Abakus: Accelerating *k*-mer Counting with Storage Technology

10

LINGXI WU, University of Virginia, USA
MINXUAN ZHOU and WEIHONG XU, University of California San Diego, USA
ASHISH VENKAT, University of Virginia, USA
TAJANA ROSING, University of California San Diego, USA
KEVIN SKADRON, University of Virginia, USA

This work seeks to leverage Processing-with-storage-technology (PWST) to accelerate a key bioinformatics kernel called *k*-mer counting, which involves processing large files of sequence data on the disk to build a histogram of fixed-size genome sequence substrings and thereby entails prohibitively high I/O overhead. In particular, this work proposes a set of accelerator designs called Abakus that offer varying degrees of tradeoffs in terms of performance, efficiency, and hardware implementation complexity. The key to these designs is a set of domain-specific hardware extensions to accelerate the key operations for *k*-mer counting at various levels of the SSD hierarchy, with the goal of enhancing the limited computing capabilities of conventional SSDs, while exploiting the parallelism of the multi-channel, multi-way SSDs. Our evaluation suggests that Abakus can achieve 8.42×, 6.91×, and 2.32× speedup over the CPU-, GPU-, and near-data processing solutions.

CCS Concepts: • **Computer systems organization → Architectures;**

Additional Key Words and Phrases: Computer architecture, storage device, application-specific acceleration, bioinformatics

ACM Reference format:
Lingxi Wu, Minxuan Zhou, Weihong Xu, Ashish Venkat, Tajana Rosing, and Kevin Skadron. 2024. Abakus: Accelerating *k*-mer Counting with Storage Technology. *ACM Trans. Arch. Code Optim.* 21, 1, Article 10 (January 2024), 26 pages.
https://doi.org/10.1145/3632952

[1]New article, not an extension of a conference paper.

L. Wu and M. Zhou contributed equally to this research.
This work was supported by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.
Authors' addresses: L. Wu, A. Venkat, and K. Skadron, University of Virginia, P.O. Box 400246, Charlottesville, VA, 22904-4246; e-mails: lw2ef@virginia.edu, venkat@virginia.edu, skadron@virginia.edu; M. Zhou, W. Xu, and T. Rosing, University of California San Diego, 9500 Gilman Dr, La Jolla, California, USA, 92093; e-mails: skadron@virginia.edu, wexu@ucsd.edu, tajana@ucsd.edu.

1544-3566/2024/01-ART10 $15.00
https://doi.org/10.1145/3632952

1 INTRODUCTION

The scramble for vaccine development during the global COVID-19 pandemic has highlighted the profound importance of accelerating key bioinformatics tasks, particularly those that aid in vaccine research, therapeutics against bioterror, and pathogen surveillance. One of the most commonly occurring computational kernels in many bioinformatics algorithms is k-mer counting, which involves building a histogram of genome sequence substrings of a fixed size. For example, *de novo* genome assemblers that piece together an unknown genome from a collection of short reads, such as in characterizing a new virus, require a filtering step where k-mers that appear fewer times than a set threshold are regarded as erroneous and dismissed [1, 31, 47, 60, 64, 71, 79–81]. k-mer frequency information is also extensively used in the identification of repeat sequence regions [15, 43, 46, 48, 50, 62], variant calling [63], and alignment of multiple DNA or protein sequences [19, 69, 70].

This work seeks to address the critical need for accelerating k-mer counting in a scalable way for current and future bioinformatics workloads. Bioinformatics pipelines typically analyze unknown genome samples of various sizes, ranging from small viruses (e.g., a COVID test) to extremely large environmental data in metagenomics (e.g., analyzing soil samples). Investigating a k-mer counting accelerator design has tremendous economic and societal benefits. For example, the market share of metagenomics alone is expected to reach $1.4 billion by 2025 [78]. As another example in the emerging precision medicine domain, a patient's sample is first sequenced on the NovaSeq instrument in under 48 hours, producing 6∼12 TB microbiome and human DNA/RNA data. This raw sequence data is then passed through various stages, including *de novo* genome assembly for ∼3,600 CPU hours, out of which ∼60% is spent on k-mer counting [81]. Overall, efficient execution of k-mer counting can help transform many bioinformatics tasks important to human health from vision to reality. With the rapid growth of NGS, genomics is projected to soon become the largest data producer, surpassing astronomy, particle physics, and websites such as YouTube and Twitter [67], and the number of reads that need to be assembled is growing at a rate vastly outstripping Moore's Law [3], putting forth a great pressure on executing k-mer counting more efficiently.

While the idea of counting k-mers is straightforward, doing so while achieving high memory- and time efficiency is challenging. The traditional approach is to leverage large hash tables to count k-mers, and parallel implementations of these approaches distribute the input reads among several worker threads, where each thread independently extracts and counts k-mers from its share of the input [6, 8, 57, 59]. However, the size of the hash table increases exponentially with the size of k, making it infeasible to store and maintain it in memory for large genomes with many k-mer patterns [55]. Furthermore, multiple threads are bound to compete for accessing the same set of k-mer entries, resulting in frequent serialization [57]. Therefore, these approaches tend to scale poorly, imposing prohibitively high overheads in performance and hardware resource requirements.

To alleviate these overheads, state-of-the-art k-mer counting tools [7, 17, 18, 20, 34, 41] typically adopt a two-phase, disk-based (out-of-core) approach, where the input data is first partitioned into a set of files containing a subset of all k-mers to be counted in a subsequent parallel counting phase. This not only results in a much smaller memory footprint, as the memory only needs to hold a few partitions and their corresponding k-mer histograms at each iteration, but also minimizes thread contention by allowing each thread to independently build partial k-mer histograms from its share of partitions without competition from other threads. However, these approaches are also easily susceptible to overheads imposed by secondary storage devices. In particular, a large amount of data needs to be moved across the deep hardware stack (hierarchies within an SSD, main memory, cache layers, etc.) and system software stack (flash transaction layer, NVMe protocols, OS file

systems, etc.) between CPU and the hard drive, which incurs significant command and control overhead. Moreover, the external host I/O data links are typically lagging and difficult to improve compared to the internal aggregated disk bandwidth potential. In fact, our profiling experiments on a state-of-the-art disk-based *k*-mer counting software with optimized I/O access [20] reveal that a significant portion of its execution time (over 75%, see Section 3.1) is spent on file handling alone, constantly stalling the processor.

Several prior efforts have sought to accelerate *k*-mer counting using GPUs [20], FPGAs [12, 58], and processing-in-memory architectures [29, 33, 81]. However, these approaches do not consider the I/O bottleneck—while some only accelerate the compute-intensive counting phase [12, 20] or assume that the data is already loaded into memory [29, 81], others accelerate one-phase in-memory *k*-mer counting [33, 58] algorithms that do not scale with larger workloads. To improve the end-to-end performance of state-of-the-art *k*-mer counting algorithms, the I/O overhead, which is increasingly more likely to be the real bottleneck, needs to be addressed.

Integrating logic as close to the data storage media as possible is a promising alternative that addresses the I/O-bound nature of data-intensive applications. Such storage-centric architectures come in two flavors, **In-storage processing (ISP)** and **Processing-with-storage-technology (PWST)**, which are characterized by different tradeoffs and design philosophies. ISP typically directly leverages the embedded multi-core CPU controllers and DRAM inside the **solid state drive (SSD)** with modified firmware to offload computation [11, 27, 39, 42, 44, 45, 74, 76]. Commercial products include Samsung's SmartSSD [5], which features an FPGA-enhanced SSD, can also be considered an ISP implementation. *An ISP device is fundamentally still a storage product with small hardware overhead to enable computing at the place where the data reside.* This solution is less intrusive but does not always guarantee speedups [39, 40, 76]. In contrast, *a PWST architecture from the ground up is built to be a standalone, performance-optimized accelerator leveraging storage devices by* aggressively integrate custom logic at different layers of the SSD internal hierarchy (i.e., chip-, channel-, and SSD-level) to handle a variety of applications [9, 40, 49, 54, 56], and SSD is simply a helpful technology that enables processing near the huge volume of data involved in the task. This work leverages PWST to propose novel and scalable accelerator designs, collectively named *Abakus*, to eliminate the I/O overheads imposed by out-of-core *k*-mer counting.

To enable an effective end-to-end PWST-based acceleration of *k*-mer counting, we provide custom hardware solutions for a set of key *k*-mer counting operations and distribute them at different SSD levels to (1) enhance the limited computing capabilities of the existing SSD infrastructure and (2) take advantage of the multi-channel, multi-way setup of an SSD for better parallelism. We optimize performance with bioinformatics domain-specific knowledge, notably a set of hardware-implemented Bloom filters, to reduce the data volume and subsequently improve execution efficiency. The add-on logic is not only lightweight but also reusable for different purposes such as read partitioning, Bloom Filter operations, partition statistics calculations, and counting table probing.

Note that Abakus is first and foremost an accelerator, and SSD is a technology choice selected to build this accelerator for its high capacity of storing a large volume of bio-sequence data, high bandwidth, and closest proximity to raw data to largely eliminate data movement. We do not propose modifying the design of a conventional, data-storage-oriented SSD; we leverage SSD technology to build a new accelerator. Although Abakus can still act as a data storage unit, it does not need to compete in the commodity SSD market, similar to References [54] and [56]. The large size of the bioinformatics market suggests that there is a potential market for a product that is purely an accelerator that overcomes the I/O bottleneck. Furthermore, future computing environments are increasingly more likely to be heterogeneous and accelerator-abundant [35, 52]. Therefore, we envision Abakus to be deployed in the cloud with other genomics accelerators to fulfill the need for

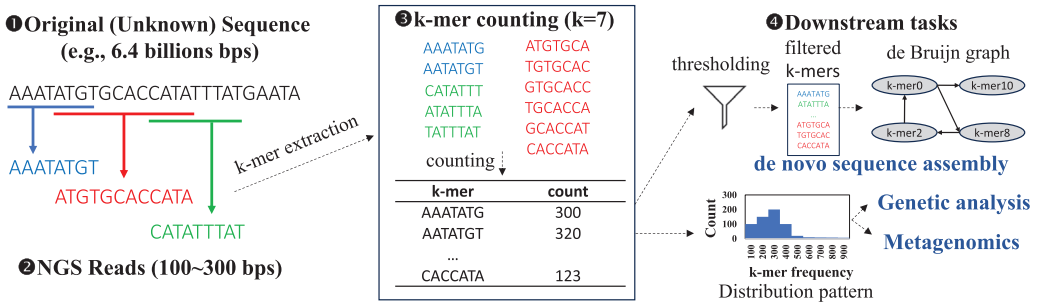


Fig. 1. The application of k-mer counting in bioinformatic pipelines.

faster genome analysis, amortizing the **Non-recurring engineering (NRE)** cost and the **Total cost of ownership (TCO)** of developing and maintaining Abakus among the entire community of users. Since data centers composed of proprietary accelerators for non-general-purpose computing such as Bitcoin mining, high-frequency trading, and web search acceleration are common nowadays, and genomic analysis is growing rapidly with high-performance sensitivity, it seems reasonable to posit interest in cloud support for faster *k*-mer counting. Due to the extensive presence of *k*-mer matching in bioinformatics, Abakus has the potential to be a staple residing in the genomic cloud to support many high-volume, planet-scale genomics analysis tasks.

We propose three designs, namely, (a) Abakus-Basic, where a set of near-storage-processing logic fits at the chip level, (b) Abakus-BF, which significantly reduces the data volume by leveraging a set of distributed Bloom filters, and (c) Abakus-OP (one-phase), which overlaps different operations to form a pipeline. Designing a *k*-mer counting accelerator as a specialized product is a flexible solution to support a variety of downstream bioinformatics pipelines, because it is such a widely used bio-kernel. Through hardware/software co-design and optimization, we incrementally add more complexity to unlock more performance. We compare the performance of Abakus with that of CPU-, GPU-, and PIM-based accelerators using large real-world genomes. Our evaluation suggests our most aggressive design, Abakus-OP, is able to achieve 6.95×/11.20× average/maximum *end-to-end* speedup over a conventional system (CPU + GPU) and 2.32×/9.84× average/maximum *end-to-end* speedup over the state-of-the-art near-data processing accelerator.

2 BACKGROUND AND RELATED WORK

2.1 *k*-mer Counting Basics

Definition. Let Σ = {A, C, G, T} denote the alphabet of DNA nucleotide (AKA base pair) sequences. A read r of length l is a sequence of nucleotides over the alphabet Σ. A k-mer is a substring of length k in r ($k \leq l$). All k-mers of a read r can be obtained by sliding a window of size k over r. Let R be a collection of such input reads. k-mer counting is defined as finding the total number of occurrences of each distinct k-mer pattern that is present in R. Consider a read set R of 3 reads: {ACGGTA, CGGTAC, TTTAC}. For k = 3, a k-mer counting algorithm would recover eight distinct 3-mers and their respective number of occurrences from read set R: {ACG:1, CGG: 2, GGT: 2, GTA:2, TAC:2, TTT:1, TTA:1, TAC:1}. k-mer is a critical step in several bioinformatic pipelines including sequence assembly [64], genetic analysis [10], metagenomics [68], and so on, as shown in Figure 1.

Use Case. The predominant genome sequencer today is based on the **Next Generation Sequencing (NGS)** technology, which cannot output the entirety of a genome sequence in one sitting but instead produces many overlapping short reads that are pieced together into the underlying genome through a process called genome assembly. Due to the errors introduced in the underlying chemical and electrical processes of the sequencers, the output reads have an error

rate of roughly one in a thousand bps. To ensure each region of the genome is correctly covered at least a minimum threshold times for a highly accurate assembly result, genomes need to pass the sequencer multiple times.

A quintessential use case of *k*-mer counting arises in the context of *de novo* genome assemblers based on the ***de* Bruijn graph (DBG)**, which leverages the overlapping portion of the NGS reads to put them together into a complete genome. *De novo* assembly is used when the sequenced reads are from an organism whose genome sequence is yet to be constructed, and there is no available reference sequence. Currently, there are only 3,500 species of complex life that have been sequenced, and only about 100 have been sequenced at "reference quality" [53], so DBG assemblers will remain an essential stage of the genome sequencing pipeline. DBG is a form of directed multigraph where each unique *k*-mer is represented as a node in the graph, and an edge is formed between two nodes if the "*k*-1" suffix of the first node exactly matches the "*k*-1" prefix of the second node. An Eulerian path that visits each node exactly once represents the target genome sequence.

The primary purpose of *k*-mer counting in a DBG assembler is to reduce the data size by removing potentially erroneous *k*-mers (i.e., graph nodes). Since each genome region has coverage of multiple NGS reads to defeat the inherent sequencing error rate, low-frequency *k*-mers such as those that appear only once or twice are likely caused by sequencing errors and, therefore, disregarded. The number of erroneous *k*-mers can be fairly large in real-world genome datasets (up to 80%), because one incorrect base pair can result in *k* erroneous overlapping *k*-mers. For this reason, *k*-mer counting is an essential step to address the genome sequencing and assembly data explosion problem. Furthermore, *k*-mer frequency information is also used to resolve branches in DBG graph traversal [81].

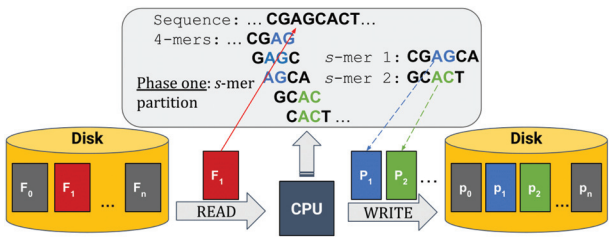
2.2 Out-of-core *k*-mer Counting

k-mer counting implementation has been thoroughly studied, and various data structures (hash tables, Tries, suffix array, etc.) and methodologies (sorting, hashing, etc.) have been employed to accelerate it. A generic histogram framework can be applied to solve the *k*-mer counting problem, but to achieve higher performance, the characteristics of genome data have to be considered, such as those that leverage minimizers and Bloom filters, introduced in the following sections.

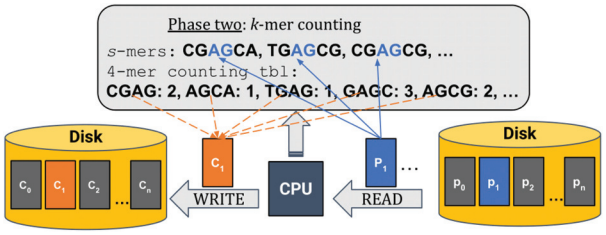
One way of generating *k*-mer histogram is to use atomics and maintain an in-memory *k*-mer frequency count table. An example is Jellyfish [57]. However, Jellyfish might have difficulty handling large genome files, because it keeps the histogram in memory [55]. This is the limitation of in-memory *k*-mer counting tools in general. One solution is batched processing but then it creates partial histograms and requires merging, degrading the benefit of in-memory counting by creating the I/O overhead. For data that fits in memory, it performs similarly to other tools [55]. Since the number of distinct *k*-mer patterns in a production genome dataset is often astronomical, resulting in a huge peak memory footprint, it is worthwhile to consider counting *k*-mers out-of-core in a batched manner. The memory consumption of processing one batch can be tuned to fit inside the memory of a workstation. Batches that are not currently being processed are temporarily saved in the secondary storage devices and later brought into the memory. Such an out-of-core design allows a small desktop to process large genomes.

Many high-performance out-of-core *k*-mer counting tools such as Gerbil [20], KMC3 [41], and DSK execute in two distinct phases: a partition phase and a counting phase, and they differ mainly in their strategies to partition input reads and their approaches to count *k*-mers (e.g., sorting vs. hashing). Figure 2 illustrates the high-level workflow of these tools.

Partitioning Phase. The partitioning phase splits reads into smaller chunks and shuffles them into a number of files. Many partition algorithms make use of a *minimizer*, which is a substring of



(a) **Phase one: input reads partitioning and distribution**



(b) **Phase Two: counting *k*-mers in each partition and build histogram**

Phase One Pseudocode

```
1  for (read in input_reads):
2      kmer_set = extract_kmers(read)
3      s-mer = " ", prev_minimizer = " "
4      for (k-mer in kmer_set):
5          mmer_set = extract_mmers(k-mer)
6          minimizer = get_minimizer(mmer_set)
7          if (prev_minimizer == minimizer):
8              s-mer = extend(k-mer)
9          else:
10             partitions[minimizer].add(s-mer)
11             s-mer = " "
12         prev_minimizer = minimizer
```

Phase Two Pseudocode

```
1  histogram.initialize()
2  for (partition in partitions):
3      hashTbl.initialize()
4      for (s-mer in partition):
5          kmer_set = extract_kmers(s-mer)
6          for (k-mer in kmer_set):
7              hashTbl[k-mer] += 1
8      histogram.add(hashTbl)
```

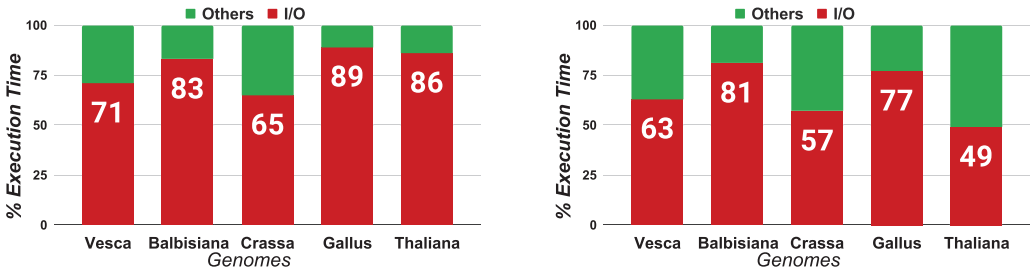
Fig. 2. Illustration of a two-phase disk-based k-mer counting algorithm workflow (F: input sequence files, P: s-mer partition files, C: k-mer counting table files).

a k-mer whose ranking is the lowest with respect to a total ordering (e.g., lexicographical order) of all possible substrings of the same size m ($m < k$). Consecutive k-mers that share the same minimizer are grouped together into a *super-mer* or s-mer and saved into a file. Figure 2 illustrates the process of splitting one read CGAGCACT into two s-mers. Let $k = 4$, $m = 2$, and all minimizer patterns are ranked based on their lexicographical order, i.e., A < C < G < T. Since the first three contiguous 4-mers {CGAG, GAGC, AGCA} share the same lexicographically smallest 2-mer, AG, they are grouped into one s-mer CGAGCA. Similarly, GCAC and CACT belong to the same s-mer GCACT. Phase one utilizes two nested sliding windows: an outer one of size k that generates overlapping k-mers from the input reads and an inner one of size m to identify a minimizer within each k-mer. Each partition file is responsible for saving s-mers generated by one or more minimizer pattern(s), which guarantees that identical k-mer patterns are saved into one partition file. Besides using the lexicographical order to rank minimizers, there are numerous other strategies to achieve partitioning effects such as even partition file sizes or shorter/longer average s-mers [20, 41].

Counting Phase. In this phase, each partition file is read from the disk to memory for k-mer extraction and counting (Figure 2). Both hashing and sorting-based approaches are viable, but sorting can be slower for longer k values [20, 55]. The sorting-based approach puts identical k-mers in adjacent positions and their counts naturally emerge. Hashing-based approaches store k-mers as keys and counters as values, and collisions can be resolved through quadratic hashing. Since partitioning guarantees that no k-mers can be found in more than one partition, the final k-mer frequency can be obtained by simply concatenating the individual k-mer histograms.

2.3 Bioinformatics Accelerators

The field of Bioinformatics has received unprecedented attention from the computer architecture community, and there has been an explosion of hardware accelerator designs targeting different stages of sequence alignment [14, 24, 28, 32, 38, 61], assembly [23, 75, 81], and various other tasks such as k-mer matching [78], insertion-deletion realignment [77], variant calling [25], and the



(a) Phase one I/O Overhead

(b) Phase two I/O Overhead

Fig. 3. Gerbil [20] I/O overhead.

list is growing. This is the first work to recognize the I/O bound nature of *k*-mer counting and proposes an in-storage-processing solution to holistically improve its end-to-end execution. We have discussed several prior *k*-mer counting accelerators [12, 20, 29, 33, 58, 81] in Section 1, including their specific hardware technologies and limitations. We also extensively analyzed a wide variety of ISP projects [9, 11, 27, 39, 40, 42, 44, 45, 49, 54, 56, 66, 74, 76] in Section 3. Among them, the most related one is GenStore [56], which accelerates read mapping, a key performance bottleneck in some genome sequence analysis applications. However, read mapping is used for guided genome assembly, whereas *k*-mer counting is usually integrated into the unguided genome assembly pipelines.

3 MOTIVATION AND KEY IDEAS

3.1 I/O Is the Bottleneck

Prior work [33] has shown that I/O greatly affects the performance of Gerbil, one of the best *k*-mer counting tools available today [20, 55]. First, one-third of Gerbil's instructions are composed of memory and I/O operations. Such frequent data accesses result in poor CPU utilization (idle for over 75% of the time). Second, as the number of intermediate files increases (necessary for larger genomes), Gerbil's runtime also linearly increases, further decreasing the CPU activity. These observations also broadly match our profiling results using VTune [30]—Gerbil's execution does not sufficiently exercise the computing capability of the underlying architecture. We then measure stalls caused by I/O. Gerbil adopts a pipelined design where pipeline stages collaborate through a set of consumer-producer queues. In phase one, a set of threads read raw sequence data from files and put them into a queue for subsequent "splitters" threads to extract *s*-mers. This requires minimal computation, and the data access is strictly sequential for both reading (from the disk) and writing (to the queue), therefore, its latency approximates the I/O response time. The second phase has a similar setup to read *s*-mers from the partition file. We estimate the I/O overhead by measuring how often the splitter threads are idle due to an empty input queue. Figure 3 shows the I/O overhead of two Gerbil phases. Clearly, I/O causes a significant overhead, and simply removing I/O overhead could improve performance by ~10×.

Note the ratio of I/O in *k*-mer counting can be different for different input genomes due to factors such as file types (compressed or uncompressed) and sequence formats (FASTQ or FASTA), which can change the amount of time the CPU spends processing the raw input, subsequently resulting in different ratios of I/O in the overall execution time. Genome characteristics also influence the I/O overhead. For example, in phase one, genome patterns determine the size of each *s*-mer (also the total number of s-mers), leading to diverse latencies to write back *s*-mer files; in phase two, some genomes work well with the hashing scheme (fewer numbers of probings per *k*-mer insertion)

while others do not, leading to longer/shorter CPU processing time and thus decreasing/increasing I/O time ratio. Regardless, we found the I/O consistently occupies a significant portion (> 50%) of runtime.

3.2 ISP *k*-mer Counting Considerations

Benefits of PWST. While *k*-mer counting can be accelerated through GPU [20], FPGA [12, 58], and even near-data-processing approaches [29, 33], PWST can fundamentally solve the bottleneck caused by data movement issues. Several characteristics of *k*-mer counting make it a good candidate to be processed at the location *where the data initially resides*. First, SSD has a notable internal (between flash chips and the SSD controller) and external (between host and SSD) bandwidth gap. Moreover, the internal bandwidth is easier to scale up, for example, by providing more channels (~1.2 GB/s per channel × number of channels [56]), while the external bandwidth (~7 GB/s for PCIe-4) is limited by expensive data pins. Furthermore, *k*-mer counting features simple computation patterns that exhibit a low compute-to-data ratio. Thus, moving the computation into SSD is always a more effective and scalable solution than bringing data out to compute if SSD can support sufficient compute throughput that saturates the internal bandwidth. Second, the input genome dataset contains a high percentage of erroneous *k*-mers, which are filtered out at the end. Moreover, a standard genome input file may also include a large chunk of information that is useless to *k*-mer counting. For example, genome files coded in the standard FASTQ format include a quality score for each base pair that are thrown away as soon as they arrive at the processor, meaning ~50% of the data brought in is never touched. However, prior accelerator work still has to pay the price of transferring such a bloated dataset to the main memory and compute units, which is sub-optimal, considering there are multiple choke points (e.g., limited external I/O and off-chip memory bandwidth) along the data path and *k*-mer counting exhibits a strong streaming pattern with limited data reuse. In our evaluation, even when all computation is free, simply reading the entire dataset into the memory makes up about 50% to 80% of the execution time. For this reason, even if a workstation is fitted with enough main memory, the I/O bottleneck still persists. Additionally, PWST approaches can offer better energy efficiency due to the reduction of unnecessary data movement. Finally, processing genome data in storage can be more scalable and cost-effective than processing-in-memory, considering an off-the-shelf dual-socket server supports over 16 NVMe SSDs that provide tens of TB of storage capacity to accommodate large genome data and dozens of GB/s of bandwidth, all at a 20–40 times lower price point than DRAM [21].

Which Storage-centric Solution Is Suitable? We consider two storage-centric architectures: (1) a centralized ISP organization that directly leverages the SSD controller and its DRAM [11, 27, 39, 42, 44, 45, 74, 76] and (2) a PWST solution with distributed and dedicated custom compute elements deeply integrated along the SSD internal data path to do the processing [9, 40, 49, 54, 56]. While both successfully reduce the data volume coming out of the storage devices, they have different capabilities and tradeoffs. We argue that the second approach is more suitable for *k*-mer counting. The embedded commodity SSD controller is usually an energy-efficient CPU (3–4× lower power than the host CPU) clocked at merely several hundred megahertz [40, 72, 76] and the DRAM is also usually smaller capacity (e.g., a few GBs), weaker (e.g., single-channel), and lower generation (DDR3). Besides that, an SSD controller could only allocate 30% to 70% of its processing time for ISP kernels, because it needs to perform other management tasks such as garbage collection [27]. Simply executing *k*-mer counting logic using SSD core results in compute-bound, offsetting the benefit of removing its I/O bottleneck.

Another motivation for adopting PWST is its better parallelism potential, which benefits both phase one and phase two. Specifically, the key operation of phase one is scanning raw reads to

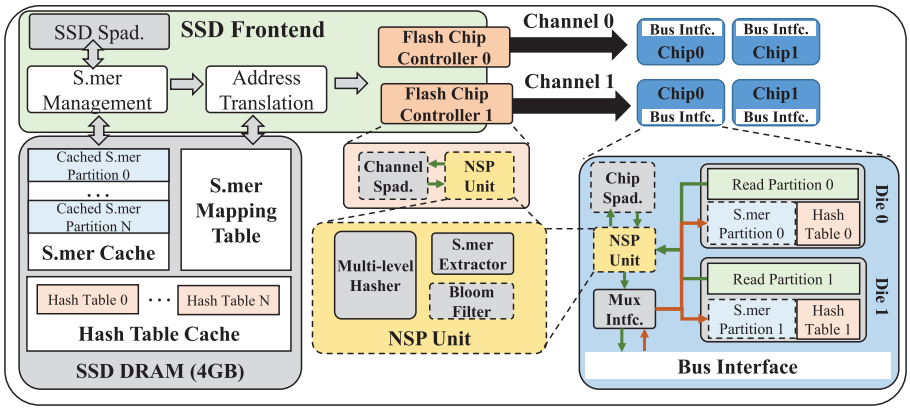


Fig. 4. The overall architecture of Abakus.

extract *s*-mers, and the key operation of phase two is scanning *s*-mers to extract *k*-mers from partitions. Both can be handled independently by a pool of "workers" (CPU threads or other comparable processing units). In addition, the logic required by each phase is relatively simple (string manipulation and hashing, which are discussed in Section 4.4.1), so we can implement a set of lightweight dedicated accelerator logic and distribute them at different levels (e.g., SSD-channel and SSD-chip) to fully exploit parallelism. A high-end SSD with 32 channels and four chips/channel can provide 128 chip-level processing units, which is difficult to achieve in a centralized ISP design where the application logic is handled in one place, such as the SSD controller. This has been noticed in prior work [9, 54], whose design space explorations conclude that a group of channel-level "weak" processors outperforms a single "beefy" SSD-level processor. Furthermore, a distributed PWST scheme offers better performance scaling as adding more chips/channels increases both data bandwidth and processing capabilities [9].

Finally, prior work [66] finds that SmartSSD [5] is limited by DRAM, because the data from the flash must be first written to the SSD DRAM and then read into the FPGA kernels. In comparison, PWST inserts logic at the chip or channel level, gaining more direct access to the flash data page, thus avoiding the trip to DRAM. *k*-mer counting is a stable algorithm and is unlikely to receive major updates; therefore, its need for performance outweighs the need for flexibility.

4 ABAKUS-BASIC DESIGN

4.1 Overview of the PWST Architecture

Figure 4 provides the architectural overview of Abakus for both the basic (Abakus-Basic) and the two optimized versions (Abakus-BF and Abakus-OP), based on a standard SSD structure. Abakus contains multiple channels, and each channel controls multiple flash chips through a **flash memory controller (FMC)**. The key components include an SSD controller (small CPU cores), a DRAM, and other control units for FTL and garbage collection (not shown in the figure) along with a custom **near-storage-processing unit (NSPU)** that is responsible for extracting *k*-mers from raw input reads and independently building partial histograms. Each NSPU directly interfaces with the flash chip page buffer, alleviates the bandwidth pressure of the SSD DRAM, and connects to a data buffer (SRAM scratchpad) to hold the data required for each operation. Note that this basic design, dubbed Abakus-Basic, is only able to exploit chip-level parallelism. In Section 5, we describe mechanisms to integrate logic into the channel and SSD levels to extract greater performance.

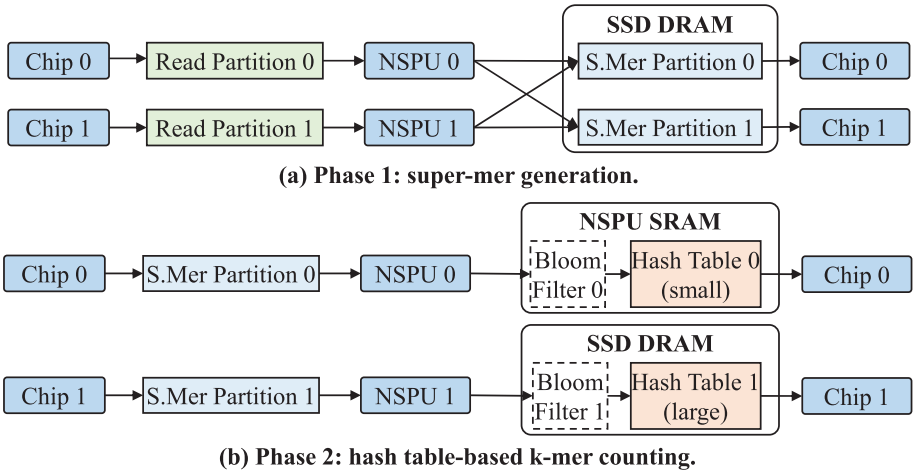


Fig. 5. The basic two-phase hardware workflow of Abakus. Bloom filter is effective in Abakus-BF (Section 5.1).

4.2 Abakus-Basic

Figure 5 shows the design and workflow of Abakus-Basic that directly maps the two-phase algorithm onto the SSD.

4.2.1 Abakus-Basic Overview. In the first phase, reads are split into s-mers that are then gathered into the same partition if they share the same minimizer (Section 2). The raw input reads are evenly distributed to each chip *a priori* so each NSPU is able to continuously read pages containing raw inputs provided to it, generate s-mers, and deposit them into its SRAM scratchpad. A partition tag is provided for each extracted s-mer to indicate its destination partition. Once the scratchpad memory is full, the corresponding NSPU transfers its data (i.e., s-mers) to the SSD DRAM that stores the received s-mer in a reserved space called the s-mer cache that is further divided into multiple sets, with each set storing s-mers that belong to the same partition. If a set is full, then all of its s-mers are written to its target chip based on the partition-to-chip mapping table, and the set space is reclaimed. We generate the mapping table based on a partitioning strategy described in Section 4.3. The partial partition is combined in the destination chip with those from the previous DRAM write-back to form the final partition. Phase one concludes when every NSPU finishes its share of input reads and the s-mer cache is emptied, with each chip storing a number of s-mer partitions as a result.

In the second phase, each chip-level NSPU reads pages that contain partitions and attempts to build one hash table for each partition to count k-mers in that partition. We adopt hash-based counting rather than a sorting-based approach given its more stable performance [20] and due to the fact that the hashing logic can be reused to enable optimizations such as the Bloom filter, a feature we use in our optimized designs (Section 5). Each chip-level NSPU has a small bookkeeping data structure (~2 KB) that tracks the address of each partition. Once the NSPU completes counting the k-mers for a partition, its associated hash table is saved/written back into the chip. For a large partition where its hash table exceeds the chip-level scratchpad memory at runtime, the unfinished partition and its hash table are transferred to the larger capacity SSD DRAM, and the SSD controller takes over the work of building the hash table. Once the SSD controller completes counting k-mer for the large partition, the hash table is written back to the chip. Note that while this basic version

executes phase one and phase two separately, similar to the CPU baseline, we later introduce a pipelined version (Section 5.2 Abakus-OP) as an optimization.

4.3 Partitioning Strategy

Clearly, the performance of the proposed design is bottlenecked by the number of large partitions whose hash tables will not fit in the chip-level NSPU's scratchpad memory. Since large partitions need to be sent to the SSD and processed using the SSD controller and DRAM, too many large partitions could degrade performance. Given our ability to process multiple partitions in parallel, we reduce the number of large partitions by dividing *s*-mers into a number of smaller partitions, allowing us to continue to exploit parallelism while minimizing additional data transfer costs. In our design, we maintain a one-to-one mapping between a minimizer and a partition, namely, one partition containing all *s*-mers that are generated from the same minimizer, therefore, the more the minimizers, the more partitions, and the smaller each partition will be. For a minimizer of length m, there are 4^m minimizers (four possible base pairs at each position). If $m = 9$ and we let the chip-level scratchpad size be 1 MB, then for the set of genomes in our evaluation, only 0.03% to 1.04% partitions are too large to be processed at the chip level. This percentage is expected to dwindle further with a larger m and scratchpad memory.

The second factor that affects performance is the basis of the assignment of partitions to chips. Uneven distribution of partitions could create a performance bottleneck similar to thread divergence resulting from workload imbalance. However, the exact sizes of partitions are unknown until the end of the first phase. To this end, we explore three possible mapping strategies: (1) a round-robin strategy where the partition corresponding to minimizer i is assigned to chip i, (2) a random distribution scheme where any partition can be assigned to any chip, and (3) a heuristic-based scheme that leverages the inherent ordering of all minimizers. We observe that a minimizer with a lower ranking is likely to generate more *s*-mers than ones with a higher ranking. Therefore, we assign pairs of partitions to chips where each pair contains a partition corresponding to a low-ranking minimizer and another one corresponding to a high-ranking minimizer. Our evaluation shows that the heuristic-based mapping has a slight performance edge compared to the random scheme, and both outperform round-robin consistently.

Note that all aforementioned partition strategies in this work fully utilize the computation resources and parallelism. The number of partitions can be calculated as 4^{m}, where m is the minimizer size. We let $m \geq 9$, which leads to at least 26,2144 partitions. A high-end SSD comes with 32 4-way channels (128 chips or NSPUs). It is unlikely to have more NSPUs than partitions. All proposed partition schemes would distribute an equal amount of partitions (2,048) to each NSPU to process. Moreover, we find out that the proposed prediction-based scheme can further minimize the tail latency, balance workload among NSPUs, and reduce Flash chip wear (Section 7.4.1).

Finally, data reduction could also improve performance. In particular, phase one shuffles input reads in the form of *s*-mers that are written first to DRAM and then to a chip, wasting bandwidth if they eventually land back on the same chip. For such *s*-mers, we save them directly to their respective partitions. Our evaluation suggests that trimming off this portion of data yields a small but noticeable (7%–10%) speedup.

4.4 Custom Hardware Design

We introduce custom logic in different levels of SSD to accelerate *k*-mer counting. Specifically, in Abakus-Basic, chip-level NSPUs process chip-independent operations (*s*-mer extraction and hash table building). We also introduce a set of custom designs at the SSD level to handle global operations or those that exceed the capability of chip-level hardware.

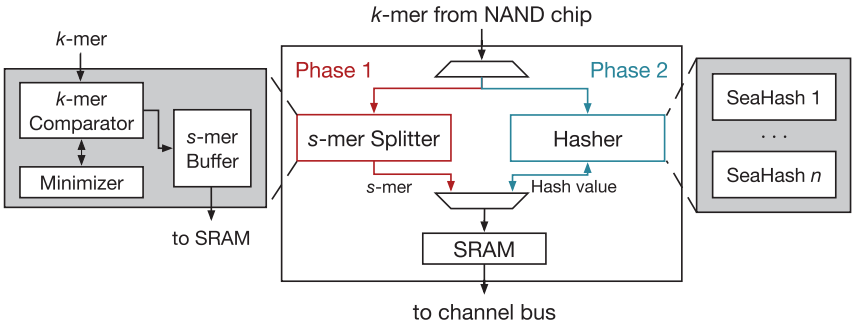


Fig. 6. Diagram of the near-storage processing unit (NSPU).

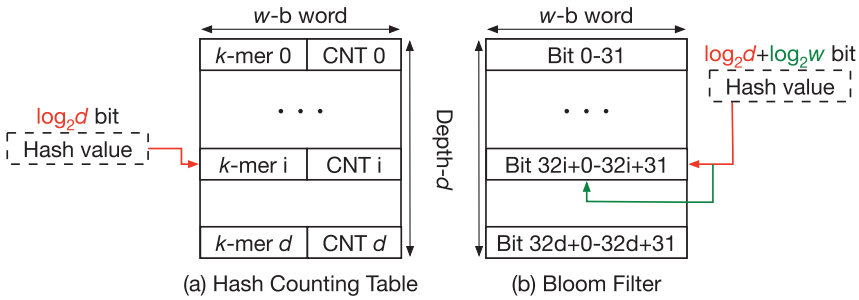


Fig. 7. Mapping for hash table and Bloom filter modes.

4.4.1 Chip-level NSPU. Each flash chip implements one NSPU to provide k-mer counting-related computations. As shown in Figure 6, each NSPU contains three main components: (1) an s-mer splitter for s-mer extraction, (2) a Hasher module to compute hash values for given k-mer, and (3) an SRAM that stores intermediate data. In phase one, the s-mer splitter is activated to iteratively compare the incoming k-mer with the stored minimizer. The k-mer is concatenated and cached in the s-mer buffer. When the next minimizer is detected (by a k-mer comparator), the cached s-mer is sent to the SRAM. The hasher module implements $n = 8$ SeaHash [2], a lightweight hashing scheme with low collision probability, with different seeds to calculate hash values. The hasher is activated during phase two to support either the hash table or the Bloom filter, depending upon the memory mapping in SRAM (shown in Figure 7). While operating in the hash table mode, the k-mer string and the **counting value (CNT)** are concatenated in one row with w-bit width, with the $\log_2 d$-bit address truncated from the hash value. The Bloom filter mode needs bit-level data granularity, so an additional $\log 2w$-bit address is added to the address. In this case, the w-bit word is first fetched from SRAM by the $\log_2 d$-bit hash, and the target bit in the row is indexed by the remaining $\log 2_w$-bit hash.

4.4.2 SSD-level Processing. While our design philosophy avoids heavy usage of SSD-level resources, we still need customized SSD-level processing to efficiently support end-to-end k-mer counting. There are multiple use cases of SSD-level processing in the Abakus workflow. First, phase one needs to merge s-mers from different chips for each partition, which is then written back to the corresponding chip. Second, during phase two, we need SSD-level processing for a large counting table that cannot fit in the low-level (e.g., chip-level) scratchpad. To support such operations, Abakus adds custom control logic and buffer at the SSD level and repurposes the SSD DRAM to store various data structures.

In conventional SSD, the SSD-level DRAM primarily acts as a write cache to hide the latency of costly SSD write. In Abakus, we re-purpose it to store the metadata as well as the global intermediate results. In phase one, it stores an *s*-mer cache and an *s*-mer mapping table to merge *s*-mers from different chips and track the locations of partitions for different *s*-mers. When chip-level NSPUs extract *s*-mers and send them to the SSD-level, the Abakus front-end stores the received *s*-mers in the corresponding *s*-mer cache set and writes the buffered set back to the chip if it is full as described earlier. In phase two, the SSD DRAM serves as backup storage for counting hash tables when a partition requires a large hash table that cannot fit in the low-level scratchpad. Since all chips share the DRAM, the SSD-level counting for different partitions needs to be serialized. Therefore, too many large hash tables could result in a performance loss.

5 ABAKUS OPTIMIZATIONS

Abakus-Basic significantly improves the execution of *k*-mer counting. However, its performance can be bottlenecked by the capacity of the chip-level scratchpad. In this section, we first describe an optimized design, Abakus-BF, that integrates a Bloom filter per NSPU leveraging the characteristics of the *k*-mer dataset, and then propose a more aggressive design, Abakus-OP, which leverages a set of additional channel-level and SSD-level NSPUs to aggressively overlap operations to merge two *k*-mer counting phases into one.

5.1 Abakus-BF

5.1.1 Abakus-BF Motivation. As alluded to in the previous sections, a Bloom filter can optimize the performance of *k*-mer counting, since low-frequency *k*-mers can be disregarded (as is typical in most use cases [1, 31, 47, 60, 64, 71, 79–81]). The exact frequency threshold varies, but it is safe to assume that single-occurrence *k*-mers is always erroneous and can be discarded. A Bloom filter is a space-efficient data structure that can be used to determine if an item has appeared previously with a small false positive rate but with zero false negative rates. It consists of *n* hash functions and a bit vector. When it encounters an item, it computes *n* hash values indexing into *n* positions of the bit vector. If all indexed bits are ones, then it assumes that it has *probably* seen the item. If some indexed bits are zeros, then it assumes that it has *definitely* never seen the item and can be inserted into the filter by flipping those zero bits to ones. In the context of *k*-mer counting, we integrate a Bloom filter to preemptively filter out as many single-occurrence *k*-mers as possible before they make it to the hash table. The procedure is to query the Bloom filter for each extracted *k*-mer before inserting it into the hash table. If the Bloom filter returns true, then the *k*-mer is inserted into the hash table. Otherwise, it is inserted into the Bloom filter. In other words, only *k*-mers that appear more than once are inserted into the hash table. Filtering out single-occurrence *k*-mers can be immensely helpful in terms of reducing the hash table size for each partition by reducing the number of keys, because single-occurrences *k*-mers make up a large portion of *k*-mer patterns (e.g., 98.32% for the Thaliana genome). See Table 2.

Notice each *k*-mer pattern can only appear in one specific partition, thanks to the minimizer-based partitioning strategy (Section 2.2 Partitioning Phase). Since each partition is assigned to a specific chip/NSPU, and no partition is split to more than one chip, there will not be any *k*-mer patterns that appear in more than one private Bloom filter.

Determining when, where, and how to incorporate a Bloom filter into Abakus is a large design space exploration problem. In this subsection, we introduce one such solution (Abakus-BF) where a set of Bloom filters is instantiated in phase two at the chip level. In the next subsection (Section 5.2), we discuss another variation where the Bloom filters are used earlier.

5.1.2 Abakus-BF Overview. Figure 5(b) illustrates the workflow of phase two in Abakus-BF. Most of the features of Abakus-Basic are retained, with the additional step of probing Bloom

filters before inserting k-mers into the hash tables during phase two. Note that Abakus-BF maintains a separate Bloom filter for each partition instead of keeping a centralized one. This is because a big Bloom filter that tracks the single k-mers in all of the partitions would be too large to fit in the chip-level scratchpad, so it has to be kept in the SSD DRAM at runtime. Subsequently, all of the chip-level NSPUs have to access the DRAM to perform their Bloom filter operations, creating a bottleneck. Alternatively, if each partition can maintain its own private (albeit smaller) Bloom filter, then each chip-level NSPU can be fully independent, preserving the parallelism.

5.1.3 Estimate the Bloom Filter Size. Building an effective Bloom filter for each partition entails solving several issues. The first is to determine an appropriate false positive rate P to find an optimal size of the bit vector without taking up an excessive amount of chip-level scratchpad memory. In Abakus-BF, both the Bloom filter, specifically its bit vector, and the hash table have to be stored in the chip-level scratchpad memory. As previously stated, a Bloom filter has a false positive rate, which means that it might incorrectly determine that a k-mer occurs multiple times, even though it occurs only once, due to which a single-occurrence k-mer might slip through the Bloom filter and get added to the hash table, incurring unnecessary hash lookups and potentially making the hash table too large to fit inside the scratchpad memory.

The interplay of the bit vector size m, false positive rate P, and the number of items to be inserted into the Bloom filter n (i.e., number of unique k-mers of a partition) can be captured in the formula: $m = -\frac{n \times \ln P}{(\ln 2)^2}$, which indicates that the false positive rate declines as the bit vector size increases, given a certain number of elements that need to be inserted to the Bloom filter. We first vary P from 1% to 25% and empirically measure the expected Bloom filter and hash table sizes for all partitions of the five selected input genomes, assuming n for each partition is known. We discover that as P decreases, the hash table sizes decrease, because more single-occurrence k-mers are filtered out. But at the same time, the Bloom filter size increases, because a more powerful Bloom filter requires a larger bit vector. A sweet spot is around $P = 5\%$, where both the bit vector and the hash table can be fit inside the chip-level scratchpad memory for the largest number of partitions per genome. Another possibility is to develop a sophisticated control unit to dynamically adjust an optimal P for each partition based on variables such as n, scratchpad memory, and the performance of the previous Bloom filter, although it may entail additional latency and control complexity.

5.1.4 Estimate Partition Cardinality. The next challenge is to estimate n, the number of unique k-mers, for each partition. A naïve approach would be to scan each partition and add its unique k-mers into a dictionary prior to phase two. However, this approach is extremely expensive in terms of space and latency and, moreover, entails performing redundant operations. Our solution is to leverage a cardinality approximation algorithm called Hyperloglog [22], which stems from its basic form called Loglog, which uses a counter x to track the longest streak of trailing (or leading) zeros of the hashed values of all the elements (i.e., k-mers) in a set (i.e., partition). The total number of unique elements in the set is then estimated as 2^x. This algorithm only needs a few bits to count tens of billions of unique elements, but it tends to have large variances, especially with smaller sets. Hyperloglog improves its accuracy using additional counters and other statistical measures to remove outliers. To integrate partition cardinality estimation into Abakus-BF, we store the counter bits per partition inside the SSD DRAM. The SSD core performs the Hyperloglog computation for that s-mer set before it is evicted to the target chip. This adds an insignificant overhead in execution time ($< 1\%$), because the SSD core is mostly idle and there is enough surplus computing power to spare (Abakus uses the SSD core very conservatively). The additional storage overhead for counters is less than 14 MB for all partitions.

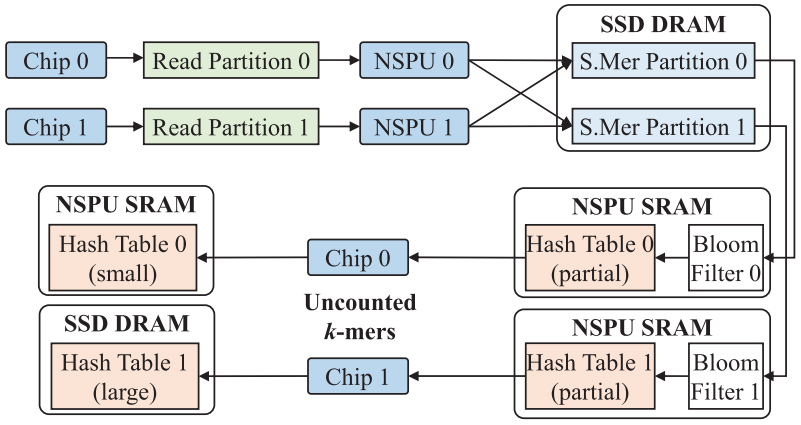


Fig. 8. Abakus-OP workflow.

5.2 Abakus-OP

5.2.1 Motivation. The performance of Abakus-Basic and Abakus-BF is primarily limited by the chip-level SRAM scratchpad memories (512 KB in current design). If a partition's Bloom filter and/or hash table is too large, then the data and computation need to be transferred to the SSD core and the DRAM to handle (Section 4.2.1), creating additional data movement and resource contention. Fitting a larger scratchpad at the chip level might be challenging due to potential power delivery issues and area overheads. Previous works have explored the placement of logic and memory at the channel and SSD levels [9, 49, 54], trading parallelism for better processing power and area budget [9, 54]. In Abakus-OP, we propose keeping the chip-level NSPU phase one logic unmodified but moving its phase two logic into the SSD and channel levels. Specifically, Abakus-OP adds an SSD-level SRAM scratchpad memory (SSD S.pad in Figure 4) to store Bloom filters and a series of channel-level NSPUs and their SRAM scratchpad memories for hash tables. With the larger capacity of the SSD and channel-level scratchpad memories, nearly all of the partitions' Bloom filters and hash tables can be accommodated without resorting to the DRAM.

Further, recall that in both Abakus-Basic and Abakus-BF, the *s*-mers partitions are written to the chips in phase one and read out again in phase two. If the partitions are converted to hash tables right away, we can skip the step of storing them back and eliminate the cost of reading the partitions out. To this end, in Abakus-OP, we orchestrate the operations pertaining to the two phases to overlap in a pipelined fashion.

5.2.2 Abakus-OP Overview. Figure 4 illustrates the architecture, and Figure 8 illustrates the workflow of Abakus-OP, which represents our most aggressive Abakus variation, where the custom logic is distributed and integrated along the SSD data path at all levels. At the SSD level, there is a large (32 MB in the current design) SRAM scratchpad memory that buffers Bloom filter(s) for one or more partition(s), and at each channel level, there is a scratchpad memory (swept from 256 KB to 32 MB for a sensitivity study in Section 7.4.3) to buffer hash tables. The chip level NSP is simplified to only have the logic that extracts *s*-mers, as the counting is performed at the channel level. We keep the total aggregated chip-level scratchpad memory of each channel at the same capacity as that of channel-level scratchpad memory.

The chip-level NSPUs extract *s*-mers and send them to the SSD DRAM to aggregate partitions. This step is exactly the same as that in Abakus-Basic and Abakus-BF. Once a set that contains *s*-mers for a partition is full, Abakus-OP loads the Bloom filter for that partition into the SSD

scratchpad memory from the chip to filter out single k-mers by breaking each s-mers down to a bag of loose k-mers used to probe the Bloom filter. The SSD-level scratchpad typically has enough capacity to simultaneously cache more Bloom filters than the number of channels, and further increasing its size offers no perceivable speedup. k-mers that passed its Bloom filter will be directed to the channel-level NSPUs for hashing, and its hash table is cached at the channel-level scratchpad that can store multiple hash tables for different partitions. As more s-mer sets corresponding to different partitions arrive, some of the cached Bloom filters in the SSD-level scratchpad and the hash tables in the channel-level scratchpad need to be evicted to make room. We once again leverage the total ordering of the minimizers to keep the "hot" ones in the scratchpads and write those corresponding to lower-ranking minimizers to the chips. This replacement scheme is highly effective, because the lower-ranking minimizers often generate smaller partitions, and their s-mer set only needs to be evicted once, with their Bloom filters, and hash tables also used only once.

Once all s-mer sets in the DRAM are drained, the entire k-mer counting process terminates. Note that the partitions are not saved and read back out in the process. However, we can still occasionally encounter large partitions whose hash table memory requirement exceeds that of the channel-level scratchpad, even after passing the Bloom filter. When this happens, the bag of loose k-mers created from the Bloom filter probing and the corresponding hash table is temporarily saved to the chips to be later processed using the SSD core and the DRAM. While this does negatively impact the performance of Abakus-OP due to the additional data movement, it is also extremely rare. Of all the genomes that we evaluated, with a 4 MB channel-level scratchpad setup, the worst case has only seven large partitions that need separate handling.

5.2.3 Abakus-OP Estimate Partition Cardinality. Although the separation of phase one (s-mer extraction at chip level) and phase two logic (k-mer counting at channel level) allows for a pipelined implementation, the partition k-mer cardinality estimation, which is essential to sizing the Bloom filters, still remains unaddressed. In Abakus-BF, this step is piggybacked with the s-mer set write-back in phase one, and Bloom filters are only later instantiated in phase two. But in Abakus-OP, s-mer sets are used to build the hash tables right away, leaving us no chance of finalizing the unique k-mer count for each partition. To this end, we add an additional stage called phase zero, where each chip locally scans reads to estimate cardinality information and sends the results (i.e., Hyperloglog counters) to the DRAM to aggregate a final estimation. The resulting data footprint is small, since only the integer counters are communicated, rather than the actual s-mers.

6 EXPERIMENT SETUP

Baseline. We compare the performance of Abakus against several existing platforms for k-mer counting, including multi-core CPU, GPU, and previous DIMM-based accelerators [29]. For CPU and GPU baselines, we use a state-of-the-art disk-based k-mer counting tool, Gerbil [20], that provides the best performance and memory efficiency among other tools [55]. The DIMM-based accelerator, NEST [29], adds parallel processing elements for k-mer counting in the rank-level of LDDIMM. NEST only accelerates the counting phase (similar to phase 2 in our algorithm) when the DRAM can fit the whole original read and the counting table. For a fair comparison, we adopt 128 GB of memory (1 channel and 2 DIMMS), which can hold all tested datasets. We use the timing and energy values reported in the NEST paper to build the roofline model, which takes in the k-mer statistics for performance evaluation. We also implement a roofline evaluation for Abakus-OP on a commercial product (SmartSSD [5]), which has an SSD-level FPGA accelerator with DDR4 SDRAM@2,400 Mbps, consuming 25 W power in total. We assume the SSD-level accelerator can efficiently process all k-mer counting operations with the same frequency as Abakus, representing the state-of-the-art in-storage processing acceleration. We assume SmartSSD has infinite

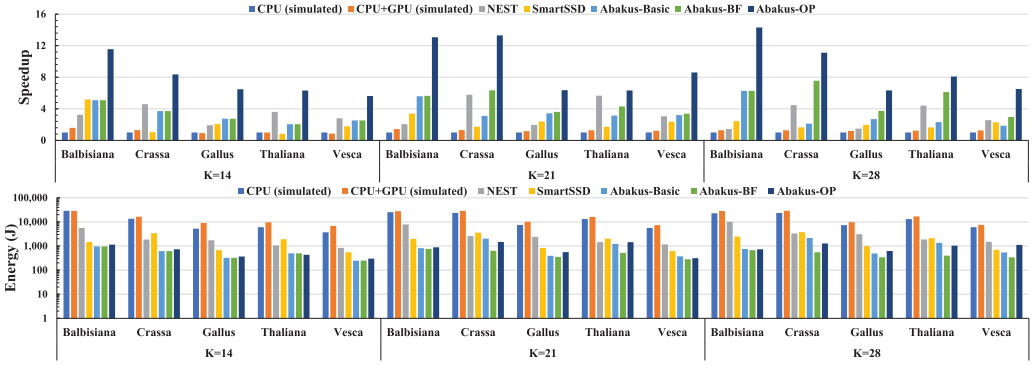


Fig. 9. The overall performance and energy across different platforms, genomes, and *k* sizes.

(unrealistic) compute throughput and DRAM capacity and evaluate the performance mainly based on internal SSD and DRAM bandwidth.

The evaluation is conducted on a server with Intel i7-11700K CPU and 64 GB DDR4-2400 RAM and NVIDIA RTX 4090 GPU. We measure CPU and GPU energy consumption using Intel Power Gadget and `nvidia-smi`. The equipped SSD is SK Hynix Gold P31 NVMe SSD with 2 TB size and 3D TLC. It is an integrated PCIe 3 ×4 bus and LPDDR4-4266 DRAM to realize a peak 3.5 GB/s sequential read rate. For a fair comparison, we follow a similar methodology described in a prior in-storage acceleration paper [54] for a simulated host baseline using the same SSD specifications as Abakus. Specifically, we collect the real SSD traces on the baseline systems and feed the collected traces to our simulation infrastructure. Figure 9 shows the performance and energy efficiency of the simulated baselines, which are 7.6% to 12.8% faster than the performance measured on the real machine.

Workloads. We evaluate five genome datasets from different species: Balbisiana, Crassa, Gallus, Thaliana, and Vesca (see Table 2), which are large enough to sufficiently exercise all hardware components in Abakus. All datasets are downloaded from NCBI [4] by entering their SRA codes from Gerbil [20].

Simulation Infrastructure. We model the performance of Abakus in a modified, trace-driven, state-of-the-art SSD simulator, MQSim [72]. We implement several new SSD commands in MQSim to simulate read, write, and *k*-mer counting computation in the chip and the channel level. We also implement a new DRAM cache mode to simulate the behavior of SSD DRAM for *k*-mer counting. We first collect *k*-mer traces of Gerbil running on the CPU workstation, as well as the statistics of each partition, and then sweep parameters relating to various Bloom filter setups, partitioning strategies, Hyperloglog parameters, and NSPU configurations including scratchpad memory sizes, to generate detailed traces that feed into the custom MQSim simulator for performance modeling. We note that our simulation platform based on MQSim [72] simulates end-to-end behaviors of SSD requests, including the host, the device, and host-device communication (e.g., PCIe bus). Table 1 summarizes the parameters for Abakus. We assume that the SSD has 32 channels and each channel has 4 chips by default. We use the **triple-level cell (TLC)** technology for flash chip, which features 60μs read latency and 700 μs write latency for an 8 KB page [26, 51]. The configuration of NSPU and buffer depends on the design. The NSPU is implemented using Verilog HDL and synthesized using Synopsys Design Compiler using TSMC 40 nm technology node. The clock frequency is 200 MHz, and the design is scaled to 22 nm. Timing and energy values of SRAM are extracted from CACTI-3DD [16] in 22 nm.

Table 1. Area and Power Breakdown

Component	Area (mm^2)	Leakage Power(mW)	Dynamic Energy(nJ)
k-mer splitter	0.004	0.001	0.001
SeaHash ×8	0.027	0.001	0.004
SRAM 512 KB	0.48	63.5	0.008
SRAM 2 MB	1.71	617.7	0.017
Abakus-Basic (128 × 512 KB SRAM/chip)	0.51/chip	**Peak Power (W)** 8.6	
Abakus-BF (128 × 512 KB SRAM/chip)	0.51/chip	**Peak Power (W)** 8.6	
Abakus-OP (32 × 2 MB SRAM/channel)	1.83/channel	**Peak Power (W)** 20.0	

7 EVALUATION

7.1 Area Overhead Analysis

Table 1 shows the area and power breakdown for NSPU and the three Abakus designs. The ASIC components of Abakus-Basic and Abakus-BF are implemented in the flash chips, resulting in 0.51 mm^2 additional overhead for each chip. Abakus-Basic and Abakus-BF have the same total area of 65.9 mm^2, while Abakus-OP (58.6 mm^2) is slightly smaller, because 2 MB SRAM has higher area efficiency than 512 KB SRAM. We observe that state-of-the-art flash chips [36, 37] have over 120 mm^2 total area, and around 10% of the area is reserved for peripheral circuits. Thus, Abakus-Basic and Abakus-BF have around 4% area overhead for each flash chip. For Abakus-OP, the overhead is negligible, since the 2 MB SRAM is implemented in FMC. Although we lack the resources to model the exact area of FMC, we note that LDPC ECC [73], a module implemented in FMC, has a comparable area with 2 MB SRAM. Therefore, we believe that all three Abakus variants are practical for manufacturing and have a minor impact on storage density.

7.2 Overall Performance and Energy Efficiency

Figure 9 shows the overall performance and energy consumption across the different platforms. All Abakus architectures adopt 32 SSD channels where each channel consists of 4 chips. We assume the same amount of distributed NSPU SRAM scratchpad (64 MB) in all architectures for a fair comparison. Specifically, both Abakus-Basic and Abakus-BF have a 512 KB scratchpad in each chip, while Abakus-OP features a 2 MB scratchpad in each channel. We find that our most aggressive design, Abakus-OP, is 8.38×, 6.95×, and 2.32× faster and consumes 15.22×, 19.93×, and 3.23× less energy than Gerbil CPU, Gerbil CPU+GPU, and NEST, respectively. As compared to SmartSSD [5], Abakus-OP is 3.47× faster and consumes 2.18× less energy. This indicates the state-of-the-art in-storage acceleration for k-mer counting cannot effectively utilize the internal bandwidth of SSD device and waste a large amount of energy using the shared SSD-level processing. The speedup over NEST is more significant for larger k than smaller k, demonstrating its substantial scalability benefits. We also observe that Abakus-OP significantly improves the performance of the naïve design (Abakus-Basic) and its optimization (Abakus-BF), outperforming them by 2.57× and 1.76× while consuming 0.98× and 1.66× energy, respectively, since the power of each scratchpad memory does not linearly scale with capacity.

We make three major observations regarding Abakus's performance in relation to its input data characteristics. First, Abakus-BF improves upon Abakus-Basic the most when there is a large percentage of single-occurrence k-mers, as evidenced by Crassa and Thaliana genomes when $k = 28$

Table 2. Input Genome Datasets (Default k = 28)

Dataset	Size (GB)	# 28-mers	# Unique	# Single
Balbisiana	91	20.5 billion	965.7 million	518.4 million
Crassa	23.3	15.7 billion	15.0 billion	14.8 billion
Gallus	28	6.3 billion	1.4 billion	479.2 million
Thaliana	17	8.9 billion	8.4 billion	8.3 billion
Vesca	13.5	5.8 billion	1.8 billion	1.4 billion

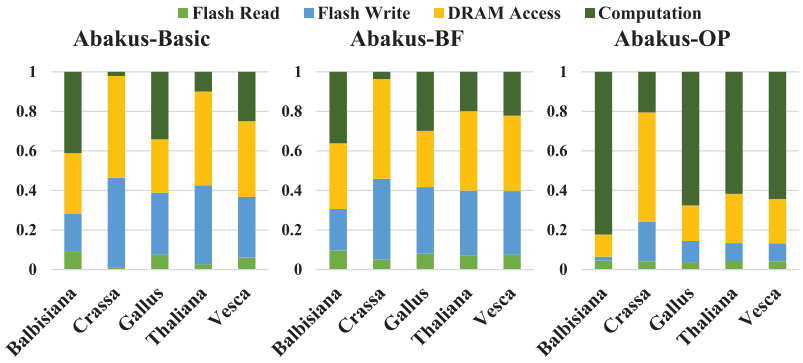


Fig. 10. Performance breakdown for the three Abakus designs.

(See Table 2). This is because Bloom filters reduce the size of each partition's hash table by preemptively removing the single-occurrence k-mers, and the number of large partitions to be processed using SSD-core and DRAM (Section 4.2.1 and 4.4.2), resulting in an overall reduction in the number of Flash writes and DRAM accesses. In fact, Abakus-BF only performs marginally better than Abakus-Basic when k = 14, because the percentage of single-occurrence 14-mers per partition is low (2%∼3%) and all partitions are small enough to fit in the chip-level scratchpad. Second, on workloads that generate large s-mer partitions, such as Balbisiana and the Crassa, where a large number of Flash writes is required, Abakus-OP significantly outperforms Abakus-BF and Abakus-Basic by removing the latency spent on saving the s-mer partitions. In addition, for these workloads, their resulting k-mer histograms, which can be estimated using #_Unique - #_Single k-mers in Table 2, are rather small, further reducing the number of Flash writes, providing a substantial speedup. Third, while Abakus-OP outperforms other Abakus setups and prior proposals in most cases, it might suffer from data explosion and workload imbalance for some input, for example, Vesca at k = 28. This is due to one large s-mer partition, which generates an excessive amount of loose k-mers for the local channel-level scratchpad to handle (Section 5.2.2), resulting in a significant increase of read/write commands that are handled by one chip.

7.3 Performance Breakdown

Figure 10 shows the performance breakdown and bandwidth utilization of three designs. We measure the execution time spent on SSD DRAM operations, chip read, chip write, and NSPU computation. As shown in the figure, Abakus-OP has the highest utilization of NSPU, where the computation takes up 59.3% of execution time on average, more than doubling the utilization rate of Abakus-Basic and Abakus-BF. This is because Abakus-Basic and Abakus-BF spends more time on costly Flash write operations which are significantly reduced in Abakus-OP via hardware Bloom filters, pipelined operation of the two phases, and an overall reduction in the DRAM access latency due to fewer occurrences of large tables in the DRAM.

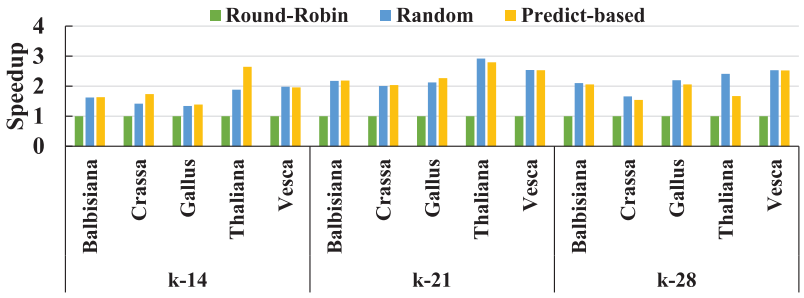


Fig. 11. The performance of different partitioning strategies.

7.4 Sensitivity Analysis

7.4.1 Partition Allocation. We first analyze the effect of different partitioning schemes for Abakus, including a naïve round-robin scheme, a fully random scheme, and a scheme using the prediction-based heuristic method (see Section 4.3). Recall that the partitioning scheme has an impact on data distribution (e.g., *s*-mers, Bloom filters, and hash tables) and ISP operations, and an unbalanced partitioning may lead to long tail latency. Figure 11 shows the result of this exploration. We observe that the random scheme uniformly outperforms the round-robin (by 1.65–2.18×) for all values of k. The prediction-based heuristic scheme is 1.87×, 2.36×, and 1.97× faster than the round-robin scheme when k is set to 14, 21, and 28, respectively. While the prediction-based scheme overall only outperforms the random scheme slightly, it does offer the benefit of distributing the data more evenly among chips, potentially limiting Flash wear. Thus, we default the partitioning strategy to the prediction-based scheme.

7.4.2 SSD Scalability. We scale the number of SSD channels from 8 to 32, which also increases the parallelism by 4×. We observe that the 16- and 32-channel architectures are 1.63× and 2.44× faster than the 8-channel architecture, respectively. The performance improvements due to the increased hardware parallelism vary across different workloads, but overall, Abakus achieves good scalability because of its ability to limit contention for high-level shared resources.

7.4.3 Buffer Size. Figure 12 explores the performance sensitivity due to varying the SRAM buffer size. As compared to 4 MB, 8 MB, 16 MB, and 32 MB channel-level buffers, we observe that the 2 MB buffer is 1.03×, 1.08×, 1.17×, and 1.43× slower. However, if the buffer size is larger than 4 MB, then the custom hardware requires more than 42.7 W power and 104.2 mm^2 area in a 32-channel SSD. At the same time, 2 MB is 1.19× faster than 1 MB while only requiring 19.8 W power and 54.6 mm^2 area overhead. Therefore, 2 MB channel-level buffers provide a good balance of performance and area/power efficiency.

Overall, Abakus-OP with a smaller channel-level scratchpad suffers from performance degradation, because it cannot efficiently handle large partitions due to a larger number of additional chip read/write commands generated for loose k-mers after Bloom Filter probing (Section 5.2.2). However, smaller scratchpad designs can exhibit better area/power efficiency than larger scratchpad designs. For example, Abakus-OP with 512 KB scratchpad per channel-level NSPU is 1.98× slower but only requires 15.47 mm^2/2.03 W area/power overhead, which is 3.53×/9.73× better than its 2 MB counterpart. Furthermore, 512 KB configuration is only 4% slower than its 2 MB counterpart in three out of five workloads. This observation shows the possibility of further reducing the overhead of Abakus while maintaining the acceleration benefits for some workloads.

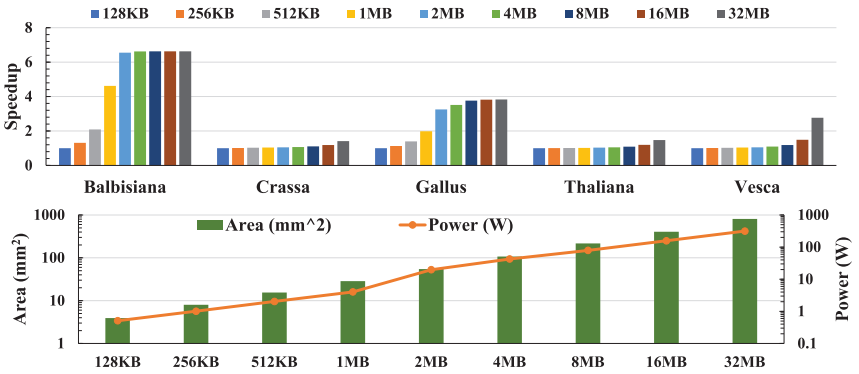


Fig. 12. Exploration of different buffer sizes for Abakus-OP.

8 DISCUSSION

Impact on Error Detection and Correction. A major concern of ISP and PWST is flash memory errors. The error detection and correction mechanisms are typically located outside the flash. For example, there is usually an ECC module at each channel-level FMC to ensure the data integrity of a page [13, 39, 42, 56]. However, the chip-level NSPUs in Abakus tap into the flash chips for fast data access, which means the error correction is skipped. Providing an ECC module per chip-level NSPU can be challenging. We argue this should not present an issue for Abakus for several reasons. First, most bioinformatics algorithms, including *k*-mer counting, are inherently error-tolerant. In fact, there have been accelerator designs using a probabilistic data structure called counting bloom filters to approximately counting *k*-mers [29, 58]. In general, *k*-mer counting algorithms do not have to be exact for most use cases [59]. Second, the raw NGS reads already have an average error rate of 0.1%, meaning there is one erroneous base pair in every thousand base pairs, which is much worse than the raw bit error rate of a flash chip (in the order of 10^{-6} [65]). Third, we simulate a process of counting 28-mers of the *E. coli* genome without ECC by randomly flipping bits based on the flash raw bit error rate [65]. We discover that roughly 7% of 28-mers are miscounted, but over 90% of them are off by only one or two. We then input this miscounted 28-mer set into a DBG assembler [47] and get no assembly score degradation, showing that ECC is likely not needed for the specific case of *k*-mer counting in storage.

Wear-leveling and Write Amplification. As the initial effort of enabling a PWST *k*-mer counting algorithm, Abakus does not lead to more severe endurance issues than the CPU baseline. First, the amount of data that needs to be written to the chips are smaller (Abakus-OP) or at least equal to (Abakus-Basic and Abakus-BF) that of the CPU baseline. Second, our partitioning scheme 4.3 ensures that each chip handles a similar amount of writes for *s*-mer partitions and hash tables. Third, writes of *s*-mer partitions and hash tables only access sequential data once in the SSD chip, making the offline remapping an effective and simple wear-leveling scheme. Write amplification happens when an SSD writes more data to disk than the host submits. Counting *k*-mers in Abakus would not cause significant write amplification, since the intermediary partitions can be written back to the chip in any order. Abakus simply appends a set of *s*-mers from SSD-DRAM to a chip. Thus, each write block can be written to an SSD chip without extensive meta-data management to erase and copy blocks of data.

Interfacing/coordinating with SSD internals/frontend Similar to how a GPU-based DNN accelerator would not need to support gaming simultaneously, Abakus is intended to function primarily as an accelerator/co-processor rather than a data storage unit; therefore, its SSD internals

does not handle requests from other applications while it is processing k-mer counting, and data pages can be safely pinned in the chip page buffers. Abakus interacts with its SSD frontend (i.e., FTL and garbage collection) minimally when counting k-mers, because the physical addresses are statistically determined by the partitioning algorithm (Section 4-C), which happens to also support wear-leveling to a certain degree, avoiding the necessity of designing a custom garbage collector.

9 CONCLUSIONS

This work proposes Abakus, a set of hardware accelerators for k-mer counting using emerging PWST architecture. The key idea is to integrate a set of custom hardware logic at the chip, channel, and SSD levels to take advantage of the internal bandwidth and parallelism potential of a modern SSD. By exploiting real DNA sequence characteristics, we optimize our design with a set of distributed Bloom filters to aggressively prune data volume. Furthermore, we propose several hardware-aware algorithm-level modifications to the classic two-phase algorithm to fully exploit the benefits of PWST. These optimizations synergistically offer the combined benefit of speedups and energy savings over the state-of-the-art CPU+GPU system by 6.95× and 19.93×.

REFERENCES

[1] Xinkun Wang. 2016. De novo genome assembly from next-generation sequencing (NGS) reads. *Next-generation Sequencing Data Analysis*. 144–155. DOI: https://doi.org/10.1201/b19532-11

[2] 2023. Crate seahash. Retrieved from https://docs.rs/seahash/latest/seahash/

[3] National Human Genome Research Institute. 2023. DNA Sequencing Costs: Data. Retrieved from https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data

[4] National Center for Biotechnology Information. 2023. NCB. U.S. National Library of Medicine. Retrieved from https://www.ncbi.nlm.nih.gov/sra

[5] Xilinx. 2023. Samsung SmartSSD. Retrieved from https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html

[6] R. S. Roy, D. Bhattacharya, and A. Schliep. [n. d.]. Turtle: Identifying frequent k-mers with cache-efficient algorithms. Retrieved from https://pubmed.ncbi.nlm.nih.gov/24618471/

[7] Peter Audano and Fredrik Vannberg. 2014. KAnalyze: A fast versatile pipelined k-mer toolkit. *Bioinformatics* 30, 14 (2014), 2070–2072. DOI: https://doi.org/10.1093/bioinformatics/btu152

[8] P. Pandey, M. A. Bender, R. Johnson, R. Patro, and B. Berger. 2023. SQUEAKR: An exact and approximate k-mer counting system. Retrieved from https://pubmed.ncbi.nlm.nih.gov/29444235/

[9] Duck-Ho Bae, Jin-Hyung Kim, Yong-Yeon Jo, Sang-Wook Kim, Hyun-Kyo Oh, and Chanik Park. 2013. Intelligent SSD: A turbo for big data mining. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*.

[10] Nathan A. Baird, Paul D. Etter, Tressa S. Atwood, Mark C. Currey, Anthony L. Shiver, Zachary A. Lewis, Eric U. Selker, William A. Cresko, and Eric A. Johnson. 2008. Rapid SNP discovery and genetic mapping using sequenced RAD markers. *PloS One* 3, 10 (2008), e3376.

[11] Simona Boboila, Youngjae Kim, Sudharshan S. Vazhkudai, Peter Desnoyers, and Galen M. Shipman. 2012. Active flash: Out-of-core data analytics on flash storage. In *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST'12)*. 1–12. DOI: https://doi.org/10.1109/MSST.2012.6232366

[12] Nicola Cadenelli, Zoran Jaksić, Jordá Polo, and David Carrera. 2019. Considerations in using OpenCL on GPUs and FPGAs for throughput-oriented genomics workloads. *Fut. Gen. Comput. Syst.* 94 (2019), 148–159. DOI: https://doi.org/10.1016/j.future.2018.11.028

[13] Yu Cai, Saugata Ghose, E. Haratsch, Yixin Luo, and O. Mutlu. 1970. Errors in flash-memory-based solid-state drives: Analysis, mitigation, and recovery: Semantic scholar. Retrieved from https://www.semanticscholar.org/paper/Errors-in-Flash-Memory-Based-Solid-State-Drives%3A-Cai-Ghose/ade903df1e67fb59069b51a0a8fc227853a4a8dc/figure/28

[14] Damla Senol Cali, Gurpreet S. Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S. Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, Anant Norion, Allison Scibisz, Sreenivas Subramoneyon, Can Alkan, Saugata Ghose, and Onur Mutlu. 2020. GenASM: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*. 951–966. DOI: https://doi.org/10.1109/MICRO50266.2020.00081

[15] D. Campagna, C. Romualdi, N. Vitulo, M. Del Favero, M. Lexa, N. Cannata, and G. Valle. 2004. Rap: A new computer program for de novo identification of repeated sequences in whole genomes. *Bioinformatics* 21, 5 (2004), 582–588. DOI: https://doi.org/10.1093/bioinformatics/bti039

[16] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2012. CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'12)*. IEEE, 33–38.

[17] Sebastian Deorowicz, Agnieszka Debudaj-Grabysz, and Szymon Grabowski. 2013. Disk-based k-mer counting on a PC. *BMC Bioinf.* 14, 1 (2013). DOI: https://doi.org/10.1186/1471-2105-14-160

[18] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. 2015. KMC 2: Fast and resource-frugal k-mer counting. *Bioinformatics* 31, 10 (2015), 1569–1576. DOI: https://doi.org/10.1093/bioinformatics/btv022

[19] Robert C. Edgar. 2004. MUSCLE: Multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Res.* 32, 5 (03 2004), 1792–1797. DOI: https://doi.org/10.1093/nar/gkh340

[20] Marius Erbert, Steffen Rechner, and Matthias Müller-Hannemann. 2017. Gerbil: A fast and memory-efficient k-mer counter with GPU-support. *Algor. Molec. Biol.* 12, 1 (2017). DOI: https://doi.org/10.1186/s13015-017-0097-9

[21] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. 2022. TriCache: A user-transparent block cache enabling high-performance out-of-core processing with in-memory programs. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. USENIX Association, 395–411. Retrieved from https://www.usenix.org/conference/osdi22/presentation/feng

[22] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and Frédéric Meunier. 2012. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. *Discr. Math. Theoret. Comput. Sci.* DMTCS Proceedings vol. AH (03 2012). DOI: https://doi.org/10.46298/dmtcs.3545

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

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

[25] Tae Jun Ham, David Bruns-Smith, Brendan Sweeney, Yejin Lee, Seong Hoon Seo, U. Gyeong Song, Young H. Oh, Krste Asanovic, Jae W. Lee, and Lisa Wu Wills. 2020. Genesis: A hardware acceleration framework for genomic data analysis. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*. 254–267. DOI: https://doi.org/10.1109/ISCA45697.2020.00031

[26] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. 2021. ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*. 147–162.

[27] Yu-Ching Hu, Murtuza Taher Lokhandwala, Te I., and Hung-Wei Tseng. 2019. Dynamic multi-resolution data storage. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52)*. Association for Computing Machinery, New York, NY, 196–210. DOI: https://doi.org/10.1145/3352460.3358282

[28] Wenqin Huangfu, Xueqi Li, Shuangchen Li, Xing Hu, Peng Gu, and Yuan Xie. 2019. MEDAL: Scalable DIMM based near data processing accelerator for DNA seeding algorithm. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52)*. Association for Computing Machinery, New York, NY, 587–599. DOI: https://doi.org/10.1145/3352460.3358329

[29] Wenqin Huangfu, Krishna T. Malladi, Shuangchen Li, Peng Gu, and Yuan Xie. 2020. NEST: DIMM based near-data-processing accelerator for sea counting. In *Proceedings of the IEEE/ACM International Conference On Computer Aided Design (ICCAD'20)*. 1–9.

[30] Intel. 2019. Intel VTune Amplifier. Retrieved from https://software.intel.com/en-us/vtune

[31] David B. Jaffe, Jonathan Butler, Sante Gnerre, Evan Mauceli, Kerstin Lindblad-Toh, Jill P. Mesirov, Michael C. Zody, and Eric S. Lander. 2003. Whole-genome sequence assembly for mammalian genomes: Arachne 2. *Genome Res.* 13, 1 (2003), 91–96. DOI: https://doi.org/10.1101/gr.828403

[32] L. Jiang and F. Zokaee. 2021. EXMA: A genomics accelerator for exact-matching. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*. IEEE Computer Society, 399–411. DOI: https://doi.org/10.1109/HPCA51647.2021.00041

[33] Biresh Kumar Joardar, Priyanka Ghosh, Partha Pratim Pande, Ananth Kalyanaraman, and Sriram Krishnamoorthy. 2019. NoC-enabled software/hardware co-design framework for accelerating k-mer counting. In *Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip (NOCS'19)*. Association for Computing Machinery, Article 4, 8 pages. DOI: https://doi.org/10.1145/3313231.3352367

[34] Lauris Kaplinski, Maarja Lepamets, and Maido Remm. 2015. GenomeTester4: A toolkit for performing basic set operations - union, intersection and complement on k-mer lists. *GigaScience* 4, 1 (2015). DOI: https://doi.org/10.1186/s13742-015-0097-y

[35] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor. 2017. Moonwalk: NRE optimization in ASIC clouds. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*.

[36] Chulbum Kim, Doo-Hyun Kim, Woopyo Jeong, Hyun-Jin Kim, Il Han Park, Hyun-Wook Park, JongHoon Lee, JiYoon Park, Yang-Lo Ahn, Ji Young Lee, Seung-Bum Kim, Hyunjun Yoon, Doeg Jae Yu, Nayoung Choi, NaHyun Kim, Hwajun Jang, JongHoon Park, Seunghwan Song, YongHa Park, Jinbae Bang, Sanggi Hong, Youngdon Choi, Moo-Sung Kim, Hyunggon Kim, Pansuk Kwak, Jeong-Don Ihm, Dae Seok Byeon, Jin-Yub Lee, Ki-Tae Park, and Kye-Hyun Kyung. 2018. A 512-Gb 3-b/cell 64-stacked WL 3-D-NAND flash memory. *IEEE J. Solid-State Circ.* 53, 1 (2018), 124–133.

[37] Doo-Hyun Kim, Hyunggon Kim, Sungwon Yun, Youngsun Song, Jisu Kim, Sung-Min Joe, Kyung-Hwa Kang, Joonsuc Jang, Hyun-Jun Yoon, Kanabin Lee, et al. 2020. 13.1 A 1Tb 4b/cell NAND flash memory with t PROG = 2ms, t R = 110μs and 1.2 Gb/s high-speed IO rate. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC'20)*. IEEE, 218–220.

[38] Jeremie S. Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. 2018. GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies. *BMC Genom.* 19, S2 (2018). DOI: https://doi.org/10.1186/s12864-018-4460-0

[39] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, and Sang-Won Lee. 2011. Fast, energy efficient scan inside flash memory. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures In conjunction with VLDB (ADMS@VLDB'11)*.

[40] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. 2016. In-storage processing of database scans and joins. *Inf. Sci.* 327, C (Jan. 2016), 183–200. DOI: https://doi.org/10.1016/j.ins.2015.07.056

[41] Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. 2017. KMC 3: Counting and manipulating k-mer statistics. *Bioinformatics* 33, 17 (05 2017), 2759–2761. DOI: https://doi.org/10.1093/bioinformatics/btx304

[42] Gunjae Koo, Kiran Kumar Matam, Te I., H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. 219–231.

[43] Stefan Kurtz, Apurva Narechania, Joshua C. Stein, and Doreen Ware. 2008. A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genom.* 9, 1 (2008). DOI: https://doi.org/10.1186/1471-2164-9-517

[44] Yunjae Lee, Jinha Chung, and Minsoo Rhu. 2022. SmartSAGE: Training large-scale graph neural networks using in-storage processing architectures. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA'22)*. Association for Computing Machinery, 932–945. DOI: https://doi.org/10.1145/3470496.3527391

[45] Young-Sik Lee, Luis Cavazos Quero, Youngjae Lee, Jin-Soo Kim, and Seungryoul Maeng. 2014. Accelerating external sorting via on-the-fly data merge in active SSDs. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*. USENIX Association. Retrieved from https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/lee

[46] A. Lefebvre, T. Lecroq, H. Dauchel, and J. Alexandre. 2003. FORRepeats: Detects repeats on entire chromosomes and between genomes. *Bioinformatics* 19, 3 (2003), 319–326. DOI: https://doi.org/10.1093/bioinformatics/btf843

[47] Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. 2015. MEGAHIT: An ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics* 31, 10 (01 2015), 1674–1676. DOI: https://doi.org/10.1093/bioinformatics/btv033

[48] Ruiqiang Li, Jia Ye, Songgang Li, Jing Wang, Yujun Han, Chen Ye, Jian Wang, Huanming Yang, Jun Yu, Gane Wong, and Jun Wang. 2005. ReAS: Recovery of ancestral sequences for transposable elements from the unassembled reads of a whole genome shotgun. *PLoS Computat. Biol.* preprint, 2005 (2005). DOI: https://doi.org/10.1371/journal.pcbi.0010043.eor

[49] Jilan Lin, Ling Liang, Zheng Qu, Ishtiyaque Ahmad, Liu Liu, Fengbin Tu, Trinabh Gupta, Yufei Ding, and Yuan Xie. 2022. INSPIRE: In-storage private information retrieval via protocol and architecture co-design. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'22)*. Association for Computing Machinery, 102–115. DOI: https://doi.org/10.1145/3470496.3527433

[50] Binghang Liu, Yujian Shi, Jianying Yuan, Xuesong Hu, Hao Zhang, Nan Li, Zhenyu Li, Yanxiang Chen, Desheng Mu, and Wei Fan. 2013. Estimation of genomic characteristics by analyzing k-mer frequency in de Novo Genome projects. Retrieved from https://arxiv.org/abs/1308.2012v1

[51] Hiroshi Maejima, Kazushige Kanda, Susumu Fujimura, Teruo Takagiwa, Susumu Ozawa, Jumpei Sato, Yoshihiko Shindo, Manabu Sato, Naoaki Kanagawa, Junji Musha, Satoshi Inoue, Katsuaki Sakurai, Naohito Morozumi, Ryo Fukuda, Yuui Shimizu, Toshifumi Hashimoto, Xu Li, Yuuki Shimizu, Kenichi Abe, Tadashi Yasufuku, Takatoshi

Minamoto, Hiroshi Yoshihara, Takahiro Yamashita, Kazuhiko Satou, Takahiro Sugimoto, Fumihiro Kono, Mitsuhiro Abe, Tomoharu Hashiguchi, Masatsugu Kojima, Yasuhiro Suematsu, Takahiro Shimizu, Akihiro Imamoto, Naoki Kobayashi, Makoto Miakashi, Kouichirou Yamaguchi, Sanad Bushnaq, Hicham Haibi, Masatsugu Ogawa, Yusuke Ochi, Kenro Kubota, Taichi Wakui, Dong He, Weihan Wang, Hiroe Minagawa, Tomoko Nishiuchi, Hao Nguyen, Kwang-Ho Kim, Ken Cheah, Yee Koh, Feng Lu, Venky Ramachandra, Srinivas Rajendra, Steve Choi, Keyur Payak, Namas Raghunathan, Spiros Georgakis, Hiroshi Sugawara, Seungpil Lee Takuya Futatsuyama, Koji Hosono, Noboru Shibata, Toshiki Hisada, Tetsuya Kaneko, and Hiroshi Nakamura. 2018. A 512Gb 3b/Cell 3D flash memory on a 96-word-line-layer technology. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC'18)*. IEEE, 336–338.

[52] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor. 2016. ASIC clouds: Specializing the datacenter. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'16)*.

[53] Smithsonian Magazine. 2018. *Ambitious Project to Sequence Genomes of 1.5 Million Species Kicks off*. Retrieved from https://www.smithsonianmag.com/smart-news/ambitious-project-sequence-genomes-15-million-species-kicks-180970697/#:~:text=Currently%2C%20scientists%20have%20only%20sequenced,used%20for%20in%2Ddepth%20research

[54] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia de Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mei Hwu. 2019. DeepStore: In-storage acceleration for intelligent queries. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52)*. Association for Computing Machinery, 224–238. DOI: https://doi.org/10.1145/3352460.3358320

[55] Swati C. Manekar and Shailesh R. Sathe. 2018. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience* 7 (2018). DOI: https://doi.org/10.1093/gigascience/giy125

[56] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr, Rachata Ausavarungnirun, Nandita Vijaykumar, Mohammed Alser, and Onur Mutlu. 2022. GenStore: A high-performance in-storage processing system for genome sequence analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*. Association for Computing Machinery, 635–654. DOI: https://doi.org/10.1145/3503222.3507702

[57] Guillaume Marçais and Carl Kingsford. 2011. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics* 27, 6 (01 2011), 764–770. DOI: https://doi.org/10.1093/bioinformatics/btr011

[58] Nathaniel Mcvicar, Chih-Ching Lin, and Scott Hauck. 2017. K-mer counting using Bloom filters with an FPGA-attached HMC. In *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*. 203–210. DOI: https://doi.org/10.1109/FCCM.2017.23

[59] Páll Melsted and Jonathan K. Pritchard. 2011. Efficient counting of K-MERS in DNA sequences using a Bloom filter. *BMC Bioinf.* 12, 1 (2011). DOI: https://doi.org/10.1186/1471-2105-12-333

[60] Jason R. Miller, Arthur L. Delcher, Sergey Koren, Eli Venter, Brian P. Walenz, Anushka Brownley, Justin Johnson, Kelvin Li, Clark Mobarry, and Granger Sutton. 2008. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics* 24, 24 (2008), 2818–2824. DOI: https://doi.org/10.1093/bioinformatics/btn548

[61] Anirban Nag, C. N. Ramachandra, Rajeev Balasubramonian, Ryan Stutsman, Edouard Giacomin, Hari Kambalasubramanyam, and Pierre-Emmanuel Gaillardon. 2019. GenCache: Leveraging in-cache operators for efficient sequence alignment. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52)*. Association for Computing Machinery, 334–346. DOI: https://doi.org/10.1145/3352460.3358308

[62] A. L. Price, N. C. Jones, and P. A. Pevzner. [n. d.]. De novo identification of repeat families in large genomes. Retrieved from https://pubmed.ncbi.nlm.nih.gov/15961478/

[63] Fanny-Dhelia Pajuste, Lauris Kaplinski, Märt Möls, Tarmo Puurand, Maarja Lepamets, and Maido Remm. 2017. FastGT: An alignment-free method for calling common SNVs directly from raw sequencing reads. *Sci. Rep.* 7 (2017). DOI: https://doi.org/10.1101/060822

[64] Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. 2001. An Eulerian path approach to DNA fragment assembly. *Proc. Nat. Acad. Sci.* 98, 17 (2001), 9748–9753. DOI: https://doi.org/10.1073/pnas.171285098

[65] Shigui Qi, Dan Feng, Nan Su, Linjun Mei, and Jingning Liu. 2017. CDF-LDPC: A new error correction method for SSD to improve the read performance. *ACM Trans. Stor.* 13, 1, Article 7 (Feb. 2017), 22 pages. DOI: https://doi.org/10.1145/3017430

[66] Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, and Jason Cong. 2021. FANS: FPGA-accelerated near-storage sorting. In *Proceedings of the IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'21)*. 106–114. DOI: https://doi.org/10.1109/FCCM51124.2021.00020

[67] Mordor Intelligence. 2017. Metagenomics Market Size, Share & Trends Analysis Report By Product (Sequencing & Data Analytics), By Technology (Sequencing, Function), By Application (Environmental), And Segment Forecasts, 2018 - 2025. Grand View Research. https://www.mordorintelligence.com/industry-reports/metagenomics-market

[68] Christian S. Riesenfeld, Patrick D. Schloss, and Jo Handelsman. 2004. Metagenomics: Genomic analysis of microbial communities. *Annu. Rev. Genet.* 38 (2004), 525–552.

[69] L. Salmela and J. Schroder. 2011. Correcting errors in short reads by multiple alignments. *Bioinformatics* 27, 11 (2011), 1455–1461. DOI: https://doi.org/10.1093/bioinformatics/btr170

[70] Leena Salmela and Jan Schröder. 2011. Correcting errors in short reads by multiple alignments. *Bioinformatics* 27, 11 (04 2011), 1455–1461. DOI: https://doi.org/10.1093/bioinformatics/btr170

[71] Jared T. Simpson, Kim Wong, Shaun D. Jackman, Jacqueline E. Schein, Steven J. M. Jones, and Inanç Birol. 2009. Abyss: A parallel assembler for short read sequence data. Retrieved from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2694472/

[72] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. 2018. MQSim: A framework for enabling realistic studies of modern multi-queue SSD devices. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. USENIX Association, 49–66. Retrieved from https://www.usenix.org/conference/fast18/presentation/tavakkol

[73] Yuta Toriyama and Dejan Marković. 2018. A 2.267-Gb/s, 93.7-pJ/bit non-binary LDPC decoder with logarithmic quantization and dual-decoding algorithm scheme for storage applications. *IEEE J. Solid-state Circ.* 53, 8 (2018), 2378–2388.

[74] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2016. Morpheus: Creating application objects efficiently for heterogeneous computing. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 53–65. DOI: https://doi.org/10.1109/ISCA.2016.15

[75] Yatish Turakhia, Gill Bejerano, and William J. Dally. 2018. *Darwin: A Genomics Co-processor Provides up to 15,000X Acceleration on Long Read Assembly*. Association for Computing Machinery, 199–213. DOI: https://doi.org/10.1145/3173162.3173193

[76] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. 2016. SSD in-storage computing for list intersection. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN'16)*. Association for Computing Machinery, Article 4, 7 pages. DOI: https://doi.org/10.1145/2933349.2933353

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

[78] Lingxi Wu, Rasool Sharifi, Marzieh. Lenjani, Kevin Skadron, and Ashish Venkat. 2021. Sieve: Scalable in-situ DRAM-based accelerator designs for massively parallel k-mer matching. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'21)*.

[79] E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, M. J. Flanigan, S. A. Kravitz, C. M. Mobarry, K. H. Reinert, K. A. Remington, E. L. Anson, R. A. Bolanos, H. H. Chou, C. M. Jordan, A. L. Halpern, S. Lonardi, E. M. Beasley, R. C. Brandon, L. Chen, P. J. Dunn, Z. Lai, Y. Liang, D. R. Nusskern, M. Zhan, Q. Zhang, X. Zheng, and Rubin. 2000. A whole-genome assembly of drosophila. *Science* 287, 5461 (2000), 2196–2204. https://doi.org/10.1126/science.287.5461.2196

[80] Daniel R. Zerbino and Ewan Birney. 2008. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.* 18, 5 (2008), 821–829. DOI: https://doi.org/10.1101/gr.074492.107

[81] Minxuan Zhou, Lingxi Wu, Muzhou Li, Niema Moshiri, Kevin Skadron, and Tajana Rosing. 2021. Ultra efficient acceleration for de novo genome assembly via near-memory computing. In *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques (PACT'21)*. 199–212. DOI: https://doi.org/10.1109/PACT52795.2021.00022

Received 11 February 2023; revised 29 October 2023; accepted 1 November 2023