baseline, using all the 512 GB of DRAM, and SMUFIN-F, capping its DRAM budget with cgroups to only 31 GB (leaving 1 GB out of our 32 GB budget for OS and other background software) and using four PCIe NVMe storage devices to store intermediate data. We use this system to show the impact of reducing the DRAM budget while using the same CPU. In "Workstation", a desktop-class machine with only 32 GB of DRAM and four SATA SSDs, we execute SMUFIN-F to demonstrate how it can run the full SMUFIN software pipeline on much cheaper machines. We also evaluate "WorkstationNVMe", which augments Workstation with two M.2 PCIe NVMe storage devices to accelerate random accesses in the Label unit. In both FatNode3 and Workstation, storage is organized into a software RAID-0 using the Linux md driver.

For each combination of system and implementation, we use different numbers of partitions in order to fit the working set in the available DRAM budged of each system. Unless differently specified, we report the aggregated time- and energy-to-solution metrics of all partitions executed sequentially. To minimize the overhead of context switches, the number of main CPU threads also vary from configuration to configuration to match the number of hyper-threaded cores that each CPU offers. In each execution, we process the same input genomes used in Chapter 3.4.

5.4.2 *SMUFIN vs SMUFIN-F*

Figure 24 compares the performance of the baseline SMUFIN implementation using 384 GB of DRAM and 48 threads on MareNostrum 4, and using 512 GB of DRAM and 48 threads on FatNode3, against the proposed SMUFIN-F using only 32 GB of DRAM and 48 threads on FatNode3, and 32 GB of DRAM and 12 threads on Workstation and WorkstationNVMe.

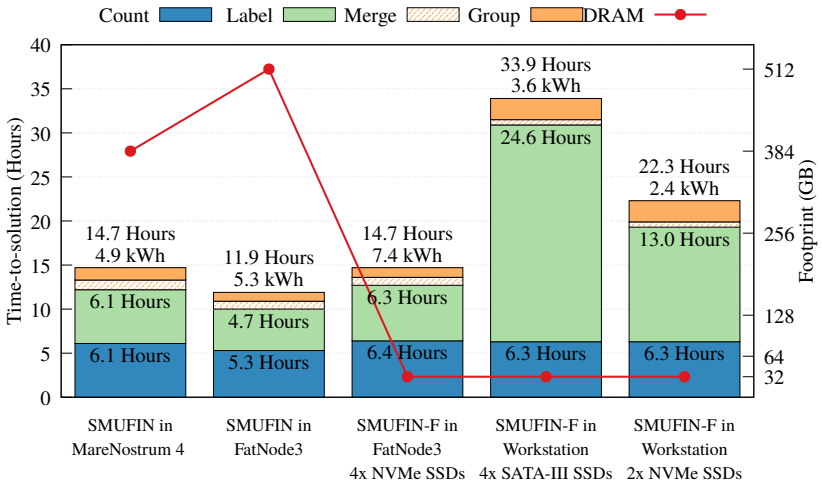


Figure 24: (§5.4.2) Time- and Energy-to-solution of the entire SMUFIN and SMUFIN-F pipeline on different hardware configurations.

Regarding the Count unit, Figure 24 shows that time-to-solution is not significantly degraded with the SMUFIN-F implementation that uses Sort-Reduce. Compared to `MareNostrum 4`, the memory footprint is reduced by 12x, with a loss of performance of only 4 %. Compared to the baseline on `FatNode3`, both SMUFIN-F executions are able to achieve around 70% of its performance while reducing the memory capacity by 16x. It is worth pointing out that the performance of the SMUFIN-F Count unit is similar between the costly `FatNode3` and the much cheaper `Workstation`. This is due to the fact that in the current version of Sort-Reduce, some portions of the last-level merge of the merge sort implementation is dependant on single-core performance, and while some parts of the Count unit benefits from more threads, in some parts `Workstation` benefits from a newer processor, running at a higher frequency. We are currently working to remove this limitation, after which the performance on `FatNode3` is expected to improve further.

Regarding the Label unit, the SMUFIN-F implementation using 32 GB of DRAM on `FatNode3` achieves 97% and 75% of the performance of the baseline SMUFIN running on `MareNostrum 4` and `FatNode3` using all available DRAM, respectively. On the other hand, when executed on `Workstation`, with much slower storage and a fourth of the CPU threads, the performance of SMUFIN-F Label unit is 5.2x slower than baseline SMUFIN running on `FatNode3`. System traces in Figure 25 indicate how the SATA III SSDs are the bottleneck, and how faster NVMe drives can significantly increase the read bandwidth and, thus, shrink the execution time. Indeed, by augmenting `Workstation` with two NVMe SSDs just for the Label unit, the performance of the Label unit improves significantly, reducing the Label performance gap to 2.7x. Moreover, in contrast to the Count unit, the Label unit scales with the number of CPU threads available. Thus, a CPU with more cores could reduce the execution time on `Workstation` even further. We analyze these two options, NVMe instead of SATA III SSDs and a CPU with more cores, in more detail in Section 5.4.2.2.

Regarding the Merge and Group units that were left unchanged, Figure 24 shows that running these units in systems with less DRAM and a slower CPU reduces performance. However, the loss is marginal compared to other units. The only noteworthy difference regarding these two

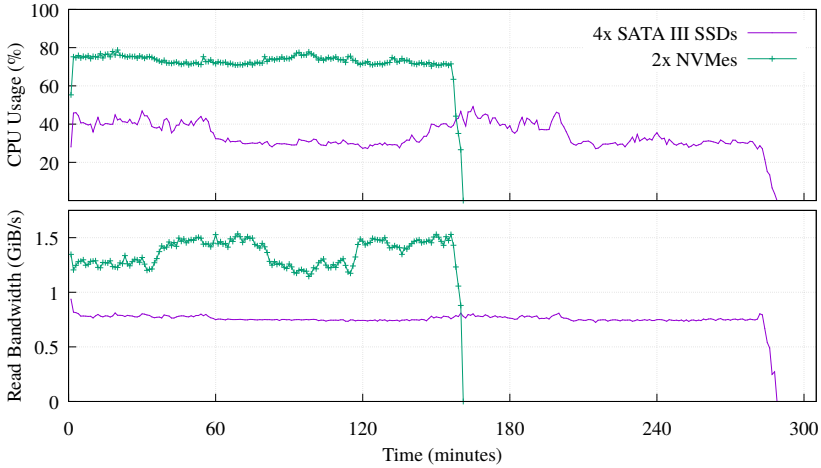


Figure 25: (§5.4.2) CPU usage and Read Bandwidth of the proposed Label unit, running the first partition out of five, on `Workstation` using 4x SATA III SSDs and 2x NVMes.

units is that the Group unit, which is I/O intensive, is slower with the reduced bandwidth of the SATA III SSDs.

In terms of overall performance, all configurations using only 32 GB of DRAM – using `FatNode3`, `Workstation`, and `WorkstationNVMe` – were able to achieve, respectively, 81%, 35%, and 53% of the performance of `FatNode3` with 512 GB of DRAM.

5.4.2.1 *Energy Consumption and Cost*

The true benefit of SMUFIN-F is in its power and cost reduction. Figure 24 also reports the energy-to-solution for each configuration. While SMUFIN-F on `WorkstationNVMe` is 1.87x slower than SMUFIN on `FatNode3`, it also consumes 2.2x less energy to completion. Even the slower `Workstation`, while being 2.85x slower, consumes 1.47x less energy to completion. During execution, the peak power for `FatNode3` was 549 W while `Workstation` and `WorkstationNVMe` only reached 120 W.

From a total cost of ownership point of view, while SMUFIN-F is 1.87x slower than the baseline running on a machine that costs 4x less, the cost differential means we can use two desktop-class machines similar to `WorkstationNVMe` to process two patients in parallel in less time than it would take on one `FatNode3`. Today, this would require an initial investment of less than half the price, approximately 3300 USD rather than 7000 USD. Besides, the two desktop-class nodes would consume only 45% of energy compared to `FatNode3`, 4.8 kWh against 10.6 kWh every two patients. This difference in power consumption also directly translates to power dissipation costs, such as server room cooling.

These efficiency results are summarized in Figure 26, which shows the capital cost, power consumption, and energy-to-solution of each system configuration, normalized to SMUFIN on `FatNode3`. This Figure shows that not only are SMUFIN-F-based solutions cheaper to buy, but also they consume significantly less energy per patient, which directly translates to the cost of operation.

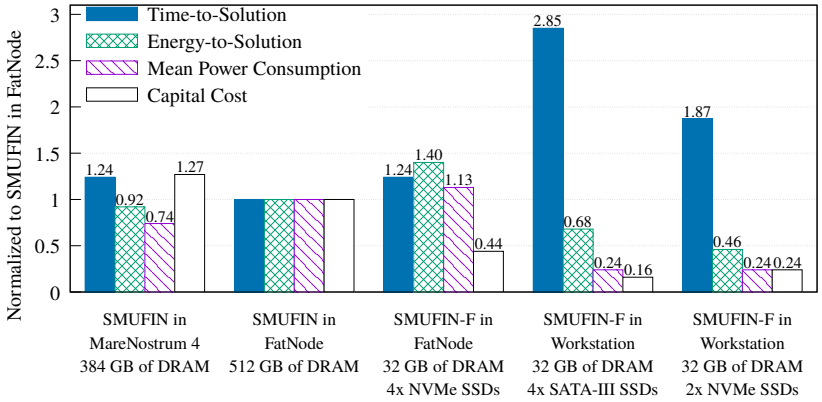


Figure 26: (§5.4.2) Normalized (to SMUFIN on `FatNode3`) time-to-solution, energy-to-solution, mean power consumption, and capital cost of SMUFIN and SMUFIN-F.

5.4.2.2 *Impact of Better Hardware*

In an attempt to lower the execution time of SMUFIN-F, one could choose to spend more on hardware. To understand the impact of hardware upgrades, we ran additional experiments to understand the performance impact that faster storage and CPU would have.

First, to prove that the Label unit is limited by the random read bandwidth, we upgraded the storage of `Workstation` with two NVMes drives and created `WorkstationNVMe`. As shown in Figure 25, this upgrade significantly increases the read bandwidth, from a steady 800 MiB/s to peaks of 1.5 GiB/s, and reduced the time-to-solution of the Label unit from 24.57 to 13.40 hours. This upgrade reduces the overall execution time from 33.9 hours to around 23, improving the energy-to-solution from 3.6 kWh to 2.4 kWh. As seen in Table 6, NVMe devices are more costly compared to SATA SSDs, and augmenting the machine with two NVMes adds a significant capital cost. However, NVMe prices are expected to get closer and closer to those of SATA SSDs, making it more and more appealing.

Second, to understand the impact of the CPU core count on the Label unit, we ran the Label unit on `FatNode3` with fewer threads to see how the time-to-solution degrades. Results showed that halving the number of threads doing the lookups from 40 to 20 already increased the execution time from 6.3 hours to around 8.9, a 40% increase. With only 10 threads, instead, the execution time rises to 16.5 hours, a 161% increase. This number somewhat matches the performance we obtained from `WorkstationNVMe`, which has a similar system configuration. Using fewer threads also leads to less pressure on the key-value store, which translates to a lower storage bandwidth usage. In fact, storage read bandwidth dropped from the range of 2.8-3.5 GiB/s to 1.8-2.5 GiB/s with 20 threads, and to 1-1.3 GiB/s with 10 threads.

These results indicate how a CPU with a higher core count could significantly reduce the execution time of the proposed Label unit, but only if the storage is fast enough to scale with it. Thus, storage should be the first component to be upgraded.

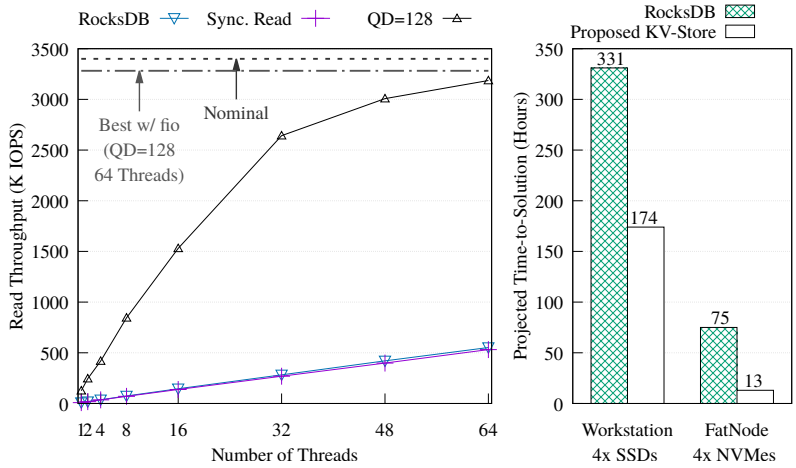


Figure 27: (§5.4.3) Throughput comparison of the proposed key-value store vs. RocksDB on `FatNode3` with 4 NVMes (left). Projection of the time-to-solution to process the required 149 billions of positive lookups – one per each non unique k-mer – during the Label unit at the maximum throughput achieved by both key-value stores on `Workstation` and `FatNode3` (right).

5.4.3 *SMUFIN-F External Key-Value Store Performance*

We performed two different evaluations of our external key-value store's performance. First, we show that our application-specific key-value store for the histogram is far superior to RocksDB key-value store. Second, we show how our key-value store achieves the maximum bandwidth of Linux's `libaio` asynchronous I/O library.

5.4.3.1 *Comparison Against RocksDB*

In order to evaluate the performance of our application-optimized key-value store, we compared against RocksDB version 5.15.10, a high-performance general-purpose key-value store. We ran the same benchmark with the exact same keys on both systems. To make the comparison as fair as possible and achieve the best performance from RocksDB, we first created the key-value store, adding the keys sequentially with compression disabled. Then, for each experiment, the database was opened read-only, auto with compaction disabled, and level0 filters and index blocks were pinned in cache.

Figure 27 offers a comparison of the throughput between RocksDB and the proposed key-value store. The figure shows how RocksDB performs very similarly to our synchronous read implementation. This is possibly due to the fact that in both cases, the read time is dominated by I/O latency due to the lack of access locality. In an effort to increase the throughput of RocksDB, we also considered its `MultiGet()` function to perform multiple lookups in batch. However, even with this function, the I/O still seems to be done in a synchronous fashion, and the performance does not get any better.

Figure 27 shows the projected execution time of the Label unit, taking into consideration only the time to perform the 149 billion positive lookups required for each patient, at the throughput obtained with the benchmark. This ignores the performance improvements obtained by our key-value cache or the Bloom filter in order to compare the raw performance difference. On `FatNode3`,

the Label unit running our optimized key-value store with asynchronous reads would require 13 hours to finish, while RocksDB would take 75 hours, demonstrating the benefit of our application-specific design. On `Workstation`, with one fourth of the CPU threads and slower SATA III SSDs, the Label unit would take hundreds of hours to finish. Finally, the projection shows how, to cope with the four orders of magnitude longer latency, data-intensive applications like genomics with 100s of billions of random lookups need asynchronous I/O on NVMes, and also a cache with SSDs.

5.4.3.2 *Saturating `libaio` Bandwidth with Key-Value Store*

We demonstrate the efficiency of our external key-value store without in-memory caching using a synthetic 100% random reads workload, which represents a workload similar to the lookups of the Label unit. The benchmark was executed using different numbers of CPU threads and using synchronous reads and asynchronous reads with different Queue Depths (QD). In each run, each thread performed 10 million positive lookups on a pre-built key-value store of 1 billion items (approximately 127 GB). The sizes of keys and values are set to 8 and 128 B respectively to match SMUFIN use case. We also compare the results against the nominal throughput and the best throughput of `libaio` measured using `fio` and direct I/O to emulate the same loads using random 4 KiB reads.

Figure 28a shows how, with four NVMes, the key-value store is able to achieve a maximum throughput of up to 3.1 M IOPS (around 12 GiB/s) with QD=128 and 64 threads, and how close this is to the performance obtained with `fio` and the nominal performance of the drives. The figure also displays how both queue depth and the number of CPU threads are important to achieve high throughput. Figure 28b shows how the performance increases as the combined queue depth increases, and how a reduced number of threads limits the benefit of larger combined queue sizes.

On the other hand, Figure 28c reveals how on `Workstation` both `fio` and the proposed key-value store can reach only around 60% of the nominal performance of the SATA III SSDs. In this system, a high number of threads is still essential for performance. However, as Figure 28d shows, increasing the queue depths only provides benefits up to a combined queue depth of 128 – which is exactly the combined queue depth of the SSDs used – and deeper queues show lower and stable performance. This result shows that in order to get optimal performance for `Workstation`, the combined queue depth should be sized to match the combined queue depth of the SSDs.

We also ran our key-value store implementation using synchronous I/O, and both of these Figures show how synchronous read delivers poor throughput, matching the throughput obtained with asynchronous reads at QD=1. While the benefit of asynchronous read is around 2x on SATA III SSDs, on NVMes, and with more threads, the improvement is around 6.5x; demonstrating how asynchronous I/O is key to take advantage of the high level of parallelism offered by NVMs.

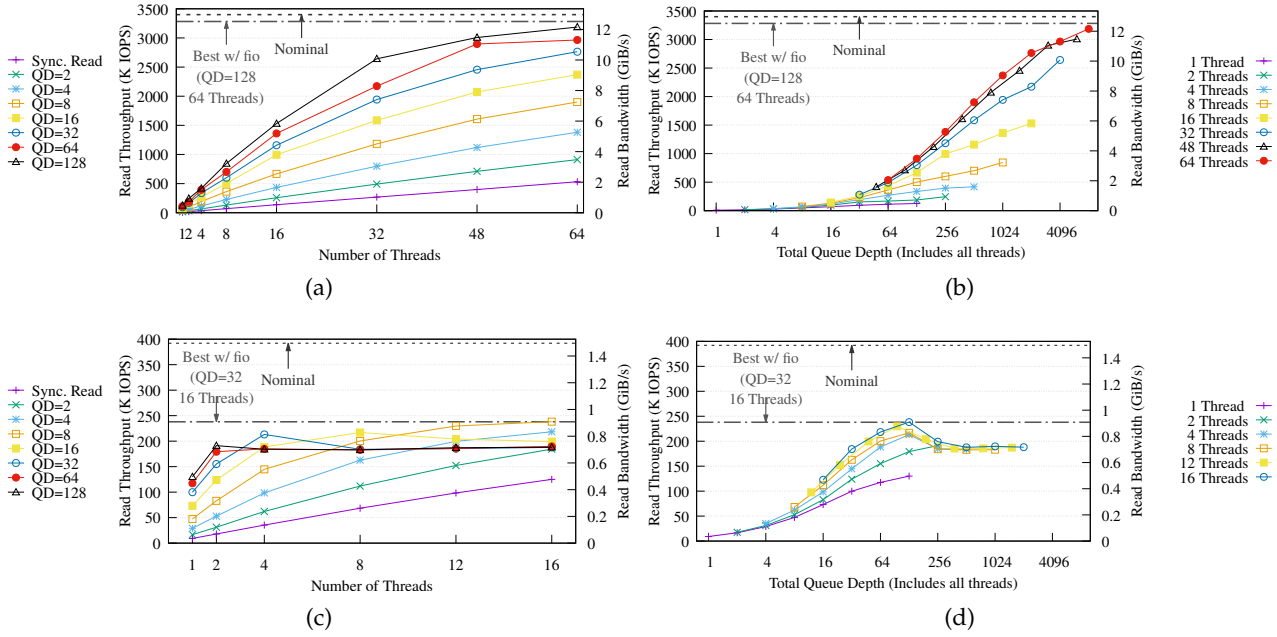
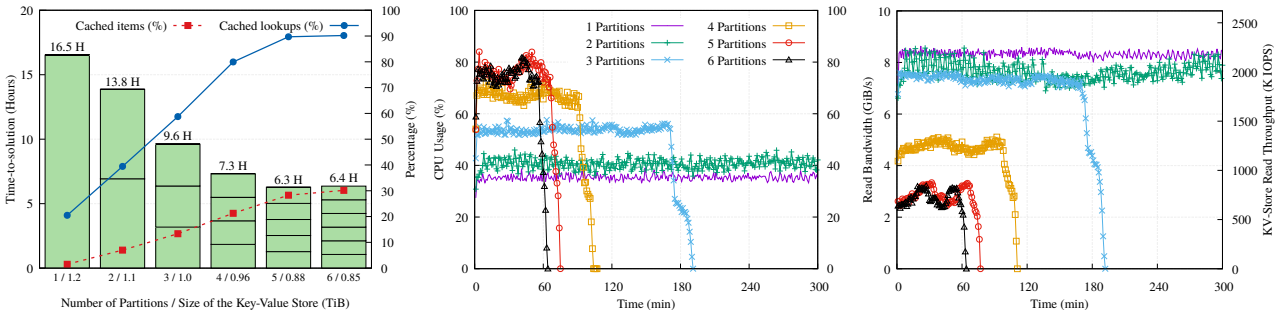


Figure 28: (§5.4.3.2) Throughput (a, c) and throughput versus total queue depth of the proposed KV-Store (b, d) on **`FatNode3` with 4x NVMes** (top) and on **`Workstation` with 4x SATA III SSDs** (bottom) running a benchmark with variable Queue Depths (QD) and numbers of threads. For each run, each thread executes 10 M randomized positive lookups on the same KV-Store populated with 1 B items (approx. 127 GB).

5.4.4 *SMUFIN-F Key-Value Cache Effectiveness*

As discussed in Section 5.3, we implemented a key-value cache that stores items that are predetermined to appear most frequently, in order to reduce the number of storage accesses and reduce execution time. Cache space can be used even more effectively when execution is divided into partitions. When the Label unit divides its input into multiple partitions and executes them in order, each partition execution will have exclusive access to the whole cache capacity, resulting in a larger total number of elements that can be serviced from the cache. Even though more partitions mean more full scans through the input reads, Figure 29a shows that the aggregated time-to-solution on `FatNode3` generally reduces as more partitions are used, by servicing more lookups from the cache. This trend continues up to five partitions, but with six partitions, the execution time stops improving.

The two lines in the figure reveal how the cache serves a considerable portion of the lookups (continuous, blue line) while storing only a minor fraction of the items (dashed, red line). The difference in the inclination between these two lines shows the effectiveness of the cache. The figure also shows that most high-frequency elements have already been cached with five partitions, as the percent of cached items improves only marginally after five partitions. This is reflected in the time-to-solution, and the performance improvements between four and five partitions are minor compared to the improvement between three and four partitions, which also explains why with six partitions, the execution time does not get shorter. This is, in fact, the turning point



(a) Time-to-Solution (b) CPU Usage (c) Storage Read

Figure 29: (§5.4.4) Time-to-solution and Cache Effectiveness of the proposed Label unit on `FatNode3` with 1 to 6 partitions(a). Traces of CPU usage (b) and Storage Read (c) of the first five hours of the first partition of each configuration.

where the extra work required to read and decompress the entire input data set once more is not compensated by more lookups served by the cache.

Figures 29b and Figure 29c show how with less partitions, the CPU load decreases and the read bandwidth instead increases. This indicates that the less partitions, the more is the time spent waiting for the read requests to the key-value store to complete.

Finally, we also ran the Label unit on `FatNode3` with five partitions and *synchronous read*, to evaluate the impact of synchronous reads once a good portion of the lookups is cached. Results showed that the time-to-solution increased of 2.57x, from the 6.3 hours to 16.17. With fewer partitions and, thus, fewer lookups served by the cache, the performance degradation is expected to increase even further, proving the importance of the cache.

5.5 RELATED WORK

5.5.1 *K-mer counting on Flash Storage*

We already compared SMUFIN k-mer counting to other methods that use storage in Section 3.5. Among these methods, Jellyfish [83] and KAnalyze [9] write sorted subsets of data to disk, and subsequently accumulate counts from each subset; DSK [108], KMC [40], and MSPKmer-Counter [91] split the k-mer space into partitions, or disk buckets, so that each partition can be loaded into memory and processed individually. SMUFIN-F uses Sort-Reduce, an optimized library that implements asynchronous I/O to take advantage of the bandwidth offered by flash storage, and a custom key-value store designed for random access of k-mer frequency counters. Our work goes beyond k-mer counting as it considers a complete genomics application, and enables reference-free somatic variant calling in a desktop-class machine. Beyond the particular application to SMUFIN, the work presented in this Chapter can also be applied to other genomics applications that rely on k-mers, including assembly-based variant calling and graph-based de-novo assembly.

5.5.2 *Flash Storage for Databases*

Flash memory can improve database and key-value store performance transparently by providing faster I/O compared to mechanical disks [77, 78, 10]. However, its true potential is achieved by using algorithms [75, 59, 29, 63] and data structures [1, 56, 112, 117] that are aware of underlying flash characteristics. Thanks to the high bandwidth and low latency of flash storage compared to magnetic disks, databases benefit from using them as a cache layer between memory and disk [62, 42, 61]. Many modern database systems in production have been designed to take advantage of the high bandwidth provided by flash storage. Such systems include Ceph [134], SILT [84], FlashDB [97]. ScyllaDB [115, 114] and uDepot [69] are key-value store built from the bottom-up to deliver the performance of fast NVM block-based devices, using a task-based runtime design to support asynchronous IO. uDepot is also extended to support the Storage Performance Development Kit [141].

One of the most prominent databases optimized for fast storage such as flash is RocksDB [123], which is a widely used open-source key-value store. MyRocks [122] is a MySQL storage engine that integrates with RocksDB and is used in the back-end at Facebook. To reduce the DRAM cache requirements of MyRocks servers, Facebook recently presented MyNVM [44] that extends MyRocks by a second-layer NVM cache.

Other key-value stores also focus on reducing the memory footprint of the indexes by doing multiple storage accesses to the storage, but that generally increases the latency of lookups. FAWN [7] is a distributed KV-store that uses an in-memory hash index to store only a fragment of the actual key. This reduces the memory requirement but introduces the chance of requiring two reads from flash. Similarly, FlashStore [37] stores compact key signatures instead of full keys to trade RAM usage with false positive flash reads. BloomStore [87] uses an index structure based on Bloom filters to store all indexes in flash storage efficiently. SkimpyStash [38] uses a hash table directory in DRAM to index key-value pairs stored on a log-structure on flash. To break the barrier of a flash pointer worth of DRAM overhead per key, it "moves" most of the pointers that locate each key-value pair from RAM to flash using chains of key-value entries. Here a lookup might translate to multiple flash lookups to traverse the chain. FPTree [100], investigate a hybrid storage-DRAM persistent and concurrent B+-Tree, that with NVM achieves similar performance to DRAM-based counterparts. In this design, leaf nodes are persisted on storage while inner nodes are placed in DRAM and rebuilt upon recovery. The Consistent and Durable Data Structures (CDDSs) B-Tree implementation [131] allows programmers to safely exploit the low-latency and non-volatile aspects of new memory technologies. It uses versioning to allow atomic updates without requiring logging. On a different approach, HiKV [138] exploits the distinct merits of hash index and B+-Tree index. HiKV builds and persists the hash index in NVM to retain its inherent ability of fast index searching. At the same time, it builds the B+-Tree index in DRAM to support range scans and avoids long NVM writes for maintaining consistency of the two indexes.

In contrast to most flash databases, SMUFIN-F reduces the size of the index leveraging the fact that the key-value entries are sorted and do not require additional reads, which would increase the lookup time. However, due to the usage of Bloom filters, it still expects and admits false positive reads.

5.6 FINAL CONSIDERATIONS

In this chapter, we explored a co-design aimed to reduce the DRAM footprint of a genomics application from 100s of GB to 32 GB, using NAND-flash storage as a replacement. Here we delved into the challenges of replacing main memory with flash storage. In particular, the need to overcome the four orders of magnitude longer latency and two orders of magnitude larger access granularity of flash storage over DRAM.

Results showed that thanks to Sort-Reduce and to the proposed key-value store and cache we reduced the system requirements for the entire SMUFIN method to the point that it can run to completion on an affordable desktop with 6-core i7 and 32 GB of memory at approximately 1/2 of the performance of the server machine with four times as many cores and 512 GB of DRAM. This desktop costs only 1/4 as much as the server and consumes 1/4 as much power, and requires only approximately 1/2 of energy per patient. As a result, a cluster of multiple desktop-class machines costs half as much compared to a cluster of servers and consumes half as much energy while maintaining a similar throughput.

6

CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS

In this thesis, we presented three complementary contributions, each representing a distinct iteration of hardware/software co-design aimed to lower the total cost of ownership of data-intensive genomics applications. As a use case, we used SMUFIN [95], a real-world method for detecting mutations acquired by an organism instead of inherited from a parent without full genome reconstruction or alignment against a reference. The SMUFIN method is supposed to run at scale on thousands of patients to identify relations between DNA mutations and cancerous tumors, and it has the potential to foster precision medicine. However, the original software implementation of the method demands plenty of computational resources, and it requires an infrastructure of enterprise-class servers, with entry and running costs that quickly become unmanageable with many patients and that most labs, clinics, and hospitals cannot afford; hindering the adoption of the method at a large scale. The three contributions of this thesis propose techniques, solutions, and difficulties to decrease the number of resources needed to run genomics applications like SMUFIN that analyses whole-human DNA samples. The final results show that it is possible to enable data-intensive genomics on desktop-class machines significantly reducing the total cost of ownership and enabling the adoption of these methods at scale and in the fields by researchers and physicians.

6.1.1 *Algorithmic Improvements to Reduce Workload Footprint*

The first contribution (C1) of this thesis is an initial hardware-software co-design to reduce the workload footprint of data-intensive applications to target fewer compute nodes, if not just one. We demonstrated how a modular structure that splits the application into different units could be used to prevent waste of resources, in particular for data-intensive applications whose DRAM requirement is considerable for a part of their execution. The same modular structure also supports dividing the data domain into partitions to target just one node where partitions can be executed sequentially. By targeting fewer nodes, this technique allows reducing capital expenditure and operational costs. To further minimize the operational costs related to energy consumption, we also explored techniques to reduce the main memory requirement to lower the number of partitions that impacts the time-to-solution. In particular, we focused on the k-mer frequency counting algorithm that is the part of SMUFIN with the highest main memory requirement – originally of 2 TB – and that dictates the number of compute nodes required to run the method. Besides, k-mer frequency counting is a technique also used by other genomics applications. Here, we investigated how a Chain of Bloom filters and a manual swapping technique to NVM allow ex-

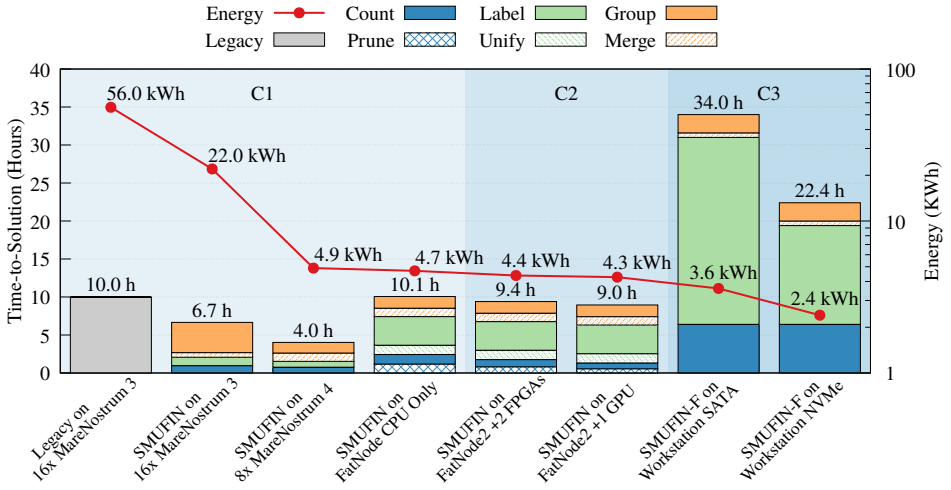


Figure 30: (§6.1) Time- and energy-to-solution of all SMUFIN versions on the different hardware configurations.

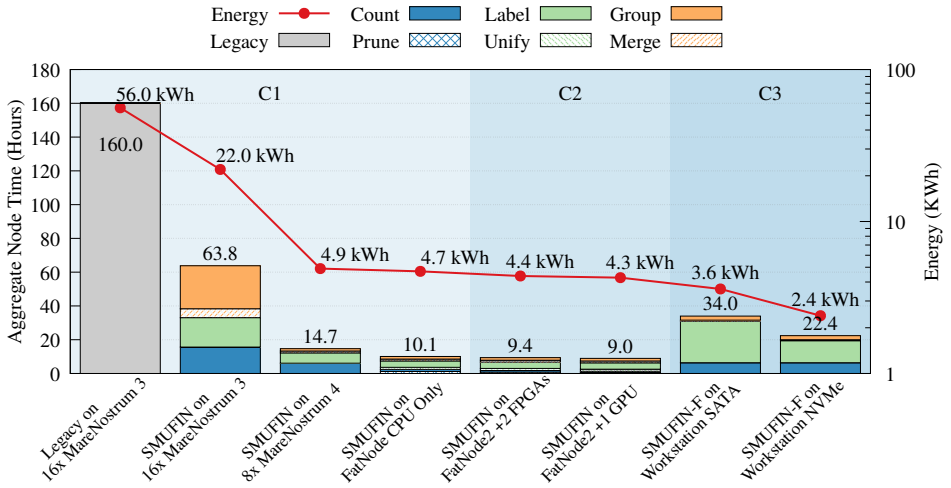


Figure 31: (§6.1) Aggregate node time and energy-to-solution of all SMUFIN versions on the different hardware configurations.

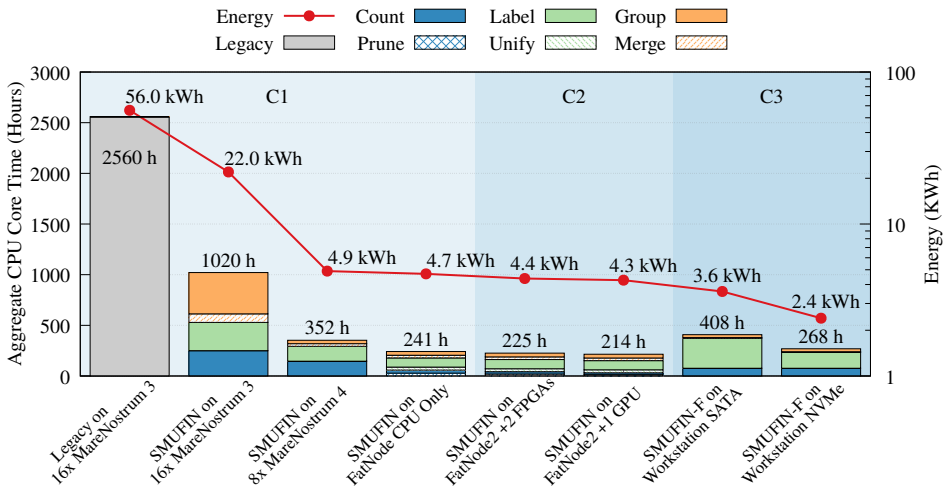


Figure 32: (§6.1) Aggregate CPU core time and energy-to-solution of all SMUFIN versions on the different hardware configurations.

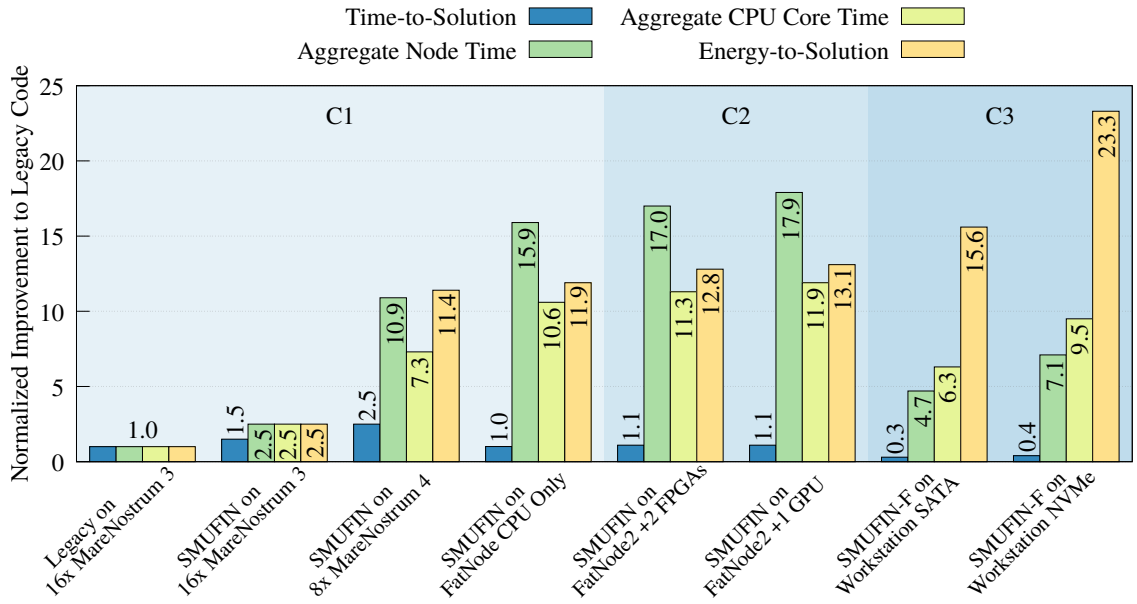


Figure 33: (§6.1) Relative improvement of time-to-solution, node time, aggregate CPU core time, and energy-to-solution of all SMUFIN versions on the different hardware configurations over the legacy code.

ecuting in just one node by lowering the memory footprint at the cost of some extra computation (Prune and Unify unit).

Results showed how, thanks to the techniques discussed in this first contribution, the workload footprint was reduced from 16 different nodes, each with 128 GB of main memory, to just one node with 512 GB of main memory and one non-volatile memory used as DRAM extension. Like Figures 30 to 33 show, the modular design proposed in 3.1 improved SMUFIN under all metrics when running in the same 16 `MareNostrum 3` nodes, reducing the time-to-solution of 1.5x, and the energy-to-solution of 2.5x, from 56 KWh to 22 KWh respectively. On `FatNode`, thanks to the solutions presented in 3.2 and 3.3, the energy-to-solution of the application was further reduced to 4.7 KWh/patient, an improvement of 11.91x compared to the Legacy code. Here, we were able to maintain a time-to-solution very close, only 1% slower, to the 16 nodes of `MareNostrum 4`, and since we are now using only one compute node, but with more CPU core, we also reduced the aggregate node time and the aggregate CPU core time of 15.9x and 10.6x, respectively. On the newer `MareNostrum 4`, thanks to more DRAM in each node, 384 GB instead than the 128 of `MareNostrum 3`, fewer partitions are required, reducing the time of both Count and Label unit. Here, the Group unit, which is I/O intensive, was also shortened. Improvement that is in part due to changes in the code outside the scope of this thesis and in part due to the use of the local NVMe drives in each node of `MareNostrum 4`. Finally, Figure 33 also shows that `FatNode` is still a valid option, that is able to outperform 8 nodes of the newer `MareNostrum 4`, except for the time-to-solution.

System traces of the execution proved that the k-mer counting algorithm is CPU bound for most of the time. These results suggested that accelerators could be leveraged to offload some of the compute-intensive work. This observation, plus the fact that k-mer counting is an algorithm also used in other methods, motivated the second contribution of this thesis.

6.1.2 *Offloading Computation to Accelerators*

The second contribution (C2) of this thesis is a hardware-software co-design to offload some compute-intensive operation of the k-mer counting algorithm to programmable accelerators to reduce the time- and energy-to-solution. This contribution was motivated by the results of the first contribution, and by the observation that data-intensive applications can easily become CPU-bound. We discussed how we offloaded the code to GPUs and how, the OpenCL GPU code needed to be redesigned to take advantage of FPGAs boards. Here we showed how, even if the code is written in OpenCL, due to the fundamental differences between the two hardware architectures, there is no performance portability; thus, the code needs to be redesigned to use FPGA-specific optimization. Besides, we presented how data-intensive applications could overcome the memory limits of accelerators by virtualizing the memory of accelerators on host DRAM. Obviously, this solution increases the PCIe traffic, and, like in the case with only one FPGA, it might become the main bottleneck of the application, particularly when using accelerators with mediocre PCIe capabilities. The offloaded coded involved naive offload of k-mer generation, a shuffling mechanism to shuffle k-mer so to minimize data movement and a technique to cooperatively build large Bloom filters using both CPU and accelerators at the same time.

Results showed that one FPGA falls short compared to the baseline CPU-only implementation. However, two FPGAs yield an improvement of 1.32x on both time- and energy-to-solution. Results also proved how the GPU outperforms the two FPGAs with an improvement of 1.77x on the energy-to-solution but, due to higher power consumption, of a lower 1.49x on the energy-to-solution. In fact, the power samples collected throughout the executions revealed how the FPGAs did not increase the power consumption of the system. Instead, in some moments, it 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 and how both kinds of accelerators need more on-board memory to permit the execution of similar data-intensive applications. Finally, we carried out a thorough analysis of the acceleration pipeline to uncover that in the faster execution, the GPU is not the bottleneck and that, instead, the CPU is the limiting factor. In particular, we identified as the main bottleneck the latency to main memory that must be paid to perform the random accesses to the Bloom filters and the hash tables.

These results suggest that the algorithm could benefit from building the main hash table in a manner that minimizes the random updates. Moreover, since this second contribution focused only on a portion of the SMUFIN method, like Figure 31 shows, the overall improvement is minimal, and it still requires plenty of power-hungry main memory, particularly for the Label unit. These observations, plus the fact that enterprise nodes require an infrastructure that most hospitals and clinics cannot afford, motivated the third contribution of this thesis.

6.1.3 *Memory footprint reduction through flash key-value store*

The third contribution (C3) of this thesis is a co-design that explores the challenge of reducing the DRAM footprint of a genomics application from 100s of GB to 32 GB, using NAND-flash

storage as a replacement. The work was motivated by the need for desktop-class machines that can perform in-situ genome analysis in hospitals and clinics, and by the observations that DRAM is facing scaling problems and its price has not been decreasing as significantly has before. To be able to run SMUFIN in a desktop-class machine composed of consumer-grade hardware, we modified the k-mer counting algorithm to take advantage of Sort-Reduce to efficiently build a histogram of k-mer frequency. Additionally, we designed a key-value store and cache tailored for the random read-only workload of the Label unit. Results and projections showed how asynchronous I/O and multiple threads are key to extract the performance of flash storage, and how important they are for workloads with billions of lookups such as genomics applications.

Results showed that we reduced the system requirements for the entire SMUFIN-F to the point that it can run to completion on an affordable desktop with 6-core i7 and 32 GB of memory, something that the previous SMUFIN implementations cannot. Like Figure 31 shows, on this desktop machine with NVMe, SMUFIN-F suffers a considerable slow-down when compared to all the other systems with plenty of CPU cores and 100s GB of DRAM. However, this is the cheapest and most efficient solution. Compared to `FatNode`, this desktop costs only 1/4 as much, consumes 1/4 as much power, and requires only approximately 1/2 of energy per patient (2.4 vs. 4.7 KWh). The other critical advantage of this solution is that it can be deployed in regular offices without the need for air-conditioned server rooms, further reducing the total cost of ownership and enabling its adoption even in small labs and clinics. As a result, a cluster of multiple desktop-class machines costs half as much compared to a cluster of servers and consumes half as much energy while maintaining a similar throughput. This work will help genomics researchers and ease the adoption of advanced methods and pipelines at the clinical level, which is key to enable precision medicine at scale.

6.1.4 *Overall Outcomes*

Although this thesis focused on a particular genomics application, many of its contributions revolved around the k-mer frequency counting algorithm and its vast memory footprint. Owing to this and to the extensive usage of such algorithm [139], most of the work done in this thesis can be applied to other genomics workloads. In particular, those that rely on whole human genomes or multiple genomes comparisons. Additionally, some of the outcomes of this thesis go beyond genomics workloads, and they also apply to other data-intensive workloads from other fields, scientific and not. These outcomes are recapitulated here:

- In the first contribution, we saw how scaling-out applications to meet their vast memory requirement can lead to significant wastage of CPU resources. Just because an application scales to multiple nodes, it does not mean that the resources are being used appropriately, wasting energy and CPU core time. This waste of resources exacerbates even more for data-intensive monolithic applications, making the co-designing for adaptability a crucial effort.

- In the first and third contribution, we showed how flash storage can be a viable, cost-effective alternative to big nodes with plenty of DRAM. However, this requires a co-design to adopt flash-specific optimizations so to leverage the parallelism of flash storage and to

hide the longer latency. In both contributions, we used NVMs to overcome the vast memory footprint of data structures memory extension. In the first contribution, an NVM was used as a faster swapping space that gave to the application the control on what and when to swap. Differently, in the third contribution, we used NVMs to store the key-value store that was used as an alternative to 100s of GB of DRAM. The swapping mechanism and the key-value store are packaged as libraries so that one could reuse them for other workloads.

- In the second contribution, we studied how the portability of heterogeneous frameworks such as OpenCL is limited, and it is not always guaranteed. Besides, performance portability between different architecture is a problem that highlights the need for co-design.
- Always in the second contribution, we showed how accelerators can be energy-efficient means for data-intensive workloads. Particularly for single-node deployments where the CPU is the main bottleneck. When choosing which accelerator should be targeted for data-intensive workloads, one should carefully consider all its components, with special attention to those critical components, such as the PCIe subsystem.
- In the second and third contribution, we saw how data-intensive workloads need asynchronous transfers to hide memory latency and to overlap computation and data transfers. In the second contribution, we used a double-buffering with asynchronous transfers to overlap transfers from and to the accelerators and computation on both CPU and accelerators. Differently, in the third contribution, we used asynchronous lookups, thus asynchronous transfers, to hide part of the extra latency to perform lookups on storage. Here we saw how this is critical to extract the throughput of new NVMe storage.
- Always in the third contribution, we demonstrated how workload-specific insights are crucial for co-design efforts. Here, the insights on SMUFIN drove and simplified the design of the key-value store that we created. These insights helped to create a key-value cache and an index with a memory footprint lower than a more traditional B+-tree and with filters to expedite negative lookups without storage requests.
- Finally, we proved how desktop-class appliances made with consumer-grade hardware can be a competitive energy-efficient alternative to cluster deployments. For data-intensive workloads, these appliances can overcome the low amount of main memory with fast storage. Obviously, this translates in longer execution times. However, power consumption could be reduced to the point that the solution is more energy-efficient that the faster cluster deployment.

6.2 FUTURE WORK

The work performed in this thesis faced the limitations of current hardware components. However, computer systems are in continuous evolution, and new emerging technologies offer interesting paths that could be explored as part of future work related to genomics workload, not necessarily related to SMUFIN.

Accelerators: In this thesis, we saw how the on-board memory of accelerators can become a problem to process data-intensive applications. Accelerators are getting faster and higher memory bandwidth, but they are slowly getting more on-board. To this end, manufacturers are starting to offer GPUs with 16 and also 32 GB of on-board memory. Here, high-end dual-GPU boards can offer up to 64 GB of on-board memory [5]; twice the amount of main memory of our `Workstation`! At the same time, FPGAs are getting a more competitive and more mature product. Here, thanks to the flexibility of FPGAs, manufacturers are offering very different options, from FPGA boards with up to 512 GB of on-board DDR4 [15] to Near Storage FPGAs [13, 14]. Similarly, this option is being explored with GPUs [4]. All these systems open up exciting opportunities. In the case of data-intensive applications, accelerators with so much memory enable the possibility to create completely offloaded key-value stores. In this direction, device-to-device memory-transfer offload with P2PDMA could also be used to build fully offloaded key-value store using more classical accelerators and NVMes. P2PDMA allows PCIe-to-PCIe transfers to bypass CPU memory using PCIe EP's memory (e.g., NVMe CMB, PCIe BAR) [111]. This requires P2P capable PCIe Root Complexes or PCIe Switches, but support from the Linux kernel and hardware starts to be available.

Finally, as a future work of our third contribution, one might want to try to offload computation to accelerators, even with a consumer workstation machine. This might dramatically increase the cost of the node, but a possible reduction of the energy-to-solution could justify the extra cost of the hardware.

Non-Volatile Memories: Non-volatile memories continue to get cheaper, bigger, affordable, and they are being deployed in large numbers. While operating systems and software start to leverage them properly, other new kinds of memory are emerging. An example is the 3DX-Point [136, 54]. This memory technology offers considerably lower latency than NAND-flash, and it is byte-addressable. 3DXPoint is being used in SSD, but it is also fast enough to be used as Non-Volatile RAM (NVRAM) in a DIMM package, namely NVDIMM. Thanks to its density, a single NVDIMM can reach up to 512 GB, it is an interesting technology for data-intensive applications. Besides, 3DXPoint NVDIMMs can be used in two different modes. The first mode, called Memory mode, is used to complement DRAM completely transparently to the application. Obviously, this comes at the cost of lower performance than DRAM, but data-intensive applications could benefit from this without the need to be changed. The second mode to use these 3DXPoint NVDIMMs, called App Mode, allows applications to see two distinct memory pools, DRAM DIMMs and NVDIMMs. In this mode, applications can use the NVDIMMs using SNIA's Persistent Memory Programming Model [120] and opening new research opportunities that are worth exploring. The impact on data-intensive applications could be tremendous, and NVDIMMs could be used to store the main data structures using just one node with up 3 TB of DRAM. In the particular case of SMUFIN, NVDIMMs could be used to build the root histogram without the need of partitions, directly improving the Count and Label unit.

Whatever future co-design efforts, it is clear that the more the hardware specializes and diversifies, the more critical hardware/software co-design becomes.

ACKNOWLEDGEMENTS

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

BIBLIOGRAPHY

[1] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *Proc. VLDB Endow.*, 2(1):361–372, August 2009. ISSN 2150-8097. doi: 10.14778/1687627.1687669. URL https://doi.org/10.14778/1687627.1687669.

[2] N. Ahmed, V. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars. Heterogeneous hardware/software acceleration of the BWA-MEM DNA alignment algorithm. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 240–246, November 2015. doi: 10.1109/ICCAD.2015.7372576.

[3] Nauman Ahmed. nahmedraja/GASAL2, July 2019. URL https://github.com/nahmedraja/GASAL2. original-date: 2018-02-22T17:35:43Z.

[4] AMD. Radeon Pro SSG GPU - Solid State Graphics | AMD, 2019. URL https://www.amd.com/en/products/professional-graphics/radeon-pro-ssg.

[5] AMD. Radeon Pro Vega II Graphics | AMD, 2019. URL https://www.amd.com/en/graphics/workstations-radeon-pro-vega-II.

[6] Ashok Anand, Aaron Gember-Jacobson, Collin Engstrom, and Aditya Akella. Design Patterns for Tunable and Efficient SSD-based Indexes. In *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '14, pages 149–160, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2839-5. doi: 10.1145/2658260.2658270. URL http://doi.acm.org/10.1145/2658260.2658270.

[7] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM. ISBN 9781605587523. doi: 10.1145/1629575.1629577. URL http://doi.acm.org/10.1145/1629575.1629577. event-place: Big Sky, Montana, USA.

[8] O. J. Arndt, F. D. Trager, T. Moss, and H. Blume. Portable Implementation of Advanced Driver-Assistance Algorithms on Heterogeneous Architectures. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 6–17, May 2017. doi: 10.1109/IPDPSW.2017.100.

[9] Peter Audano and Fredrik Vannberg. KAnalyze: a fast versatile pipelined K-mer toolkit. *Bioinformatics*, 30(14):2070–2072, July 2014. ISSN 1367-4803. doi: 10.1093/bioinformatics/btu152. URL https://academic.oup.com/bioinformatics/article/30/14/2070/2390485.

[10] Daniel Bausch, Ilia Petrov, and Alejandro Buchmann. On the performance of database query processing algorithms on flash solid state disks. In *2011 22nd International Workshop on Database and Expert Systems Applications*, pages 139–144. IEEE, 2011.

[11] Gaetan Benoit. Simka: fast kmer-based method for estimating the similarity between numerous metagenomic datasets. *JOBIM 2015*, October 2015. URL https://hal.inria.fr/hal-01231795.

[12] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. A Design for High-performance Flash Disks. *SIGOPS Oper. Syst. Rev.*, 41(2):88–93, April 2007. ISSN 0163-5980. doi: 10.1145/1243418.1243429. URL http://doi.acm.org/10.1145/1243418.1243429.

[13] BittWare. 250-SoC, 2019. URL https://www.bittware.com/fpga/250-soc/.

[14] BittWare. 250s+, 2019. URL https://www.bittware.com/fpga/250s/.

[15] BittWare. XUP-P3r PCIe FPGA Board, 2019. URL https://www.bittware.com/fpga/xup-p3r/.

[16] M. Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007. ISSN 1098-4232. doi: 10.1109/N-SSC.2007.4785534.

[17] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Commun. ACM*, 54 (5):67–77, May 2011. ISSN 0001-0782. doi: 10.1145/1941487.1941507. URL http://doi.acm.org/10.1145/1941487.1941507.

[18] Andrew J. Bromage and Thomas C. Conway. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 01 2011. ISSN 1367-4803. doi: 10.1093/bioinformatics/btq697. URL https://doi.org/10.1093/bioinformatics/btq697.

[19] A. Bustamam, E. D. Ulul, H. F. A. Hura, and T. Siswantining. Implementation of hierarchical clustering using k-mer sparse matrix to analyze MERS-CoV genetic relationship. *AIP Conference Proceedings*, 1862(1):030142, July 2017. ISSN 0094-243X. doi: 10.1063/1.4991246. URL https://aip.scitation.org/doi/abs/10.1063/1.4991246.

[20] 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 (HPCC)*, pages 434–441, December 2017. doi: 10.1109/HPCC-SmartCity-DSS.2017.57.

[21] Nicola Cadenelli, Zoran Jaksić, Jordà Polo, and David Carrera. Considerations in using OpenCL on GPUs and FPGAs for throughput-oriented genomics workloads. *Future Generation Computer Systems*, 94:148 – 159, 2019. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2018.11.028. URL http://www.sciencedirect.com/science/article/pii/S0167739X18314183.

[22] David Carrera Perez, Jordà Polo, Nicola Cadenelli, David Torrents Arenales, and Mercè Planas. A Computer-Implemented and Reference-Free Method for Identifying Variants in Nucleic Acid Sequences, jan 2018. URL https://patentscope.wipo.int/search/en/detail.jsf?docId=WO2018007034.

[23] Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: 10.1109/SC.2010.56. URL http://dx.doi.org/10.1109/SC.2010.56.

[24] Genetic Science Learning Center. Why the time is right, 2016. URL https://learn.genetics.utah.edu/content/precision/time/.

[25] Kevin K. Chang. Understanding and Improving the Latency of DRAM-Based Memory Systems. *arXiv:1712.08304 [cs]*, December 2017. URL http://arxiv.org/abs/1712.08304. arXiv: 1712.08304.

[26] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS '16, pages 323–336, New York, NY, USA, 2016. ACM. ISBN 9781450342667. doi: 10.1145/2896377.2901453. URL http://doi.acm.org/10.1145/2896377.2901453. event-place: Antibes Juan-les-Pins, France.

[27] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 266–277, February 2011. doi: 10.1109/HPCA.2011.5749735.

[28] Ken Chen, John W Wallis, Michael D McLellan, David E Larson, Joelle M Kalicki, Craig S Pohl, Sean D McGrath, Michael C Wendl, Qunyuan Zhang, Devin P Locke, et al. Breakdancer: an algorithm for high-resolution mapping of genomic structural variation. *Nature methods*, 6(9):677, 2009.

[29] Zhibo Chen and Carlos Ordonez. Optimizing olap cube processing on solid state drives. In *Proceedings of the Sixteenth International Workshop on Data Warehousing and OLAP*, DOLAP '13, pages 79–84, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2412-0. doi: 10.1145/2513190.2513197. URL http://doi.acm.org/10.1145/2513190.2513197.

[30] K. Cheung, K. Tong, K. Lee, and K. Leung. Classification of RNAs with pseudoknots using k-mer occurrences count as attributes. In *13th IEEE International Conference on BioInformatics and BioEngineering*, pages 1–4, November 2013. doi: 10.1109/BIBE.2013.6701575.

[31] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, 8(1):22, September 2013. ISSN 1748-7188. doi: 10.1186/1748-7188-8-22. URL https://doi.org/10.1186/1748-7188-8-22.

[32] Kristian Cibulskis, Michael S Lawrence, Scott L Carter, Andrey Sivachenko, David Jaffe, Carrie Sougnez, Stacey Gabriel, Matthew Meyerson, Eric S Lander, and Gad Getz. Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples. *Nature biotechnology*, 31(3):213, 2013.

[33] Phillip E. C. Compeau, Pavel A. Pevzner, and Glenn Tesler. Why are de Bruijn graphs useful for genome assembly? *Nature biotechnology*, 29(11):987–991, November 2011. ISSN 1087-0156. doi: 10.1038/nbt.2023. URL https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5531759/.

[34] S. Datta, P. Beeraka, and R. Sass. RC-BLASTn: Implementation and Evaluation of the BLASTn Scan Function. In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pages 88–95, April 2009. doi: 10.1109/FCCM.2009.15.

[35] A. P. Jason de Koning, Wanjun Gu, Todd A. Castoe, Mark A. Batzer, and David D. Pollock. Repetitive Elements May Comprise Over Two-Thirds of the Human Genome. *PLoS Genetics*, 7(12), December 2011. ISSN 1553-7390. doi: 10.1371/journal.pgen.1002384. URL https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3228813/.

[36] Dick de Ridder, Gabino Sanchez-Perez, Guusje Bonnema, Ke Lin, and Sandra Smit. Making the difference: integrating structural variation detection tools. *Briefings in Bioinformatics*, 16 (5):852–864, 12 2014. ISSN 1467-5463. doi: 10.1093/bib/bbu047. URL https://doi.org/10.1093/bib/bbu047.

[37] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-value Store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1921015. URL http://dx.doi.org/10.14778/1920841.1921015.

[38] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 25–36, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989327. URL http://doi.acm.org.recursos.biblioteca.upc.edu/10.1145/1989323.1989327.

[39] Jacob F Degner, John C Marioni, Athma A Pai, Joseph K Pickrell, Everlyne Nkadori, Yoav Gilad, and Jonathan K Pritchard. Effect of read-mapping biases on detecting allele-specific expression from rna-sequencing data. *Bioinformatics*, 25(24):3207–3212, 2009.

[40] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, May 2015. ISSN 1367-4803. doi: 10.1093/bioinformatics/btv022. URL https://academic.oup.com/bioinformatics/article/31/10/1569/177467.

[41] NVIDIA Developer. Clara Genomics, May 2019. URL https://developer.nvidia.com/Clara-Genomics.

[42] Jaeyoung Do, Donghui Zhang, Jignesh M. Patel, David J. DeWitt, Jeffrey F. Naughton, and Alan Halverson. Turbocharging dbms buffer pool using ssds. In *Proceedings of the 2011 ACM*

SIGMOD International Conference on Management of Data, SIGMOD '11, pages 1113–1124, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989442. URL http://doi.acm.org/10.1145/1989323.1989442.

[43] S. Eilert, M. Leinwander, and G. Crisenza. Phase Change Memory: A New Memory Enables New Memory Usage Models. In *2009 IEEE International Memory Workshop*, pages 1–2, May 2009. doi: 10.1109/IMW.2009.5090604.

[44] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 42:1–42:13, New York, NY, USA, 2018. ACM. ISBN 9781450355841. doi: 10.1145/3190508.3190524. URL http://doi.acm.org/10.1145/3190508.3190524. event-place: Porto, Portugal.

[45] Nature Publishing Group. Human genome at ten: The sequence explosion. *Nature*, 464 (7289):670–671, April 2010. ISSN 0028-0836, 1476-4687. doi: 10.1038/464670a. URL http://www.nature.com/articles/464670a.

[46] E. Houtgast, V. M. Sima, and 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)*, pages 492–496, October 2017. doi: 10.1109/BIBE.2017.000-6.

[47] Ernst Joachim Houtgast, Vlad-Mihai Sima, Koen Bertels, and Zaid Al-Ars. Hardware acceleration of BWA-MEM genomic short read mapping for longer read lengths. *Computational Biology and Chemistry*, 75:54–64, August 2018. ISSN 1476-9271. doi: 10.1016/j.compbiolchem.2018.03.024. URL http://www.sciencedirect.com/science/article/pii/S1476927118301555.

[48] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 10:1–10:9, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-623-6. doi: 10.1145/1534530.1534544. URL http://doi.acm.org/10.1145/1534530.1534544.

[49] Illumia. Sequencing Coverage for NGS Experiments. https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/coverage.html, 2019. URL https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/coverage.html.

[50] National Human Genome Research Institute. The Cost of Sequencing a Human Genome, 2019. URL https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost.

[51] Intel. IntelFPGA Channelizer Design Example, 2014. URL https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/channelizer.html.

[52] Intel. Accelerating Genomics Research with OpenCL and FPGAs, 2017. URL https://www.intel.com/content/www/us/en/healthcare-it/solutions/documents/genomics-research-with-opencl-and-fpgas-paper.html.

[53] Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. *De novo* assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44(2):226–232, February 2012. ISSN 1546-1718. doi: 10.1038/ng.1028. URL https://www.nature.com/articles/ng.1028.

[54] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv:1903.05714 [cs]*, March 2019. URL http://arxiv.org/abs/1903.05714. arXiv: 1903.05714.

[55] William R. Jeck, Josephine A. Reinhardt, David A. Baltrus, Matthew T. Hickenbotham, Vincent Magrini, Elaine R. Mardis, Jeffery L. Dangl, and Corbin D. Jones. Extending assembly of short DNA sequences to handle error. *Bioinformatics (Oxford, England)*, 23(21):2942–2944, November 2007. ISSN 1367-4811. doi: 10.1093/bioinformatics/btm451.

[56] Peiquan Jin, Chengcheng Yang, Christian S. Jensen, Puyuan Yang, and Lihua Yue. Read/write-optimized tree indexing for solid-state drives. *The VLDB Journal*, 25(5):695–717, October 2016. ISSN 1066-8888. doi: 10.1007/s00778-015-0406-1. URL http://dx.doi.org/10.1007/s00778-015-0406-1.

[57] Sang Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Bigsparse: High-performance external graph analytics. *CoRR*, abs/1710.07736, 2017.

[58] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. GraFboost: Using Accelerated Flash Storage for External Graph Analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 411–424, Piscataway, NJ, USA, 2018. IEEE Press. ISBN 9781538659847. doi: 10.1109/ISCA.2018.00042. URL https://doi.org/10.1109/ISCA.2018.00042. event-place: Los Angeles, California.

[59] Myoungsoo Jung, Ramya Prabhakar, and Mahmut Taylan Kandemir. Taking garbage collection overheads off the critical path in ssds. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 164–186, New York, NY, USA, 2012. Springer-Verlag New York, Inc. ISBN 978-3-642-35169-3. URL http://dl.acm.org/citation.cfm?id=2442626.2442638.

[60] Wolfgang Kaisers, Holger Schwender, and Heiner Schaal. Hierarchical clustering of DNA k-mer counts in RNA-seq fastq files reveals batch effects. *arXiv:1405.0114 [q-bio]*, May 2014. URL http://arxiv.org/abs/1405.0114. arXiv: 1405.0114.

[61] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. Flash-based extended cache for higher throughput and faster recovery. *Proc. VLDB Endow.*, 5(11):1615–1626, July 2012. ISSN 2150-8097. doi: 10.14778/2350229.2350274. URL http://dx.doi.org/10.14778/2350229.2350274.

[62] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. Flash as cache extension for online transactional workloads. *The VLDB Journal*, 25(5):673–694, October 2016. ISSN 1066-8888. doi: 10.1007/s00778-015-0414-1. URL http://dx.doi.org/10.1007/s00778-015-0414-1.

[63] Yaron Kanza and Hadas Yaari. External sorting on flash storage: Reducing cell wearing and increasing efficiency by avoiding intermediate writes. *The VLDB Journal*, 25(4):495–518, August 2016. ISSN 1066-8888. doi: 10.1007/s00778-016-0426-5. URL http://dx.doi.org/10.1007/s00778-016-0426-5.

[64] Nagarajan Kathiresan, Rashid Al-Ali, Puthen Jithesh, Ganesan Narayanasamy, and Zaid Al-Ars. Porting and Benchmarking of BWAKIT Pipeline on OpenPOWER Architecture. In Rio Yokota, MichÃšle Weiland, John Shalf, and Sadaf Alam, editors, *High Performance Computing*, Lecture Notes in Computer Science, pages 402–410. Springer International Publishing, 2018. ISBN 9783030024659.

[65] David R Kelley, Michael C Schatz, and Steven L Salzberg. Quake: quality-aware detection and correction of sequencing errors. *Genome biology*, 11(11):R116, 2010.

[66] kernel.org. Intel P-State driver - Linux Kernel documentation, 2019.

[67] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-Aware I/O Management for Solid State Disks (SSDs). *IEEE Trans. Comput.*, 61(5):636–649, May 2012. ISSN 0018-9340. doi: 10.1109/TC.2011.76. URL http://dx.doi.org/10.1109/TC.2011.76.

[68] Petr Klus, Simon Lam, Dag Lyberg, Ming Sin Cheung, Graham Pullan, Ian McFarlane, Giles SH Yeo, and Brian YH Lam. BarraCUDA - a fast short read sequence aligner using graphics processing units. *BMC Research Notes*, 5(1):27, January 2012. ISSN 1756-0500. doi: 10.1186/1756-0500-5-27. URL https://doi.org/10.1186/1756-0500-5-27.

[69] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with udepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, Boston, MA, 2019. USENIX Association. ISBN 978-1-931971-48-5. URL https://www.usenix.org/conference/fast19/presentation/kourtis.

[70] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, M. Gyang, and J. Lancaster. Biosequence similarity search on the Mercury system. In *Proceedings. 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004.*, pages 365–375, September 2004. doi: 10.1109/ASAP.2004.1342485.

[71] David Laehnemann, Arndt Borkhardt, and Alice Carolyn McHardy. Denoising DNA deep sequencing data-high-throughput sequencing errors and their correction. *Briefings in Bioinformatics*, 17(1):154–179, January 2016. ISSN 1467-5463. doi: 10.1093/bib/bbv029.

[72] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, February 2001. ISSN 0028-0836. doi: 10.1038/35057062.

[73] William B. Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. Improving CUDA DNA Analysis Software with Genetic Programming. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1063–1070, New York, NY, USA, 2015. ACM. ISBN 9781450334723. doi: 10.1145/2739480.2754652. URL http://doi.acm.org/10.1145/2739480.2754652. event-place: Madrid, Spain.

[74] Ilkka Lappalainen, Jeff Almeida-King, Vasudev Kumanduri, Alexander Senf, John Dylan Spalding, Saif ur Rehman, Gary Saunders, Jag Kandasamy, Mario Caccamo, Rasko Leinonen, Brendan Vaughan, Thomas Laurent, Francis Rowland, Pablo Marin-Garcia, Jonathan Barker, Petteri Jokinen, Angel Carreno Torres, Jordi Rambla de Argila, Oscar Martinez Llobet, Ignacio Medina, Marc Sitges Puy, Mario Alberich, Sabela de la Torre, Arcadi Navarro, Justin Paschall, and Paul Flicek. The European Genome-phenome Archive of human data consented for biomedical research. *Nature Genetics*, 47:692–695, June 2015. ISSN 1546-1718. doi: 10.1038/ng.3312. URL https://www.nature.com/articles/ng.3312.

[75] Eun-Mi Lee, Sang-Won Lee, and Sangwon Park. Optimizing index scans on flash memory ssds. *SIGMOD Rec.*, 40(4):5–10, January 2012. ISSN 0163-5808. doi: 10.1145/2094114.2094116. URL http://doi.acm.org/10.1145/2094114.2094116.

[76] Eunji Lee, Hyokyung Bahn, Seunghoon Yoo, and Sam H. Noh. Empirical Study of NVM Storage: An Operating System's Perspective and Implications. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, MASCOTS '14, pages 405–410, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-5610-4. doi: 10.1109/MASCOTS.2014.56. URL http://dx.doi.org/10.1109/MASCOTS.2014.56.

[77] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1075–1086, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376723. URL http://doi.acm.org/10.1145/1376616.1376723.

[78] Sang-Won Lee, Bongki Moon, and Chanik Park. Advances in flash memory ssd technology for enterprise database applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 863–870, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559937. URL http://doi.acm.org/10.1145/1559845.1559937.

[79] Richard M Leggett and Dan MacLean. Reference-free SNP detection: dealing with the data deluge. *BMC Genomics*, 15(Suppl 4):S10, May 2014. ISSN 1471-2164. doi: 10.1186/1471-2164-15-S4-S10. URL https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4083407/.

[80] Richard M. Leggett, Ricardo H. Ramirez-Gonzalez, Walter Verweij, Cintia G. Kawashima, Zamin Iqbal, Jonathan D. G. Jones, Mario Caccamo, and Daniel MacLean. Identifying and classifying trait linked polymorphisms in non-reference species by walking coloured de bruijn graphs, 03 2013. URL https://doi.org/10.1371/journal.pone.0060058.

[81] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, July 2009. ISSN 1367-4803. doi: 10.1093/bioinformatics/btp324. URL https://academic.oup.com/bioinformatics/article/25/14/1754/225615.

[82] Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, 20(2):265–272, 2010.

[83] Yang Li and XifengYan. MSPKmerCounter: A Fast and Memory Efficient Approach for K-mer Counting. *arXiv:1505.06550 [cs, q-bio]*, May 2015. URL http://arxiv.org/abs/1505.06550. arXiv: 1505.06550.

[84] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043558. URL http://doi.acm.org/10.1145/2043556.2043558.

[85] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, April 2012. doi: 10.1109/MSST.2012.6232369.

[86] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI. *BMC Bioinformatics*, 12:85, 2011. ISSN 1471-2105. doi: 10.1186/1471-2105-12-85.

[87] Guanlin Lu, Young Jin Nam, and David HC Du. Bloomstore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2012.

[88] L. Ma, R. D. Chamberlain, J. D. Buhler, and M. A. Franklin. Bloom Filter Performance on Graphics Engines. In *2011 International Conference on Parallel Processing*, pages 522–531, September 2011. doi: 10.1109/ICPP.2011.27.

[89] Martin Madera, Ryan Calmus, Grant Thiltgen, Kevin Karplus, and Julian Gough. Improving protein secondary structure prediction using a simple k-mer model. *Bioinformatics*, 26(5):596–602, March 2010. ISSN 1367-4803. doi: 10.1093/bioinformatics/btq020. URL https://academic.oup.com/bioinformatics/article/26/5/596/213334.

[90] Nicolas Maillet, Claire Lemaitre, Rayan Chikhi, Dominique Lavenier, and Pierre Peterlongo. Compareads: comparing huge metagenomic experiments. *BMC Bioinformatics*, 13(19):S10, December 2012. ISSN 1471-2105. doi: 10.1186/1471-2105-13-S19-S10. URL https://doi.org/10.1186/1471-2105-13-S19-S10.

[91] Guillaume MarÃ§ais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, March 2011. ISSN 1367-4803. doi: 10.1093/bioinformatics/btr011. URL https://academic.oup.com/bioinformatics/article/27/6/764/234905.

[92] 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)*, pages 203–210, April 2017. doi: 10.1109/FCCM.2017.23.

[93] Pall Melsted and Jonathan K Pritchard. Efficient counting of k-mers in dna sequences using a bloom filter. *BMC bioinformatics*, 12(1):333, 2011.

[94] D. Molka, D. Hackenberg, R. Schone, and W. E. Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *2015 44th International Conference on Parallel Processing*, pages 739–748, September 2015. doi: 10.1109/ICPP.2015.83.

[95] Valentí Moncunill, Santi Gonzalez, Sílvia Beà, Lise O Andrieux, Itziar Salaverria, Cristina Royo, Laura Martinez, Montserrat Puiggròs, Maia Segura-Wang, Adrian M Stütz, et al. Comprehensive characterization of complex structural variations in cancer by directly comparing genome sequence reads. *Nature biotechnology*, 32(11):1106–1112, 2014.

[96] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno. Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis. *IEEE Access*, 5:2747–2762, 2017. doi: 10.1109/ACCESS.2017.2671881.

[97] Suman Nath and Aman Kansal. Flashdb: dynamic self-tuning database for nand flash. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 410–419. ACM, 2007.

[98] Karl J. V. Nordstrom, Maria C. Albani, Geo Velikkakam James, Caroline Gutjahr, Benjamin Hartwig, Franziska Turck, Uta Paszkowski, George Coupland, and Korbinian Schneeberger. Mutation identification by direct comparison of whole-genome sequencing data from mutant and wild-type individuals using k-mers. *Nature Biotechnology*, 31(4):325–330, April 2013. ISSN 1546-1696. doi: 10.1038/nbt.2515.

[99] Brian D. Ondov, Todd J. Treangen, PÃ¡ll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren, and Adam M. Phillippy. Mash: fast genome and metagenome distance estimation using MinHash. *Genome Biology*, 17(1):132, June 2016. ISSN 1474-760X. doi: 10.1186/s13059-016-0997-x. URL https://doi.org/10.1186/s13059-016-0997-x.

[100] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. ACM. ISBN 9781450335317. doi: 10.1145/2882903.2915251. URL http://doi.acm.org/10.1145/2882903.2915251. event-place: San Francisco, California, USA.

[101] David A. Patterson. Latency Lags Bandwith. *Commun. ACM*, 47(10):71–75, October 2004. ISSN 0001-0782. doi: 10.1145/1022594.1022596. URL http://doi.acm.org/10.1145/1022594.1022596.

[102] J. Peltenburg, S. Ren, and Z. Al-Ars. Maximizing systolic array efficiency to accelerate the PairHMM Forward Algorithm. In *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 758–762, December 2016. doi: 10.1109/BIBM.2016.7822616.

[103] C. Poirier, B. Gosselin, and 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)*, pages 6489–6492, August 2015. doi: 10.1109/EMBC.2015.7319879.

[104] S. Ren, N. Ahmed, K. Bertels, and Z. Al-Ars. An Efficient GPU-Based de Bruijn Graph Construction Algorithm for Micro-Assembly. In *2018 IEEE 18th International Conference on Bioinformatics and Bioengineering (BIBE)*, pages 67–72, October 2018. doi: 10.1109/BIBE.2018.00020.

[105] Shanshan Ren, Koen Bertels, and Zaid Al-Ars. Efficient Acceleration of the Pair-HMMs Forward Algorithm for GATK HaplotypeCaller on Graphics Processing Units. *Evolutionary Bioinformatics*, 14:1176934318760543, January 2018. ISSN 1176-9343. doi: 10.1177/1176934318760543. URL https://doi.org/10.1177/1176934318760543.

[106] Shanshan Ren, Nauman Ahmed, Koen Bertels, and Zaid Al-Ars. GPU accelerated sequence alignment with traceback for GATK HaplotypeCaller. *BMC Genomics*, 20(2):184, April 2019. ISSN 1471-2164. doi: 10.1186/s12864-019-5468-9. URL https://doi.org/10.1186/s12864-019-5468-9.

[107] Andy Rimmer, Hang Phan, Iain Mathieson, Zamin Iqbal, Stephen RF Twigg, Andrew OM Wilkie, Gil McVean, Gerton Lunter, WGS500 Consortium, et al. Integrating mapping-, assembly-and haplotype-based approaches for calling variants in clinical sequencing applications. *Nature genetics*, 46(8):912, 2014.

[108] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. DSK: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, March 2013. ISSN 1367-4803. doi: 10.1093/bioinformatics/btt020. URL https://academic.oup.com/bioinformatics/article/29/5/652/253092.

[109] E. Rucci, C. Garcia, G. Botella, A. D. Giusti, M. Naiouf, and M. Prieto-Matias. Smith-Waterman Protein Search with OpenCL on an FPGA. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 3, pages 208–213, August 2015. doi: 10.1109/Trustcom.2015.634.

[110] Karl Rupp. 42 Years of Microprocessor Trend Data | Karl Rupp, 2018. URL https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[111] Marta Rybczynska. Device-to-device memory-transfer offload with P2pdma [LWN.net], 2018. URL https://lwn.net/Articles/767281/.

[112] Mohammad Sadoghi, Kenneth A Ross, Mustafa Canim, and Bishwaranjan Bhattacharjee. Exploiting ssds in operational multiversion databases. *The VLDB Journal-The International Journal on Very Large Data Bases*, 25(5):651–672, 2016.

[113] Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. Using Cascading Bloom Filters to Improve the Memory Usage for de Brujin Graphs. In Aaron Darling and Jens Stoye, editors, *Algorithms in Bioinformatics*, Lecture Notes in Computer Science, pages 364–376. Springer Berlin Heidelberg, 2013. ISBN 9783642404535.

[114] ScyllaDB. Learn about different I/O Access Methods and what we chose for Scylla, October 2017. URL https://www.scylladb.com/2017/10/05/io-access-methods-scylla/.

[115] ScyllaDB. ScyllaDB is the Real-Time Big Data Database - Take a Test Drive or Download Now, 2019. URL https://www.scylladb.com/.

[116] Sean O. Settle. High-performance Dynamic Programming on FPGAs with OpenCL. *hgpu.org*, October 2013. URL http://hgpu.org/?p=10816.

[117] Liang Shi, Jianhua Li, Chun Jason Xue, and Xuehai Zhou. Cooperating virtual memory and write buffer management for flash-based storage systems. *IEEE Trans. Very Large Scale Integr. Syst.*, 21(4):706–719, April 2013. ISSN 1063-8210. doi: 10.1109/TVLSI.2012.2193909. URL http://dx.doi.org/10.1109/TVLSI.2012.2193909.

[118] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. OS i/o path optimizations for flash solid-state drives. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 483–488, Philadelphia, PA, jun 2014. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/shin.

[119] Ashish Sirasao, Elliott Delaye, Ravi Sunkavalli, and Stephen Neuendorffer. FPGA Based OpenCL Acceleration of Genome Sequencing Software, 2015.

[120] SNIA. NVM Programming Model (NPM), 2017. URL https://www.snia.org/tech_activities/standards/curr_standards/npm.

[121] Young Hoon Son, O. Seongil, Yuhwan Ro, Jae W. Lee, and Jung Ho Ahn. Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 380–391, New York, NY, USA, 2013. ACM. ISBN 9781450320795. doi: 10.1145/2485922.2485955. URL http://doi.acm.org/10.1145/2485922.2485955. event-place: Tel-Aviv, Israel.

[122] Facebook Open Source. MyRocks | A RocksDB storage engine with MySQL, 2019. URL http://myrocks.io/.

[123] Facebook Open Source. RocksDB | A persistent key-value store, 2019. URL http://rocksdb.org/.

[124] S. Sridharan, P. Durante, C. Faerber, and 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)*, pages 1–7, August 2016. doi: 10.1109/FPL.2016.7577351.

[125] Adrian M. Stutz, Andreas Schlattl, Thomas Zichner, Jan O. Korbel, Tobias Rausch, and Vladimir Benes. DELLY: structural variant discovery by integrated paired-end and split-read analysis. *Bioinformatics*, 28(18):i333–i339, 09 2012. ISSN 1367-4803. doi: 10.1093/bioinformatics/bts378. URL https://doi.org/10.1093/bioinformatics/bts378.

[126] Devesh Tiwari, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Simona Boboila, and Peter J. Desnoyers. Reducing data movement costs using energy-efficient, active computation on SSD. In *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems*, Hollywood, CA, 2012. USENIX. URL https://www.usenix.org/conference/hotpower12/workshop-program/presentation/Tiwari.

[127] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 119–132, San Jose, CA, 2013. USENIX. ISBN 978-1-931971-99-7. URL https://www.usenix.org/conference/fast13/technical-sessions/presentation/tiwari.

[128] L. Di Tucci, K. O'Brien, M. Blott, and 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*, pages 716–721, March 2017. doi: 10.23919/DATE.2017.7927082.

[129] Vladimir I. Ulyantsev, Sergey V. Kazakov, Veronika B. Dubinkina, Alexander V. Tyakht, and Dmitry G. Alexeev. MetaFast: fast reference-free graph-based comparison of shotgun metagenomic data. *Bioinformatics*, 32(18):2760–2767, September 2016. ISSN 1367-4803. doi: 10.1093/bioinformatics/btw312. URL https://academic.oup.com/bioinformatics/article/32/18/2760/1743520.

[130] Raluca Uricaru, Guillaume Rizk, Vincent Lacroix, Elsa Quillery, Olivier Plantard, Rayan Chikhi, Claire Lemaitre, and Pierre Peterlongo. Reference-free detection of isolated SNPs. *Nucleic Acids Research*, 43(2):e11, January 2015. ISSN 1362-4962. doi: 10.1093/nar/gku1187.

[131] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association. ISBN 9781931971829. URL http://dl.acm.org/citation.cfm?id=1960475.1960480. event-place: San Jose, California.

[132] Z. Wang, B. He, and W. Zhang. A study of data partitioning on OpenCL-based FPGAs. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, September 2015. doi: 10.1109/FPL.2015.7293941.

[133] RenÃ© L. Warren, Granger G. Sutton, Steven J. M. Jones, and Robert A. Holt. Assembling millions of short DNA sequences using SSAKE. *Bioinformatics*, 23(4):500–501, February 2007. ISSN 1367-4803. doi: 10.1093/bioinformatics/btl629. URL https://academic.oup.com/bioinformatics/article/23/4/500/181258.

[134] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298485.

[135] Wikipedia. Write amplification, June 2016. URL https://en.wikipedia.org/w/index.php?title=Write_amplification&oldid=726740470. Page Version ID: 726740470.

[136] Kai Wu, Frank Ober, Shari Hamlin, and Dong Li. Early Evaluation of Intel Optane Non-Volatile Memory with HPC I/O Workloads. *arXiv:1708.02199 [cs]*, August 2017. URL http://arxiv.org/abs/1708.02199. arXiv: 1708.02199.

[137] Lisa Wu, David Bruns-Smith, Frank A. Nothaft, Qijing Huang, Sagar Karandikar, Johnny Le, Andrew Lin, Howard Mao, Brendan Sweeney, Krste Asanovic, David A. Patterson, and Anthony D. Joseph. Fpga accelerated indel realignment in the cloud. *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 277–290, 2019.

[138] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, Santa Clara, CA, July 2017. USENIX Association. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia.

[139] M. Xiao, J. Li, S. Hong, Y. Yang, J. Li, J. Wang, J. Yang, W. Ding, and L. Zhang. K-mer Counting: memory-efficient strategy, parallel computing and field of application for Bioinformatics. In *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 2561–2567, December 2018. doi: 10.1109/BIBM.2018.8621325.

[140] Chun Jason Xue, Youtao Zhang, Yiran Chen, Guangyu Sun, J. Jianhua Yang, and Hai Li. Emerging Non-volatile Memories: Opportunities and Challenges. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, pages 325–334, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0715-4. doi: 10.1145/2039370.2039420. URL http://doi.acm.org/10.1145/2039370.2039420.

[141] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, December 2017. doi: 10.1109/CloudCom.2017.14.

[142] Kai Ye, Marcel H Schulz, Quan Long, Rolf Apweiler, and Zemin Ning. Pindel: a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads. *Bioinformatics*, 25(21):2865–2871, 2009.

[143] Y. J. Yoo, T. Sandhan, J. Choi, and S. Kim. Towards simultaneous clustering and motif-modeling for a large number of protein family. In *2013 IEEE International Conference on Bioinformatics and Biomedicine*, pages 22–28, December 2013. doi: 10.1109/BIBM.2013.6732605.

[144] Yuan Taur and E. J. Nowak. CMOS devices below 0.1 /spl mu/m: how high will performance go? In *International Electron Devices Meeting. IEDM Technical Digest*, pages 215–218, December 1997. doi: 10.1109/IEDM.1997.650344.

[145] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and 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*, pages 409–420, November 2016. doi: 10.1109/SC.2016.34.