Near-data-processing for Data-Intensive Applications

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Lingxi Wu

September 2023

Abstract

The information technology sector has experienced explosive growth in data-intensive applications such as bioinformatics, big data analytics, and deep neural networks (DNNs). These computing tasks have a tremendous economic impact and societal benefits, but their execution on conventional Von Neumann architectures is inefficient due to excessive data movement, a problem that rapidly growing input data sizes have exacerbated. To tackle this bottleneck, the computer architecture research community has put forward many data-centric solutions that place logic inside memory or the disk drive, commonly referred to as Near-data-processing (NDP), to reduce the latency and energy cost of data access significantly. Additionally, NDP architectures usually offer much larger parallelism, higher data bandwidth, and lower peak power consumption than CPU and GPU, allowing them to achieve orders of magnitude speedup and energy saving when executing data-intensive kernels.

This dissertation outlines four new contributions to NDP, including (1) a digital bit-serial DRAM-based processing scheme that targets a wide range of computing tasks, including bioinformatics, data analytics, pattern matching, and general-purpose arithmetic, (2) a 3D-stacked memory technology with an integrated compute layer that accelerates *de novo* genome assembly, (3) a processing-with-storage-technology (PWST) HW/SW codesigned framework that targets k-mer counting, a key bottleneck of many bioinformatics tasks, and (4) a case study of how privacy and data integrity can be breached in a recent NDP-based DNN accelerator leveraging the non-volatile memory technologies (NVM), highlighting the importance of fostering future NDP accelerator design with a security focus.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Lingxi Wu

This dissertation has been read and approved by the Examining Committee:

Kevin Skadron, Adviser

Ashish Venkat, Adviser

Mircea R. Stan, Committee Chair

Adwait Jog

Felix Lin

Sandhya Dwarkadas

Accepted for the School of Engineering and Applied Science:

Jennifer L. West, Dean, School of Engineering and Applied Science

September 2023

Acknowledgements

I am grateful to many people who have contributed to shaping this dissertation. This dissertation would not have been possible without the influence, advice, and support of many colleagues, friends, and family.

First, I would like to thank my advisors, Prof. Kevin Skadron and Ashish Venkat, for their constant and genuine support, patience, and guidance during my academic career. First, they encouraged me to dream big, aim high, and never submit a paper without trying to develop it fully. Second, they always give me plenty of time to explore and learn. I have my first work published in the fourth year. Before that, they had never given me any pressure for publication. This allowed me to focus on the work rather than the stress of not having any publications. Third, they provided me with many opportunities to make connections and collaborate with talented people from both industry and academia, which greatly elevated my vision and capabilities as a researcher. I had the greatest privilege and pleasure of working with them.

Second, I would like to thank many of my co-authors, including Rasool Sharifi, Marzieh Lenjani, Minxuan Zhou, Rahul Sreekumar, Akhil Shekar, and Deyuan Guo; without their effort, dedication, and input, none of my work would be possible. They taught me valuable lessons in various technical subjects. I will forever miss the meetings and conversations we had while we were solving difficult problems in both academia and personal life. Most importantly, they are a great source of inspiration and encouragement that help me reach the finish line. I would also like to thank many other friends and colleagues from our group, including Alif Ahmed, Farzana Siddique, Sergui Mosanu, Wole Jaiycoba, Chunkun Bo, Tommy Tracy, and many more.

Lastly, I would like to thank my family and close friends. My grandparents and my parents pushed me to pursue a Ph.D. degree many years ago. Although there were moments when I felt like it was a terrible idea and wanted to quit, they nevertheless encouraged me to continue this journey. Today, I can truly say that getting my Ph.D. is a life-changing experience, and it taught me many things that will benefit me for the rest of my life. I am also grateful to have my wife, Ningle, who is always by my side to celebrate my every little achievement, tolerate my tantrums when I experience setbacks, and support me in every way possible. A special thanks to my puppy, Dobby. I don't know what I did last life to deserve such intense and unconditional love from a little angel like that. He has been giving me a tremendous amount of emotional

support. I was also lucky to meet my roommate Kun, who quickly became my close friend. He graduated last year with a Ph.D. in Chemistry from UVA as well. He always puts others before himself, and we shared many fun and memorable moments during grad school.

Pursuing a Ph.D. degree is difficult. Looking back, it's a series of seemingly never-ending struggles sprinkled with bittersweet moments. I am truly happy to finish what I started six years ago, and I'm looking forward to new chapters in my personal and professional life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data movement dominates the execution time and energy consumption of software with large data footprints due to the strict separation of computing devices (i.e., CPU) and data storage devices (i.e., main memory or disk) in modern computing systems.

First, fetching data to the processor is expensive. For example, transferring data from the off-chip main memory to the processor takes two orders of magnitude more time and consumes three orders of magnitude higher energy than a single-precision addition [9]. For applications designed to handle large data sets beyond the available memory of a machine by using a disk to cache intermediate results (i.e., out-of-core processing), the data movement overhead is even worse. In such cases, 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.

Second, the growing dataset sizes and random access patterns further exacerbate the data movement challenges. However, the underlying hardware is having an increasingly difficult time keeping up with the data growth rate due to the end of Moore's Law and Dennard's scaling. In bioinformatics alone, the amount of data that needs to be analyzed is projected to surpass astronomy, particle physics, and popular websites such as YouTube and Twitter, far exceeding the pace of Moore's Law [10]. A similar trend is observed in enterprise database analytics. A recent Google report indicates that more than 2.5 quintillion bytes of data are expected to be generated daily worldwide. Currently, BitQuery, a database analytics platform, has already consumed 10% total cycles within the hyperscalar fleet [11]. For years, traditional Von Neumann architectures have relied on caches to hide the data access latency. However, the memory access patterns of these data-intensive applications are typically random, and their working sets are often large, leading to

poor cache behavior, even on high-end servers that feature large last-level caches [12], forcing applications to initiate frequent and slow off-chip data retrieval.

Third, The impact of data movement is most severe when the computing task required by those applications is simple and unable to mask the long data access latency, amplifying the inefficiency caused by the memory wall [13], where the processor's speed outpaces the rate at which data can be transferred to and from the memory system. Essentially, for data-intensive applications, the processors frequently stall for memory requests to be serviced across all levels due to the random nature of their data access, only to perform a small computational task on that data; therefore, the overall execution time is dominated by the idle gaps created by data movements.

Across different application domains, data movement has already been shown to be the bottleneck. In deep neural networks (DNN), a recent study demonstrates that data movement in a production DNN (GoogLeNet) accounts for roughly 70% of the overall energy consumption [14]. For example, in bioinformatics, many key kernels such as k-mer counting, which builds a histogram of short DNA sequences of size k are also bottlenecked by data access [15]. Our performance profiling experiments on a state-of-the-art k-mer counting tool [7] suggest 49% to 89% of execution stalls are caused by bringing data from a secondary data storage device to the CPU and writing data back. In Online Analytic Processing (OLAP), a key step is to apply a scan operator to filter out desired data items based on predicates, which can incur large overhead because the entire database records need to be brought into the CPU to process. Our workload profiling also suggests that this key database operation is largely memory-bound and not compute-bound (see Chapter 5.2).

The above observations have inspired numerous attempts to place compute capabilities inside the data storage devices, commonly called near-data-processing (NDP), to address the data movement challenge. NDP comes with a variety of design choices. To our knowledge, a comprehensive taxonomy that fully captures the diverse world of the NDP landscape has not been officially proposed and accepted. Here, we introduce a few design principles to interpret and categorize the NDP design space. Based on the underlying data storage technology, NDP architectures are memory-based, such as DRAM or SRAM (a.k.a Processing-in-memory or *PIM*), or storage-based, such as SSD. Based on functionalities, there are *domain-specific* NDP accelerators that target high-value individual applications or kernels (DNN, bio, database, graph, etc.) or *general-purpose*, which can perform a set of common compute primitives (arithmetic, logical, relational, etc.). Based on the specific locations that the compute logic is integrated (e.g., data row buffer, I/O interface, logic layer, etc.), an NDP work is described as either *near-data* or *in-situ*. There are also NDP works that leverage the *analog* charge-sharing property of the data array to perform computing and those that rely on *digital* circuits.

Adding to the complexity of categorizing different NDP designs, another consideration is the deployment scenarios for NDP architectures. The NDP architectures can also be broadly divided into two markets—

"memory/storage-first" and "accelerator-first" NDP, that each offer varying degrees of power-performance-area tradeoffs and present unique system integration challenges. Memory/storage-first NDP design focuses on adding compute capabilities to the memory or data storage units with minimal hardware complexity. Hence, the resulting product fits in existing memory/storage-system design constraints and has minimal impact on memory capacity. The compute-enabled memory/storage still competes in the memory/storage market with other commodity memory/storage products. Accelerator-first NDP designs seek to design the best data-parallel accelerator and use memory/storage architecture as an implementation technology to achieve this without the constraints of the traditional memory interface. The end result is an accelerator that competes with other data-parallel architectures such as GPU. Data in an accelerator-first architecture can still be read and written by the processor, for example, via CXL [16]. Potentially, the data capacity and host read/write bandwidth would be lower and device power higher than what a traditional memory interface supports. While a memory/storage-first NDP design is attractive in terms of cost because it does not significantly step away from existing memory/storage architecture, it usually entails solving many system-level challenges, such as addressing mapping, maintaining data coherency, data reshaping, and more (see Chapter 6.1 and 10.2). An accelerator-first design would be easier to integrate into a host system using existing approaches similar to other PCIe-connected co-processors. However, there are still challenges regarding developing custom compilation toolchains and APIs, and it would be more expensive to manufacture than commodity memory/storage products.

NDP is suitable for many data-intensive applications because it reduces data movement by processing at the place where data resides. Additionally, the underlying memory or storage technology that forms the foundation of NDP architecture can sustain much higher internal data bandwidth and parallelism if the compute elements can be directly connected to the data arrays. For example, the SSD has a notable internal (between flash chips and the SSD controller) vs. external (between host and SSD) bandwidth gap. Moreover, the internal bandwidth is easier to scale by providing more channels, while expensive data pins limit the external bandwidth. For a 3D-stacked memory cube where the aggregated internal bandwidth of all vaults (i.e., vertical slices similar to channels) can be an order of magnitude larger than its external I/O bandwidth [17]. For a traditional 2D planar DIMM memory, the external I/O width is limited to 64-bit, but a local row buffer in each memory subarray (see also Chapter 2.1.1) has access to 1024 bytes to 4096 bytes. Second, using a bit-serial DRAM-based NDP as an example, we argue that a well-designed NDP provides much larger data parallelism. In such architecture, each memory bit sense amplifier is augmented with a 1-bit compute unit. For a single 8-Gbit DDR4 chip (8 banks/chip) with 16K bitlines per bank, there would be 128K 1-bit processing elements. The degree of hardware parallelism can be further increased with subarray-level parallelism. Recent prototypes by Micron have demonstrated that 8 million 1-bit processing

units can be fired up simultaneously in an NDP system built on top of a DDR4 8Gib_x4 chip [18] with much smaller peak power consumption ($\sim$ 7W) than CPU and GPU.

While several recent proposals [19, 20, 21, 22, 23, 19, 24, 25, 26] have tackled the challenge of improving the performance and efficiency of NDP designs, the security and privacy implications of these architectures remain largely unexplored. Notably, semiconductor supply chain attacks are critical threats that have the ability to disrupt the operations of high-value mission-critical systems. The process of setting up a trusted end-to-end production line for emerging analog eNVM-based neuromorphic accelerators can be prohibitively time-consuming and expensive, which has paved the way for a more decentralized design-manufacturing approach that inevitably invites exploitation from bad actors to stealthily inject malicious hardware Trojans into the product [27, 28, 29, 30, 31, 32]. Successful transformation of NDP designs from prototypes to products will entail solving many challenges, and the security and privacy implications of these designs remain largely unexplored despite the potential sensitive deployment scenarios for NDP devices, including mission-critical or privacy-sensitive machine learning tasks such as medical diagnostic imaging and virtual assistance.

1.1 Thesis Statement and Contributions

Observing the recent progress and unaddressed challenges of NDP, **we hypothesize** in this dissertation that NDP systems can significantly mitigate the cost of data movement, improving the application execution speed and energy consumption for data-intensive applications. However, as NDP technology matures, the security implications of future designs should be thoroughly analyzed to achieve wide adoption. Accordingly, we design and evaluate a set of NDP accelerators to alleviate the data movement bottleneck for data-intensive workloads across different application domains. In addition to exploring the design space from performance and efficiency perspectives, we explore potential security threats in the supply chain to facilitate the design of future NDP accelerators with a security focus.

This dissertation advances the area of NDP further by primarily making four contributions (detailed in the following subsections). First, we explore applying a bit-serial DRAM-based SIMD processing scheme to accelerate various computing tasks, including bioinformatics, data analytics, pattern matching, and general-purpose computing. Second, we propose a 3D-stacked memory technology with an integrated compute layer that accelerates *de novo* genome assembly. Third, we evaluate a processing-with-storage-technology (PWST) HW/SW codesigned framework that targets k-mer counting, a key bottleneck of many bioinformatics tasks. Fourth, we conduct a case study of how privacy and data integrity can be breached in a recent

NDP-based DNN accelerator, highlighting the importance of fostering future NDP accelerator design with security in mind.

1.1.1 Digital bit-serial DRAM-based SIMD Processing

Bit-serial computing sequentially processes from operands' LSB to MSB, and each bit position is computed by applying different logic operations (AND, XOR, NOT, etc.). While traditional bit-parallel computing can compute results in one shot, bit-serial computing can outperform it by simultaneously operating on a large vector of elements bit-by-bit since its performance is sensitive to the bit-length of each element rather than the number of elements. In-DRAM bit-serial computing relies on cycling through operand DRAM rows for processing. Prior work [33, 34, 35, 36, 37, 38] have explored the potential of enabling massively-parallel bit-serial SIMD-style processing with a vertical data layout in DRAM architecture. The vertical layout allows each activation to access a bit slice across a row worth of vector elements (i.e., bitlines or lanes), hence the name bit-serial computing. Our approach differs from the prior analog-based works by performing bit-serial logic digitally using a minimalistic logic integrated at the DRAM local row buffer. Based on the underlying implemented logic, such digital bit-serial architecture can be adapted to accelerate different applications with different performance and energy profiles. Specifically, we explore the following four variations.

Variation one. Sieve: Scalable In-situ DRAM-based Accelerator Designs for Massively Parallel *k*-mer Matching

The rapid influx of biosequence data, coupled with the stagnation of the processing power of modern computing systems, highlights the critical need for exploring high-performance accelerators that can meet the ever-increasing throughput demands of modern bioinformatics applications. This work argues that processing in memory (PIM) is an effective solution to enhance the performance of k-mer matching, a critical bottleneck stage in standard bioinformatics pipelines characterized by random access patterns and low computational intensity.

This work proposes three DRAM-based in-situ k-mer matching accelerator designs (one optimized for area, one optimized for throughput, and one that strikes a balance between hardware cost and performance), dubbed Sieve, that leverage a novel data mapping scheme to allow for simultaneous comparisons of millions of DNA base pairs, lightweight matching circuitry for fast pattern matching, and an early termination mechanism that prunes unnecessary DRAM row activation to reduce latency and save energy. Evaluation of Sieve using state-of-the-art workloads with real-world datasets shows that the most aggressive design provides

an average of 326X/32X speedup and 74X/48X energy savings over multi-core-CPU/GPU baselines for k-mer matching.

Variation two. DRAM-CAM: General-Purpose Bit-Serial Exact Pattern Matching

Exact pattern matching is a widely used kernel in many applications. We observe that exact-pattern-matching-intensive workloads share similarity with k-mer matching and thus can benefit from a similar architecture like Sieve. We decided to extend Sieve with several cost-effective modifications, such as a population count logic, chip-level parallelism support, and a hardware data transposition unit, making a general-purpose DRAM-CAM and key-value store that outperforms CPU and various PIM solutions.

Variation three. Membrane: A PIM-based Architecture to Accelerate Database OLAP Queries

We then apply the DRAM-based bit-serial techniques to Online Analytical Processing (OLAP) database workloads. We explore how to map queries onto subarray-level PIM, which enables parallelism across subarrays and banks. We systematically explore mapping strategies and trade-offs between bit-serial/element-parallel and bit-parallel/element-serial designs adapted from the prior Sieve and Fulcrum architectures, respectively. We find that join operations do not map well to subarray-level PIM architectures. Thus, we need to use a software pre-join/denormalization method to transform join operations into selection/filter operations. We also learn that certain operations, such as aggregation, remain better served using the CPU. Thus, we propose a cooperative approach for analytic query processing between CPU and PIM. We then explore several dimensions in the design space of PIM architectures, including different ways to perform filter operations and a new way to return data to the CPU. We conclude that a traditional columnar database layout with a scalar processing element in the PIM-enabled subarrays (Membrane-H) for the table scan, combined with a rank-level unit (RLU) for gathering the selected elements, is the best configuration. An evaluation of an end-to-end query processing on the popular analytic benchmark SSB at scale factor 100 (a 60GB database) yields a 45.39× geometric-mean speedup over a hand-optimized AVX-512 implementation of SSB.

Variation four. DRAM-BitSIMD: Exploring the Design Tradeoffs and Opportunities in DRAM-based Bit-Serial Vector Computing

Finally, after observing the speedup and energy-saving of bit-serial DRAM-based pattern matching in bioinformatics and data analytics, we hypothesize such NDP processing can be applied to more compute primitives such as arithmetic, logical, relational, and more. We comprehensively explore the design space for bit-serial general-purpose computing logic embedded in the DRAM subarray, leveraging the massive

parallelism of DRAM row operations. We show that *digital* techniques can outperform prior analog charge-sharing techniques. Digital techniques require more area but support a wider range of computing primitives and allow a sequence of logic operations to be performed at higher clock speeds between slower subarray row reads/writes. We describe a range of bit-serial architecture choices and evaluate raw performance area and energy efficiency. We also describe a high-level vector-oriented instruction set. By analyzing bit-serial operations with different complexity levels, we identify essential hardware components for performing such operations efficiently. With software and hardware co-designing, we propose a programmable DRAM-BitSIMD architecture that achieves a good balance between bit-serial computing performance and hardware costs by introducing a bit-serial logic unit with bit registers, highly optimized bit-serial instruction set, and decoupled memory and logic instruction execution. We implement a rich set of high-level operations with bit-serial microprograms, explore the system integration approaches, and evaluate the performance on multiple widely used benchmarks. Results show that the digital architecture demonstrates a 20X speedup over CPU, 5X over GPU, and 1.7X over SIMDRAM, an analog architecture.

1.1.2 Ultra Efficient Acceleration for *De Novo* Genome Assembly via Near-Memory Computing

De novo assembly of genomes for which there is no reference is essential for novel species discovery and metagenomics. In this work, we accelerate two key performance bottlenecks of DBG-based assembly, graph construction and graph traversal with near-data processing (NDP) architecture based on a 3D-stacked memory product with a logic layer. The proposed framework distributes key operations across NDP cores to exploit a high degree of parallelism and high memory bandwidth. We propose several optimizations based on domain-specific properties to improve the performance of our design.

we map the graph construction to many independent in-memory tasks that can be allocated to a specific in-memory core. We design a new memory organization for sequence data and the resulting DBG in the NDP architecture to maximize the parallelism and balance the workload in all in-memory cores. We also propose a synchronization-based parallel in-memory traversal to generate genome contigs efficiently. We integrate the proposed techniques into an existing DBG assembly tool, and our simulation-based evaluation shows that the proposed NDP implementation can improve the performance of graph construction by 33× and traversal by 16× compared to the state-of-the-art.

1.1.3 Abakus: Accelerating *k*-mer Counting With Storage Technology

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 to enhance 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.

1.1.4 New Hardware Trojan Threats in Memristor-based Neuromorphic Computing Systems

Fast and energy-efficient execution of a DNN on traditional von Neumann architectures is challenging due to excessive data movement and inefficient digital computation. Recently, emerging memristor-based neuromorphic computing systems (MNCS), which is a form of NDP architecture that mimics biological neuron computations, are gaining traction. These neuromorphic chips perform neuron computation using arrays of resistive memory cells, and the computation results are directly represented with the analog current. However, MNCS designs focus on performance, while security concerns take a back seat. Previous works have shown that DNNs running in traditional platforms such as CPU, GPU, and FPGA environments are subject to various attacks, suggesting that MNCS is also likely vulnerable.

This work demonstrates a hardware supply chain attack (i.e., insertion of hardware Trojan during design and production) against emerging MNCS devices consisting of leakage of neuron network model parameters through a covert power side-channel that could be inserted at several points in the supply chain. We then dissect a common MNCS architecture derived from several published works and identify the analog-to-digital converters (ADC), which convert the MNCS' analog output current to digital output as an attack target. The ADCs generate a series of voltage spikes that allow the adversary to correlate MNCS switching activities with secret model parameters (i.e., synaptic weights). We design a stealthy hardware Trojan that taps into the ADC circuits to create a covert power side-channel. An adversary can infer the synaptic weights by collecting a series of power traces of the victim MNCS devices. Our evaluation suggests a simple power signal analysis can recover over 90% of synaptic weights of an MNIST MLP, and the reconstructed MLP performs similarly with the victim model, making this a viable threat to MNCS.

1.2 Dissertation Organization

The remainder of this dissertation is organized as follows:

Chapter 2: Background and Prior Works introduces various memory and data storage technologies that form the foundation of our NDP designs, the background of the computational tasks that we aim to accelerate, and the fundamentals of hardware security.

Chapter 3 - 6: Digital bit-serial DRAM-based SIMD-style Processing presents the design space exploration of fitting 1-bit digital logic in the DRAM local row buffer to process data and bit-serially. This NDP architecture is characterized by its bit-serial element-parallel processing scheme, which is highly parallel and energy efficient. We discuss the tradeoffs and design decisions of the digital DRAM-based bit-serial architectures through four variations, which target different domains and have different capabilities. This chapter, in full, is a reprint of the material as it appears in the proceeding of ISCA 2021 [12] (Chapter 3) and CAL 2022 [39] (Chapter 4). A full list of authors for Chapters 3 and 4 can be found in Appendix 10.3.1. I am the primary investigator and author of these two papers. Chapter 5 (HPCA 2024 Membrane) and Chapter 6 (ASPLOS 2024 DRAM-BitSIMD) are adapted from the material of the two works under review. A full list of authors for Chapters 5 and 6 can be found in Appendix 10.3.2. Chapter 5 is a joint effort with Akhil Shekar (UVA), Kevin P. Gaffney (UW), Martin Prammer (UW), and Helena Caminal (Cornell). **My primary contribution to this work** is the design of the Membrane-V architecture and area, power, and performance evaluation of both Membrane-V and Membrane-H. Chapter 6 is a joint effort with Deyuan Guo (UVA). **My primary contribution to this work** is the programming interface, overall performance evaluation and modeling, and the overall shaping of the design space.

Chapter 7: Custom Logic in 3D-stacked DRAM memory cube presents a HW/SW codesigned framework that integrates lightweight processing cores in the logic layer of the 3D-stacked memory cube to accelerate the de Bruijn graph (DBG) based genome assembler. This chapter, in full, is a reprint of the material as it appears in the proceeding of PACT 2021 [40]. This is a joint effort with Minxuan Zhou (UCSD). A full list of authors for Chapter 7 can be found in Appendix 10.3.1. **My primary contribution to this work** is the design of the NDP parallel DBG graph construction and its evaluation.

Chapter 8: Processing with storage technology presents an NDP architecture that leveraging Processing-with-storage-technology (PWST) to accelerate a key bioinformatics kernel called k-mer counting. This chapter, in full, is a preprint of the material submitted to TACO. This is a joint effort with Minxuan Zhou (UCSD). A full list of authors for Chapter 8 can be found in Appendix 10.3.2. **My primary contribution to this work** is the design of Abakus-Basic, Abakus-BF, Abakus-OP, and several major experiments to evaluate their performance.

Chapter 9: Hardware Trojan in DNN NDP Accelerator presents for the first time the feasibility of carrying out a hardware supply chain attack against a neuromorphic DNN accelerator that performs neuron computation inside resistive memory cell arrays. This chapter, in full, is a reprint of the material as it appears in the proceeding of DATE 2023 [41]. This is a joint effort with Rahul Sreekumar (UVA). A full list of authors of Chapter 9 can be found in Appendix 10.3.1. **My primary contribution to this work** is the design of the overall attack scheme and evaluating the effectiveness of the weight recovery attack.

Chapter 10: Conclusions summarizes the dissertation and describes potential future research directions.

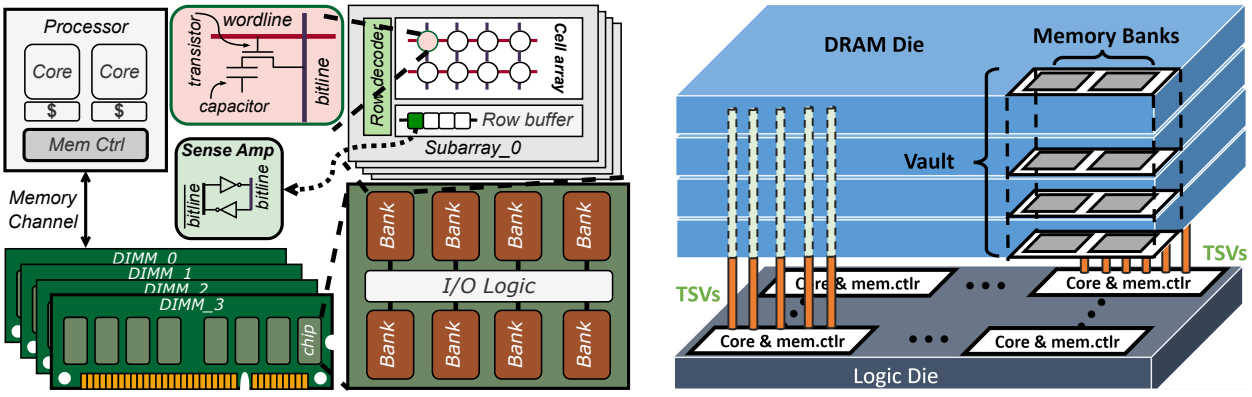
Chapter 2

Background

In this section, we introduce the fundamentals of DRAM memories.

2.1 DRAM Memory Technology and SSD

2.1.1 2D Planar DRAM



(a). 2D Planar DRAM Memory **(a). 3D-stacked DRAM Memory**

Figure 2.1: DRAM memory organization

DRAM Organization Figure 2.1 (a) illustrates the key components of a commodity 2D planar DRAM, which is usually organized as a hierarchy of memory cells with peripheral controls and I/O logic to support high-density data storage and high-bandwidth data read and write. The host CPU communicates with a set of DDR (double data rate) DRAM modules (DIMMs) through one or more memory channels controlled by the memory controller. Each DIMM contains one or two *ranks* of chips. Chips grouped into the same rank react to the same DRAM commands and behave in a lockstep manner, i.e., accessing a row from *bank_i* in one chip entails simultaneously accessing the same row in *bank_i* from all chips in the same rank. Organizing

conventional (i.e., DDR) DRAM into ranks is primarily for two reasons. (1) A large monolithic single-chip DRAM bank would be slow due to the wire delay. (2) The I/O width of a DDR DRAM chip is limited to 8 - 16 bits, so to support a cache fill, a rank is needed to achieve a wide read/write. Essentially, each logical DRAM bank is physically partitioned among a rank of chips.

Banks are further broken down into groups (32 to 64) of 2D arrays of cells called *subarrays*, and each subarray typically consists of 512–1024 rows. However, these values may vary from product to product to optimize overall DRAM density and performance. Conventionally, only one subarray per bank is activated to respond to a DRAM command. Additional performance can be extracted by exploiting subarray-level parallelism (SALP) [42]. SALP has three variations, and we leverage the most aggressive form called MASA (Multitude of Activated Subarrays). MASA exploits the fact that each subarray has its own local row buffer that latches a recently activated DRAM row; therefore, by adding one additional single-bit latch to each subarray's peripheral logic as well as a new 1-bit global control signal, the SALP mechanism can keep multiple subarrays in the same bank active, reducing bank-level request conflict and improve row-buffer hit rate to the same bank. When applied to NDP designs, subarray-level parallelism (SALP) substantially increases the computing throughput. However, sustaining the activation of several subarrays simultaneously requires more power than traditional DRAM chips and system interfaces are designed to support. However, the performance benefit usually justifies the cost.

DRAM Data Access Bits are stored as charges in the DRAM cell's capacitor and converted to digital values upon access by a row of *sense amplifiers* known as the *row buffer*. Accessing DRAM includes three essential steps: (1). Row activate, which selects a row of cells by asserting the *wordline* and connecting the cells' transistors to the *bitlines*, at which point the charge flows to the sense amplifiers. It takes *tRCD* ($\approx 15ns$) delay for the sense amps to detect and stabilize the charges. Note connecting a DRAM row to the row buffer destroys the original values, and the sense amplifiers take additional time to restore the charges. The delay between the row activation and cell charge restoration is governed by the timing parameter *tRAS* ($\approx 35ns$). (2). After the bits are latched in the row buffer, the memory controller can issue a Read or Write command and a column address to access the desired bits. (3). Finally, to access another DRAM row, a *precharge* stage lasting $\sim$ 15 ns (*tRP*) is needed to disconnect the current capacitors from the bitlines by disabling the wordline and restore the bitline voltage to their quiescent state.

2.1.2 3D-stacked Memory Cube

Emerging 3D-stacked DRAMs, such as Micron's hybrid memory cube (HMC) illustrated in Figure 2.1 (b), and high bandwidth memory (HBM) [43, 44] of different vendors such as Samsung, SK Hynix, and Micron,

are popular platforms to enable NDP functionality. Since HMC and HBM share many high-level design principles, we use an HMC cube illustrated in Figure 2.1 (b) to explain the core concepts of a 3D-stacked memory product.

HMC Organizations For each of a 3D-stacked memory module (AKA a *cube* in HMC's terminology), there are multiple (4 - 8) vertically-stacked memory dies on top of a logic die (e.g., a logic layer in HMC or a layer of interposer in HBM). A HMC cube is vertically sliced into multiple (16 - 32) independent *vaults* (8 - 32 MB / vault), similar to pseudo channels in HBM. Each vault has its own memory controller in the logic layer, and the data is transferred from the DRAM dies to the logic die through fast through-silicon vias (TSVs), which provide higher bandwidth, lower latency, and lower communication energy consumption within a cube than comparable 2D DIMM organizations [45].

HMC Characteristics Differs from traditional 2D DIMM-based memory, HMC adopts a packet-based (16B to 128B) communication protocol implemented with high-speed serialization/deserialization (SerDes) circuits [45], which archives higher raw link bandwidth than achievable with synchronous bus-based interfaces implemented in the traditional DIMM memory. Such communication protocols provide a much wider interface than DDR DIMM, which has a narrow I/O: 4 - 16 bits per chip and 64 bits per rank. In terms of the bandwidth, externally, there are eight 60 GB/s high-speed serial links as the off-chip interface, which means a single HMC cube providing 480 GB/s of host-to-memory bandwidth [46]. To scale out the HMC-based memory, multiple cubes can be connected using a crossbar network. Internally, with 32 vaults per cube and 2Gb/s of TSV signaling rate, an HMC archives an internal bandwidth of 512 GB/s [17]. Additionally, 3D stacking has the advantage of a smaller form factor compared to DIMM technology.

HMC-based Accelerators Due to its advantages, 3D-stacked memory has been used by some high-end GPUs and FPGAs to improve performance. It is also a starting point for potential consideration to achieve NDP. 3D-stacked memory-based NDP solutions come in various forms. Recent proposals can be separated into two categories: (1) in-situ processing where the logic is embedded at the row-buffer level such as Fulcrum [47], or integrating a custom core/application logic at the logic layer per vault [17, 48, 49, 50]. The NDP system can scale out by connecting multiple cubes using high-speed serial links to form a network of NDP cores. Scaling out the NDP system built on top of HMC can simultaneously increase the memory capacity, parallelism, and aggregated memory bandwidth, which is ideal for many big-data applications such as parallel genomics workloads with a large memory footprint and high bandwidth demand. We evaluate the effectiveness of NDP design in the context of HMC architecture, which provides concrete parameters accessible to researchers. However, our optimizations may also be applied directly to other 3D-stacked memories like HBM, which shares a similar organization (e.g., channels vs. vaults) [43, 44].

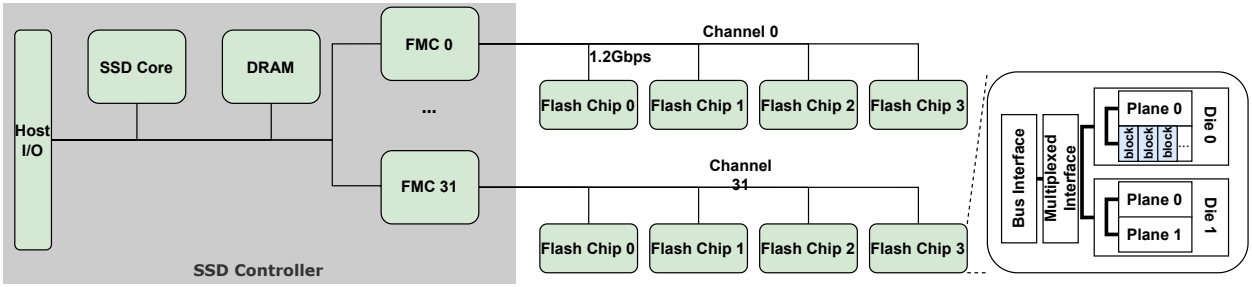


Figure 2.2: Solid State Drive (SSD) organization

2.1.3 SSD

Figure 2.2 shows the block diagram of a commodity SSD. An SSD can be conceptually divided into a front-end and a backend. The backend consists of NAND flash chips organized into multiple ($8 \sim 32$) independent channels to increase the I/O parallelism. In addition, each chip comprises several dies; each can serve a different memory request but must compete with other sibling dies for the same data and control paths [51]. Each die comprises multiple data blocks (groups of 4∼8 KB pages), organized into several planes that generally respond to commands in a lock-step manner [51]. There is usually a register at the die level [51, 52] to buffer a data page for an R/W command. The front end consists of a controller core, DRAM, and a set of flash memory controllers (FMC). An SSD controller, usually a low-frequency, energy-efficient CPU, has three responsibilities: (1) communicate with the host through SATA or NVMe protocols to handle I/O requests, (2) convert I/O requests to flash transactions and submit them to chip-level queues, in addition to supporting address translation, read/write caching, garbage collection, among other tasks, and (3) coordinate with FMCs which issue commands to and transfer data to/from the chips. The DRAM stores the data structures the controller requires to execute various flash management tasks and buffers data.

2.2 Bioinformatics

The field of bioinformatics has enabled significant advances in human health through its contributions to precision medicine, disease surveillance, population genetics, and many other critical applications. 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 bio-accelerator designs has tremendous economic and societal benefits. The market share of metagenomics alone is expected to reach $1.4 billion by 2025 [53]. 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. To develop personalized treatment from these samples, raw

```
1.  for (query_seq: query_list){
2.      kmer_list = []
3.      payload_list = []
4.      ... // store k-mers from query_seq
5.      for (kmer: kmer_list){
6.          result = query_kmer(kmer, reference k-mer set, ...)
7.          if (result != NULL) // found match, retrieve payload
8.              payload_list.add(result.payload)
9.          else
10.             ... // no match
11.     }
12.     ... // classify query_seq using payload_list
13. }
```

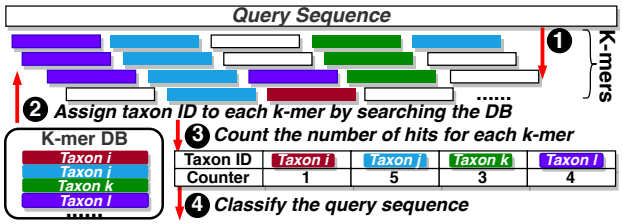


Figure 2.3: ***k*-mer matching pseudo code and illustration**

sequences are passed through, often in parallel, various metagenomics stages with k-mer matching, DBG genome assembly, and k-mer counting on the critical path. ∼68 days can be spent on software called Kraken [1] during the critical k-mer matching stage, ∼3600 CPU hours on the *de novo* genome assembly. Efficient execution of k-mer matching, DBG genome assembly, and k-mer counting can help transform many life-saving tasks from vision to reality. Each of the three bioinformatics kernels is described as follows.

2.2.1 *k*-mer matching

A DNA sequence is a series of nucleotide bases commonly denoted by four letters (bases): A, C, G, and T. k-mer are subsequences of size k. Metagenomic algorithms attempt to assign taxonomic labels to genetic fragments (sequences) with unknown origins. A "taxonomic label" assigns a sequence to a particular organism or species. Traditionally, this is done by aligning an individual query sequence against reference sequences, which can be prohibitively slow. Processing a metagenomics file containing 10^7 sequences using an alignment-based BLAST algorithm takes weeks of CPU time [54, 55]. Experts predict that genomics will soon become the most prominent data producer in the next decade [56], demanding more scalable sequence analysis infrastructure. More recently, alignment-free tools that rely on simple k-mer matching have emerged to aid large-scale genome analysis tasks, because properly labeled k-mers are often sufficient to infer taxonomic and functional information of a sequence [5, 1, 2, 57].

Figure 2.3 (a) illustrates the process of a typical k-mer-matching-based sequence classifier. In an offline stage, a reference k-mer database is built, which maps unique k-mer patterns to their taxon labels. For example, if a 5-mer "AACTG" can only be found in the E.coli bacteria sequence, an entry that maps "AACTG" to E.coli is stored. At run time, k-mer matching algorithms slide a window of size k across the query sequence, and for each resulting k-mer, they attempt to retrieve the associated taxon label from the database. The function **query_kmer** in line 6 is repeatedly called to search each k-mer in the database. If the query k-mer exists in the database (k-mer hit), its taxon label (payload) is retrieved; otherwise, we move on and compare the next k-mer in the query. Once all k-mers in a query are processed, the taxon labels of

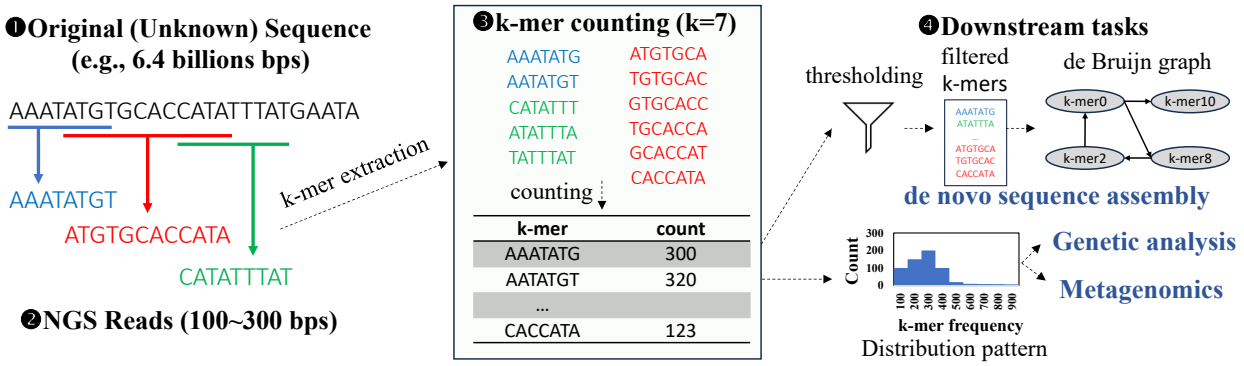


Figure 2.4: The application of k-mer counting in bioinformatic pipelines.

the matched k-mers are used to make a final decision on the originating organism for the query sequence. A popular choice is to keep a counter for each retrieved taxon label, and the taxon label with the most hits is used to classify the query sequence. See Figure 2.3 (b)for example. The reference k-mer set itself can be implemented in several ways. CLARK [2] and LMAT [5] leverage a hash table, with the k-mer pattern as the key and the taxon label as the value. Kraken [1] uses a more sophisticated data structure that is a hybrid between a hash table and a sorted list, in which k-mers that share the same "signature" are put into the same hash bucket, which is then looked up using binary search. The assumption here is that two adjacent k-mers within a query sequence are likely to share the same "signature" since they overlap by (k-1) bases and are thereby likely to get indexed into the same bucket. In theory, this improves the cache locality over purely hash table or sorted list approaches since matching the first query k-mer often brings the bucket to the cache, which will be used for the subsequent query k-mers. In our workload analysis, we find out that even with this optimization, cache performance remains poor.

2.2.2 *k*-mer counting

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 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

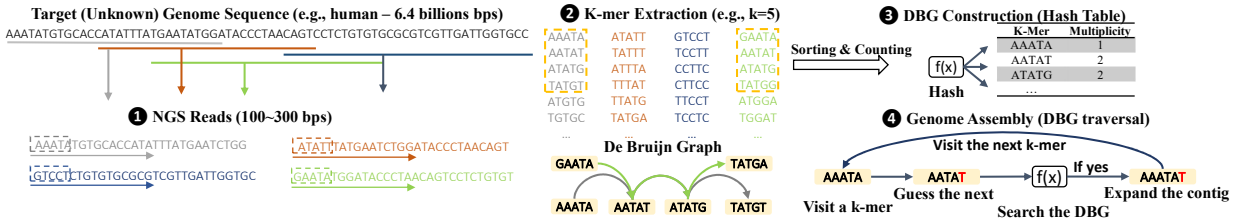


Figure 2.5: **The stages in *de novo* genome assembly using de Bruijn graph.**

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" [58], 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, represented as 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 [59].

Definition. Let $\Sigma = \{A, C, G, T\}$ 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 [60], genetic analysis [61], metagenomics [62], etc., as shown in Fig. 2.4.

2.2.3 de Bruijn Graph (DBG) Genome Assembly

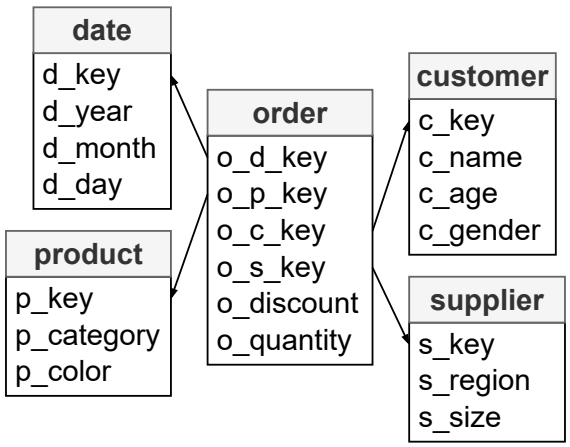
Genome sequencing is the process of determining a segment or the whole DNA sequence of an organism. *De novo* assembly is a key step of genome sequencing, where the short sequenced reads are assembled without using a reference genome [63, 64, 65, 66, 67, 68, 69, 70, 71, 72]. Currently, the most successful *de novo* assembly algorithm is based on the de Bruijn graph (DBG) algorithm. DBG is a form of directed multigraph that stores the overlapping information of k-mers (DNA subsequences of size k) extracted from DNA sequence reads. 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. DBG assembler finds a path that visits each node exactly once to assemble the DNA sequence. The DBGs are of special interest because the assembly algorithm can finish in polynomial time with respect to graph size [66].

Figure 2.5 shows the full pipeline of DBG-based genome assembly. It takes in NGS short reads sampled in the genome sequence step and then extracts k-mers from every position. The de Bruijn graph is built on the coverage relation between k-mers. Unlike a general graph, DBG follows a simple pattern where each node can only have up to four outgoing edges and four incoming edges (four possible nucleobases). Therefore, the most common data structure for DBG is a hash table, which enables efficient traversal on a forward or backward path by searching the next/previous possible k-mers[71, 67, 70]. When a k-mer appears more than once during the graph construction process, they are merged into one node and increase the count. DBG assembly can build many long sequences, which are called contigs, by traversing the Eulerian path in the DBG. DBG assemblers have many common steps, including data loading, error removal, and contig assembly.

The most time-consuming phases in a DBG assembly process are graph construction, which saves unique k-mers along with their multiplicity and connectivity from raw input reads to a hash table, and graph traversal, which traverses the graph to connect a chain of k-mers as contigs. Based on our experiments on several popular tools [71, 67, 70, 69], graph construction takes 60% of the execution time, and graph traversal for contig assembly takes 30% of execution time. Therefore, this dissertation focuses on accelerating these two phases.

2.3 Database OLAP

Online Analytic Processing (OLAP) systems are critical technologies enterprises use to unlock the potential of their vast enterprise databases. These systems employ analytic SQL queries to transform database data into visual graphs on live dashboards and generate summary reports. Many OLAP workloads are characterized by their emphasis on analyzing historical data, as compared to Online Transaction Processing (OLTP) workloads



(a) OLAP schema

Find the top 10 product categories most popular with customers in their 20s

```
SELECT
  p_category,
  SUM(o_quantity) AS popularity
FROM order, product, customer
WHERE o_p_key = p_key
  AND o_c_key = c_key
  AND c_age BETWEEN 20 AND 29
GROUP BY p_category
ORDER BY popularity DESC
LIMIT 10;
```

(b) OLAP query

Figure 2.6: An example OLAP scenario. Image credit Akhil Shekar

that generally facilitate business transactions [73]. The differences between OLAP and OLTP workloads have driven specialization in database platforms, where two separate database solutions regularly perform analytical and transactional query processing. In many cases, the archived data is stored as a read-mostly, append-only database, commonly referred to as a data warehouse [74], while the transactional database, typically much smaller, stores the latest copy of the key tables in the database.

An OLAP database typically consists of a small number (often 1) *fact* tables and many *dimension* tables. Fact tables contain historical transaction records and the dimension tables contain detailed information for specific columns in the fact table records. As an example in a shopping cart application, the fact table record may contain a record for each item that is purchased, including the id of the customer (a foreign key). A separate *customer* dimension table may contain one record for each customer, recording the unique (primary key) *customer_id* along with detailed information for that customer, such as the customer's email and address. There may be other dimension tables to record other details about the purchase, including the product description, the shipping method, etc. The fact table is generally large in size, and the dimensions tables are significantly smaller. Typical OLAP queries involve "slicing and dicing" the data along the dimensions (they do that by applying selection predicates on the dimension table columns, AKA table scan), then joining the selected dimension records with the fact table, and finally aggregating the combined results to produce a result (e.g., an ordered list of the top trending products purchased last year.)

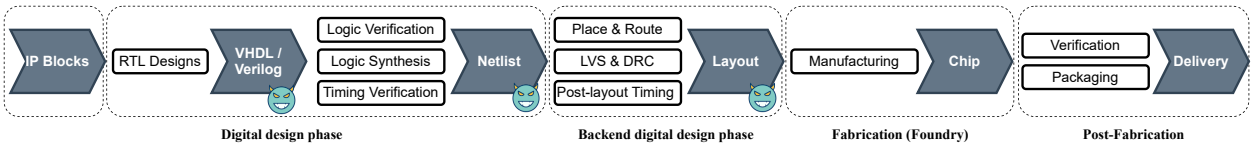


Figure 2.7: **The stages in integrated circuits design and manufacturing.**

2.4 Hardware Trojans

Fig. 2.7 captures the major milestones and deliverables of the IC design and manufacturing process [27]. Chip vendors increasingly subcontract several key steps to third parties to save costs. One popular movement is to go fabless due to the astronomically high upfront investment to set up a fabrication facility ($20 billion in 2020 for the current process node [27]). While outsourcing is economically sensible, it inevitably opens the door to an unforeseen security threat – hardware Trojans. Previous works have shown that every stage in the distributed IC supply chain is susceptible to the insertion of hardware Trojans by any entities involved. The design team might unintentionally use tainted 3rd party IP blocks or CAD tools for its RTL designs, resulting in Trojan-infected netlist or layout files(GDSII) [28, 29]. A rogue engineer in the design team can insert a Trojan directly at the RTL level [30, 31, 32]. Even if the chip specifications are correctly implemented by the design house and verified by the backend design house, the malicious foundry can tamper with the mask layout during the fabrication phase [27, 29]. Finally, the genuine IC's can be intercepted and replaced with counterfeited ones containing Trojans during the transportation at the post-fabrication phase. This work focuses on inserting a Trojan at the design or backend design phase.

Typically, a hardware Trojan consists of a trigger circuit that activates a Trojan on a specific condition and a payload circuit that causes functional perturbations, carries out catastrophic failures, or establishes a covert channel to leak private information. A good Trojan design is stealthy, which implies that the underlying malicious circuitry occupies minimal area, consumes negligible standby power, and remains dormant during its lifetime, only to be triggered by extremely rare events.

Fig. 2.8 shows a widely acknowledged classification of hardware Trojans based on action types, physical characteristics, and activation mechanisms [75, 28, 76], with the highlighted boxes indicating properties that the Trojan we design possesses. In particular, we embed our Trojan inside a memristor-based NN accelerator to *disable* some neurons from firing, allowing us to steal and *transmit* sensitive model parameters (i.e., the synaptic weights) to an adversary in the untrusted domain. The Trojan is implemented by adding *small* trigger circuits (*layout change*), separately located with the payload circuits (*loosely distributed*). It is considered *functional* instead of *parametric* because our Trojan is realized through inserting new transistors and gates, rather than tweaking the physical characteristics of existing circuits (e.g., thinning of wires,

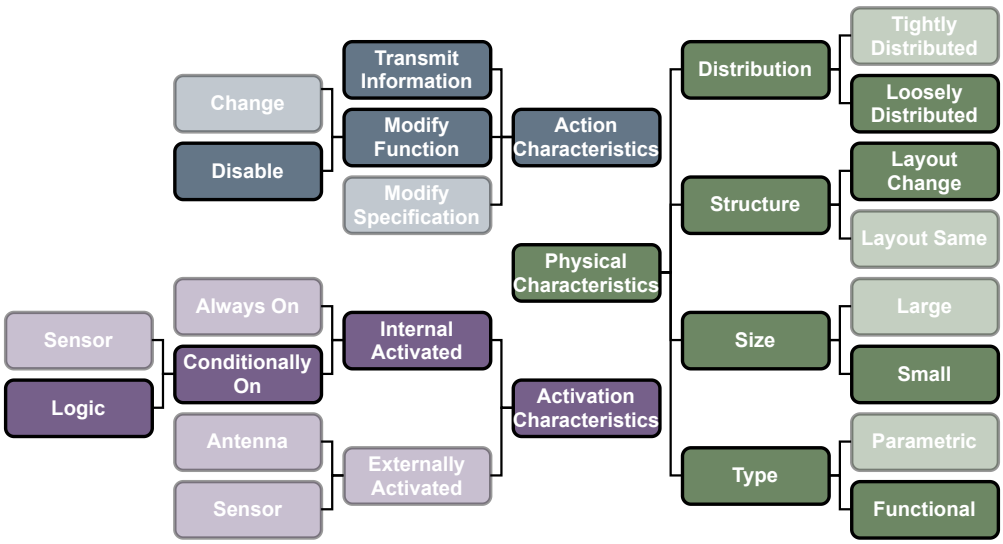


Figure 2.8: **Hardware Trojan taxonomy**

weakening of transistors, etc.) Finally, our Trojan is *conditionally* activated by monitoring specific input patterns, which change the *internal logic* state of the accelerator.

Chapter 3

Sieve: Scalable In-situ DRAM-based Accelerator Designs for Massively Parallel *k*-mer Matching

3.1 Introduction

The field of bioinformatics has enabled significant advances in human health through its contributions to precision medicine, disease surveillance, population genetics, and many other critical applications. The centerpiece of a bioinformatics pipeline is genome sequence comparison and classification, which involves aligning query sequences against reference sequences, with the goal of identifying patterns of structural similarity and divergence. While traditional sequence alignment algorithms employ computationally-intensive dynamic programming techniques, there has been a growing shift to a high-performance heuristic-based approach called *k-mer matching*, that breaks a given query sequence into a set of short subsequences of size k, which are then scanned against a reference database for hits, with the underlying assumption that biologically correlated sequences share many short lengths of exact matches. k-mer matching has been deployed in a wide array of bioinformatics tasks, including but not limited to, population genetics [4], cancer diagnosis [77], metagenomics [1, 78, 79, 2, 5, 80], bacterial typing [3], and protein classification [81]. k-mer matching may also show up in other application domains, but in this paper, we focus on bioinformatics.

The acceleration of bulk k-mer matching is of paramount importance for two major reasons. First, k-mer matching sits on the critical path of many genome analysis pipelines. Figure 3.1 shows the execution

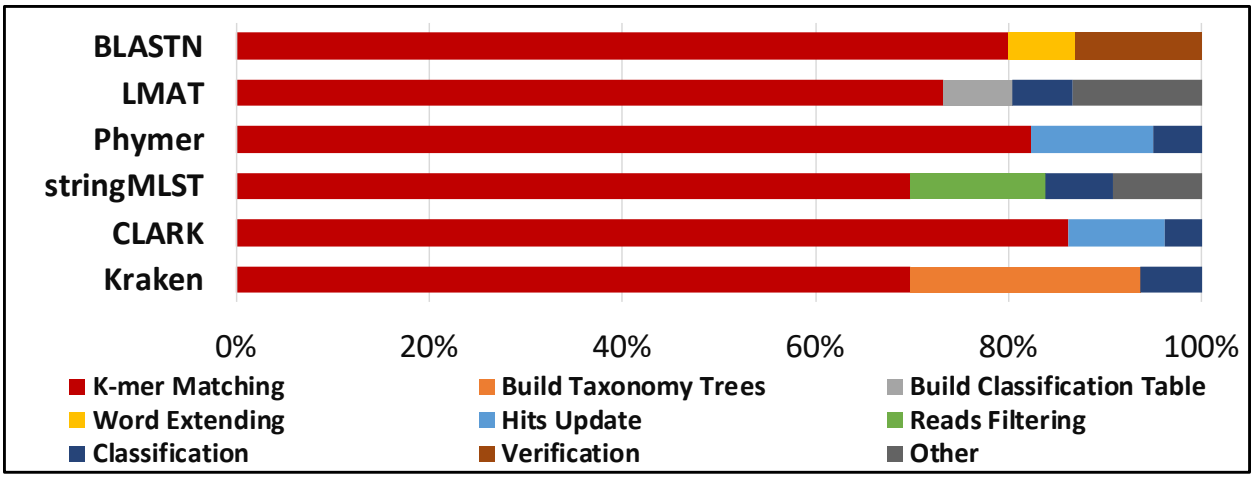


Figure 3.1: **Execution time breakdown of Kraken [1], CLARK [2], stringMLST [3], Phymer [4], LMAT [5], BLASTN [6]**

breakdown of several important bioinformatics applications that target a variety of tasks ranging from metagenomics to population genetics, and clearly, k-mer matching dominates the execution time in all applications. Second, modern sequencing technologies have been shown to generate data at a rate surpassing Moore's Law [56]. In fact, by 2025, the market share of metagenomics alone is expected to reach $1.4 billion, and the amount of data that needs to be analyzed by metagenomics pipelines is projected to surpass that of YouTube and Twitter [10]. To further exemplify the scale of data explosion and processing overhead, consider the case of precision medicine, where a patient's sample can be sequenced in roughly 48 hours on the NovaSeq instrument, producing 10 TB of microbiome and DNA/RNA data [82]. To develop personalized treatment from these samples, raw sequences are passed through, often in parallel, various metagenomics stages with k-mer matching on the critical path (e.g., ∼68 days on Kraken [1]). These tasks play a critical role in combating pandemics and treating antibiotic-resistant infections, saving billions of dollars in health care costs [82, 83].

However, despite its significance, the acceleration of k-mer matching on modern high-end computing platforms remains a challenge, due to its inherently memory-bound nature, considerably limiting downstream genome analysis tasks from realizing their full potential. In particular, k-mer matching algorithms are typically characterized by random accesses across large memory regions, leading to poor cache behavior, even on high-end servers that feature large last-level caches. The cache-unfriendliness of k-mer matching will continue to get worse with the rapid growth in the size and complexity of genomic databases, making the task a major bottleneck in modern bioinformatics pipelines. This is further exacerbated by the fact that the computation per k-mer lookup is too small to mask the high data access latency, thereby rendering existing compute-centric platforms such as multi-core CPUs and GPUs inadequate for large-scale genome analysis tasks.

Memory-centric solutions to accelerate bioinformatics applications come in a variety of flavors, but recent proposals demonstrate that *near-data* [84, 85, 86] and *in-memory processing* systems [87, 88, **?**] have promising potential to improve the efficiency of large-scale genome analysis tasks, owing to the fact that these applications are increasingly characterized by their high data movement (from memory to the processor) and low computation (within the processor) costs [13].

This work explores the design space for high-performance k-mer matching accelerators that use logic in DRAM as the basis for acceleration, including the most aggressive form of *processing-in-memory* (PIM), in-situ computing, with the goal of parallel processing of sequence data within DRAM row buffers. To this end, we propose Sieve, a set of novel **S**calable **i**n-situ DRAM-bas**e**d accelerator designs for massi**v**ely parallel k-m**e**r matching. Specifically, we offer three separate designs: Sieve Type-1, Type-2, and Type-3. Each architecture incrementally adds extra hardware complexity to unlock more performance benefits. Note that, although our approach involves modifying conventional DRAM organization, we do not propose change conventional DRAM; our goal is to only leverage DRAM technology to build a new accelerator. Ultimately, the value of the accelerator will determine whether a new DRAM-based chip is worth the design and manufacturing effort.

The advantage of in-situ computing is that the bandwidth at the row buffer is six orders of magnitude larger than that at the CPU, while the energy for data access is three orders of magnitude lower [89, 90]. However, in-situ computing also introduces several key challenges. First, in-situ acceleration necessarily requires the tight integration of processing logic with core DRAM components, which has been shown to result in prohibitively high area overheads [87, **?**]. In fact, even a highly area-efficient state-of-the-art in-situ accelerator is only half as dense as regular DRAM [87]. However, bioinformatics applications typically favor accelerators with larger memory capacity due to their ability to accommodate the ever-increasing DNA datasets that need to be analyzed within short time budgets. Second, existing in-situ approaches [87, 88] rely on multi-row activation and row-wise data mapping to perform bulk Boolean operations of data within row buffers, resulting in substantial loss of throughput and efficiency [**?**]. Finally, to capitalize on the performance benefit of in-situ computing for k-mer matching, it is imperative that the accelerator is provisioned with an efficient k-mer indexing scheme that avoids query broadcasting, and a mechanism to quickly locate and transfer payloads (e.g., genome taxon records).

Key Contributions. The distinguishing feature of Sieve is the placement of reference k-mers vertically along the bitlines of DRAM chips and subsequently utilizing sequential single-row activation rather than the multi-row activation proposed in prior works, to look up queries against thousands of reference k-mers simultaneously. The column-wise placement of k-mers further allows us to employ a novel Early Termination Mechanism (ETM) that interrupts further row activation upon the successful detection of a k-mer mismatch, thereby considerably alleviating the latency and energy overheads due to serial row activation. To the best of

our knowledge, this is the first work to introduce and showcase the effectiveness of such a column-wise data mapping scheme for k-mer matching with early termination, substantially advancing the state-of-the-art in terms of both throughput and efficiency.

By exploiting the fact that matching individual k-mers is relatively less complex than most other conventional PIM tasks such as graph processing, in this work, we design a specialized circuit for k-mer matching, with the goal of minimizing the associated hardware cost. We then meticulously explore the design space of in-situ PIM-based accelerators by placing such custom logic at different levels of the DRAM hierarchy from the chip I/O interface (Type-1) to the subarray level (Type-2/3), with a detailed analysis of the performance-area-complexity trade-offs, and a discussion of system integration issues, deployment models, and thermal concerns.

We compare each Sieve design with state-of-the-art k-mer-matching implementations on CPU and GPU, and perform rigorous sensitivity analyses to demonstrate their effectiveness. We show that the processing power of Sieve scales *linearly* with respect to its storage capacity, considerably enhancing the performance of modern genome analysis pipelines. Our most aggressive design provides an average speedup of 210X/35X and an average energy savings of 35X/71X over conventional multi-core-CPU/GPU baselines

3.2 Motivation

In this section, we explain why memory bottlenecks the overall k-mer matching execution, and we address the main challenge of designing in-situ k-mer matching accelerators, namely integrating logic into DRAM dies with low hardware overhead. We propose three separate Sieve designs to combat this issue. We then identify the key limitations of prior in-situ work when adapted for k-mer matching and motivate our novel data layout and pattern matching mechanisms. Finally, we introduce an Early Termination Mechanism (ETM) to further optimize Sieve by exploiting characteristics of real-world sequence datasets.

Memory Is the Bottleneck for *k*-mer Matching. Real-world k-mer matching applications expose limited cache locality. For sequence classifiers that store reference k-mers in a hash table, accessing a hash table generates a large number of cache misses due to the linked list traversal or repeated hashes (to resolve hash collision). While a hash table/sorted list hybrid can provide better locality, since the k-mer bucket can be fetched into the cache from the previous k-mer lookup, using Kraken and its supplied datasets, we discover that only 8% of consecutive k-mers index into the same bucket, resulting in new buckets fetched repeatedly from memory to serve requests. k-mer matching also benefits from finer-grained memory access—k-mer records are typically around 12 bytes [1], while each memory access retrieves a cache line of data, which usually serves only one request due to poor locality, resulting in waste of bandwidth and energy. Finally,

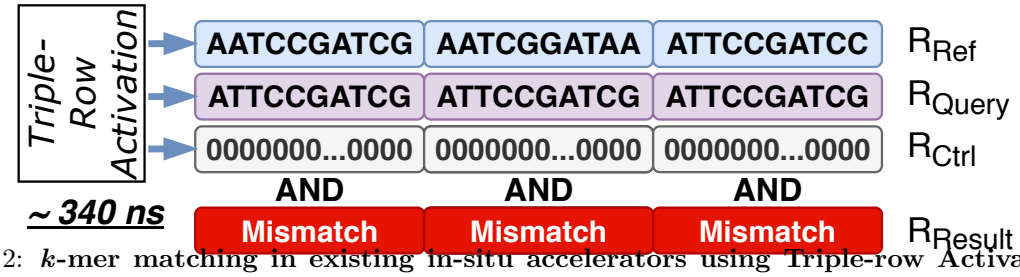


Figure 3.2: ***k*-mer matching in existing in-situ accelerators using Triple-row Activation and horizontal data layout.**

Figure 3.3: ***k*-mer matching in Sieve using Single-Row Activation and vertical data layout.**

the computational intensity of k-mer matching is too little to mask extended data access latency. While retrieving k-mers from a database takes many cycles due to cache misses, updating counters for matched k-mers is trivially inexpensive, amplifying the effects of the memory wall [13].

DRAM Overhead Concerns. While in-situ accelerators can provide dramatic performance gains for memory-intensive applications, building them with reasonable area overhead is difficult [87, **?**]. The sense amplifiers in row buffers are laid out in a pitch-matched manner, and the DRAM layout is carefully optimized to provide high storage density, fitting additional logic into the row buffer in a minimally invasive way is non-trivial. Moreover, since the number of metal layers of a DRAM process is substantially smaller than that of the logic process, building complex logic with a DRAM process incurs significant interconnect overhead [87, **?**].

We design a set of core k-mer matching operations for Sieve using simple Boolean logic. Sieve has very little hardware overhead compared to other PIM architectures, because k-mer matching, which is mainly accomplished by exact pattern matching, can be supported by a minimal set of Boolean logic.

Trade-offs of Different Sieve Designs. To explore optimal Sieve designs, we compare the placement of custom k-mer matching logic at three different levels in the DRAM hierarchy: from the I/O interface of the DRAM chips (Sieve Type-1) to the local row buffer of each subarray (Sieve Type-3), and Type-2 as the middle ground where several subarrays share one k-mer matching unit. Recall that a DRAM bank's transistor layout is highly optimized for storage, and inserting extra logic, however minimal, requires significant redesign effort. Type-1, illustrated in Figure 3.10, keeps the bank layout intact, and thus is the least intrusive design. However, it suffers from the lowest parallelism and the highest latency, because the comparison is restricted to a column of bits rather than the entire row. Sieve Type-2 increases parallelism and energy efficiency

over Type-1 by accessing a row of bits. Type-3 leverages subarray-level parallelism (SALP) [42], providing the highest performance potential, but it comes at the cost of the highest design complexity and hardware overhead.

Novel Data Layout and Pattern Matching Mechanism. We show that our column-wise k-mer data layout and row-wise matching mechanism, combined with *early termination* outperforms prior in-situ accelerators that rely on multi-row activation and conventional row-wise mapping. The majority of the k-mer matching workload is exact pattern matching, which can be performed using bulk bitwise XNOR between two operand DRAM rows. The prior arts such as Ambit and DRISA implement XNOR operation by first ANDing two rows along with a third control row (populated with 1s or 0s), and send the results to an additional logic. In the following analysis, we only consider the timing delay of the AND operation to give advantage to the previous in-situ PIM work. Ambit [88] is used as a baseline. Both Ambit and *1T1C-based* DRISA [87] are inspired by the same work [91] for in-situ AND procedure. Thus, their performance for k-mer pattern matching is similar. Ambit performs bulk bitwise AND in reserved DRAM rows (see Figure 3.2). Assuming a DNA base is encoded with two bits (by NCBI standard [92]), a common k value of 31, and a typical DRAM row width of 8192 bits, then each row fits 128 k-mer patterns if k-mers are stored in a row-wise manner. To search a query against a group of references, Ambit first copies 128 reference patterns from the data region to R_{Ref}. It then makes 128 copies of the same query in R_{Query}. Since the target operation is AND, the control row (R_{Ctrl}) is populated with 0s (copied from a preset row). Next, a triple-row activation is performed on R_{Ref}, R_{Query}, and R_{Ctrl}. Finally, the result bits are copied to another row R_{Result}. One row-wide AND takes 8 row activations and 4 precharge commands from setting up to completion, which is $8 \times tRAS + 4 \times tRP =\sim 340ns$ for a typical DRAM chip.

In contrast to these approaches, ComputeDRAM [93] enables in-memory computation in commodity DRAMs, without the need for integrating any additional circuitry. The key to this approach is the fact that issuing a constraint-violating sequence of DRAM commands in rapid succession leaves multiple rows open simultaneously, allowing row-wide copy, logical AND, and logical OR operations to be performed via bit line charge sharing, essentially free of hardware cost.

While all of these approaches can be leveraged to perform k-mer matching, our analysis suggests that significant gains in performance and energy efficiency can be achieved by employing the column-major approach we propose in this work, that not only eliminates the need for multi-row activation, but also enables a synergistic early termination mechanism that inhibits further row activations upon finding a match.

More specifically, Sieve does not compare a full-length query k-mer against a set of full-length reference k-mers at once. Instead, it compares a query with a more extensive set of references in a shorter time window ($1 \times tRAS + 1 \times tRP =\sim 50ns$), but progresses only one bit at a time (see Figure 3.3). Reference bits in Sieve

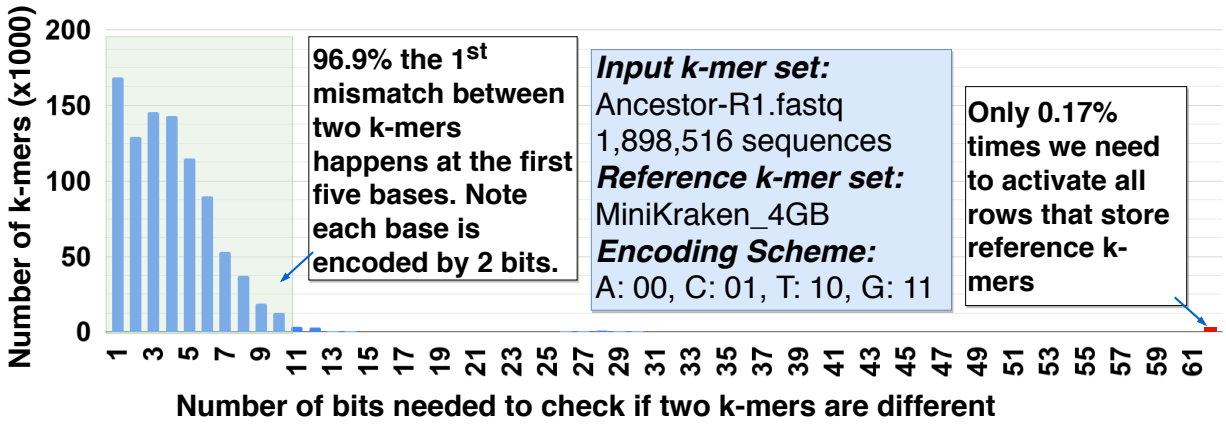


Figure 3.4: **Characterization of mismatches between k-mers.**

are laid out column-wise, along bitlines. Thus, a single row activation transfers 8K bits into the matchers embedded in row buffers for comparison. Each matcher has a one-bit latch to keep track of the matching result. The next row is activated, and a new batch of reference bits is compared, until ETM (introduced next) interrupts when all latches return zero.

Processing only one bit at a time does not hurt Sieve's performance, because it leverages parallelism across the rows; i.e., it performs 8K comparisons at once. The vertical data layout greatly expands the initial search space (128 reference k-mers to 8192 reference k-mers), and our *early termination mechanism* (ETM) quickly eliminates most of the candidates after just a few row activations. Besides the latency reduction for each row-wide pattern matching by adopting single-row activation (∼340 ns to ∼50 ns), Sieve also reduces activation energy, since raising each additional wordline increases the activation energy by 22% [88]. Thus, even if the same data mapping strategy is applied, the multi-row activation-based approach is still slower and less energy efficient than Sieve simply because of the internal data movement. Note that the internal data movements associated with multi-row activation is unavoidable, because the operand rows have to be copied to the designated area. Furthermore, arbitrarily activating three rows inside the DRAM requires a prohibitively large decoder (possibly over 200% area overhead [87]), and activating more than one row could potentially destroy the original values.

The Motivation for Early Termination. Activating consecutive rows in the same bank results in highly unfavorable DRAM access patterns that are characterized by long delays (due to more row cycles) and high energy costs (row opening dominates DRAM energy consumption [94]).

We identify a novel optimization opportunity that exploits the concept of the *Expected Shared Prefix* (ESP), which describes the first mismatch location between two random sequences. On average, for DNA sequences between 1k and 16k bases, the first mismatch is known to occur between the sixth and the eighth base [95]. The ESP is even smaller than six for short k-mers, as shown in in Figure 3.4. For random k-mers

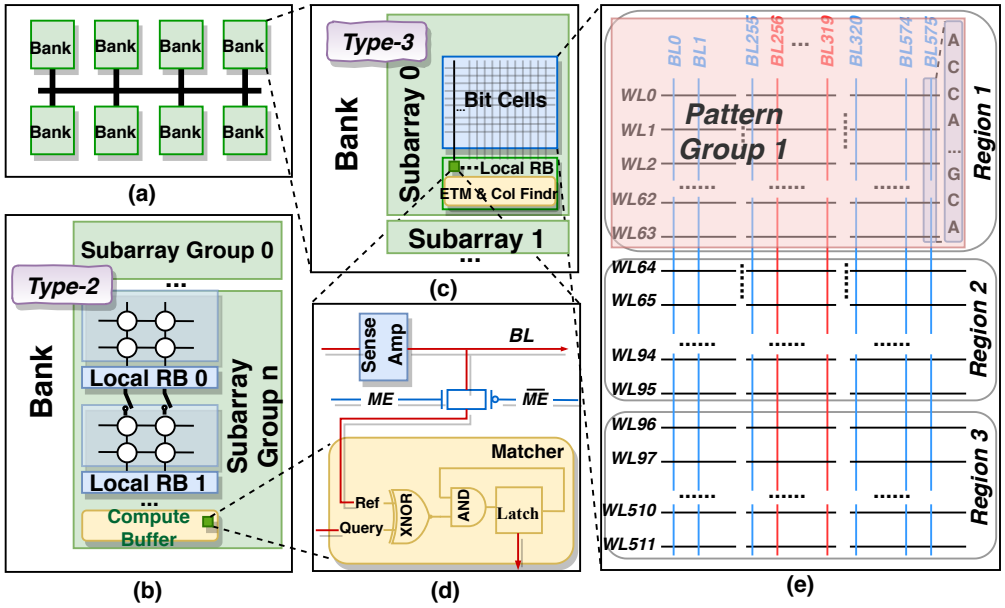


Figure 3.5: **Sieve Overview. (a) DRAM banks. (b) Type-2 Zoom-in. Subarray group facilitates inter-subarray data copy, and a compute buffer is added for each subarray group which has the matcher circuits. (c) Type-3 Zoom-in. Similar to Type-2 but the matchers reside in the local row buffers. (d) Matcher. (e) Data layout of subarray. Each subarray is partitioned into three regions for storing *k*-mer pattern groups, payload offsets, and payloads.**

extracted from metagenomics reads, when matched against reference k-mers, 97% of the first mismatch can be found within the first five bases (first 10 bits if each base is encoded by two bits).

3.3 Architecture

This section describes the three Sieve designs. We introduce Types-2 and 3 first, as they exploit greater parallelism, and follow it up with Type-1 due to difference in design details.

3.3.1 Sieve Type-2 and Type-3

Figures 3.5 (b) and (c) show the functional block diagrams of Type-2/3. They differ mainly in the placement of the add-on logic (e.g., matching circuitry) at the bank vs. subarray level, but share the same data mapping scheme.

Data Layout. k-mer patterns are encoded in binary (A: 00, C: 01, G: 10, T: 11) and transposed onto bitlines, for column-wise placement, as described in the previous section. Bit cells within each subarray are divided into three regions (Figure 3.5 (e)). However, we note that no physical modification is made to the bit cells. Region-1 stores the interleaved reference and query k-mers. Region-2 stores the offsets to the starting address of payloads (one for each reference k-mer), allowing us to precisely locate the payloads. Region-3

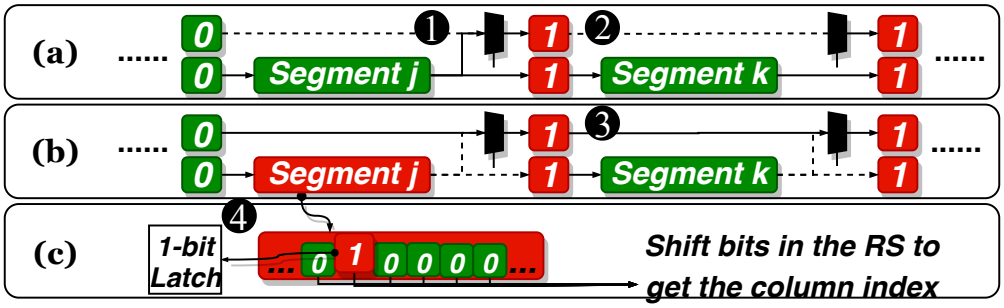


Figure 3.6: **Column Finder in Type-2/3. Segments with k-mer hits are shown in red, otherwise green.**

stores the actual payloads such as taxon labels. Data in Region-2/3 is stored in conventional row-major format. The main motivation to co-locate patterns and payloads is to minimize contention and achieve higher levels of parallelism. If patterns are densely packed into several dedicated banks/subarrays, all matching requests will be routed to them, creating bank access contention and serializing such requests.

Region-1 is further broken down into smaller pattern groups and a batch of 64 (different) query k-mers are replicated in each pattern group in the middle (red in Figure 3.5(e)). This is because the transmission delay of long wires inside DRAM chips prevents us from broadcasting a query bit to all matchers (discussed next) during one DRAM row cycle. All pattern groups in a subarray work in the lockstep manner. The exact size of a pattern group is equivalent to the number of matchers that a query bit can reach in one DRAM row cycle. In this example (DDR3_micron_32M_8B_x4_sg125), it happens to be 576 (*512 reference k-mers + 64 query k-mers*). The number of query k-mers per batch is determined by the chip's prefetch size. In this example, a chip with a prefetch size of 8 bytes writes 64 bits with a single command. A chip with smaller (larger) prefetch size has smaller (larger) batch size. After a batch of query k-mers finishes matching in a subarray, they are replaced by a new batch. The total number of write commands needed to replace a batch of 64 k-mers can be computed as *(# of pattern groups / subarray)* $\times$ *(k* $\times$ *2)*.

Matcher. We enhance each sense amplifier in a row buffer with a matcher shown in Figure 3.5 (d). The matcher of Type-2/3 is made of an XNOR gate, an AND gate, and a one-bit latch. The XNOR gate checks if the reference bit and the query bit at the current base are equal. The bit latch stores the result of the XNOR operation, indicating if a reference and a query have been matched exactly up until the current base. The value in each bit latch is set to 1 initially (default to match). The AND gate compares the previous matching result stored in the bit latch with the current result from the XNOR gate and updates the bit latch accordingly, capturing the running outcome bit-by-bit. Finally, we allow the matcher to be bypassed or engaged by toggling the Match Enable signal.

When a row is opened, both query and reference bits are sent to sense amplifiers. A subarray controller [87] (sCtrl) then selects which query to process among the 64 queries in the subarray. Each pattern group has a

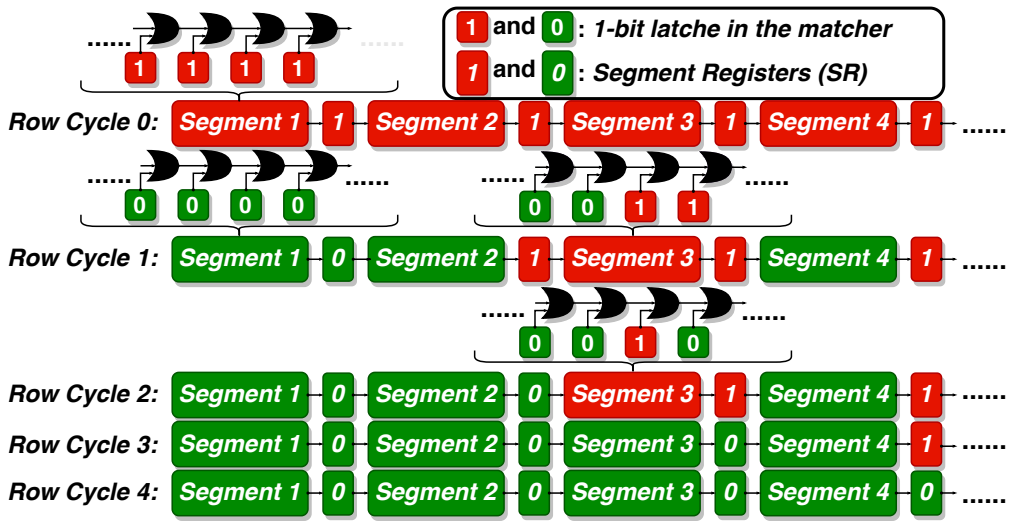


Figure 3.7: **ETM in Type-2/3.**

1-bit shared bus connecting all matchers. The selected query bit is distributed to all matchers in a pattern group through this shared bus.

Early Termination Module (ETM). The ETM interrupts further row activation by checking if the entire row of latches is storing zeros. The k-mer matching process continues if at least one latch stores 1. The natural way is to OR the whole row of latches. However, the challenge of this approach is that each OR gate adds to the latency, and during one DRAM row cycle, only a small fraction of result latches can propagate their results through OR gates. We propose a solution that breaks the row of latches into segments and propagates partial results in a *pipelined* fashion. (shown in Figure 3.7). One segment register (SR) is inserted for every 256 latches to implement the pipeline. During one DRAM row cycle, each segment takes the value from the previous SR, ORs it with all its latches, and outputs the value to the next SR. Notice that in Figure 3.7, although at row cycle 3, all latches store zeros, the last SR still holds 1. An extra cycle is needed to flush the result

Column Finder (CF). Unless interrupted by the ETM, the row activation continues until all bases of a query are checked. If a query is previously matched to a reference, one and only one latch in a row buffer stores one. The Column Finder identifies the column (bitline) that is connected to that latch. The column numbers are needed to retrieve offsets, and subsequently, payloads. Our solution is to shift a row of latched bits until we find a one. The challenge of this approach is to design a shifter with reasonable hardware cost and latency. In the worst case, where the matched column (reference k-mer) is located at the end of the row, the CF needs to shift an entire row of latched bits. We propose a pipelined, two-level shifter solution for CF. Figure 3.6 illustrates this. The CF circuits are re-purposed mainly from those of the ETM. For each ETM segment, a MUX (1) and a 1-bit Backup Segment Register (BSR) (2) are added (Figure 3.6 (a)). BSRs

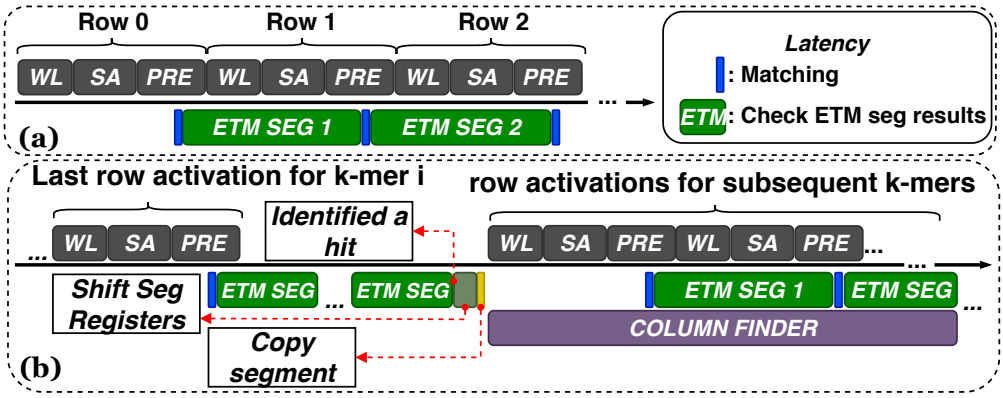


Figure 3.8: **Type-3 Timing Analysis. WL, SA, and PRE indicate latencies associated with raising the wordlines, enabling sense amplifiers and precharging the rows. (a) ETM and matchers operations overlap with row opening. (b) ETM is on the critical path only when there is a hit, as it needs extra cycles to identify the hit. Then the BSRs are shifted, followed by a copy into the RS. CF operates in parallel with row opening and ETM for the next k-mer.**

and SRs maintain the same values and are updated simultaneously during the ETM operation. Zero in a BSR means that its associated segment does not contain a match, and one implies it does. Further, we add another set of bit latches called the Reserved Segment (RS) shown in Figure 3.6 (c), which includes the same amount of 1-bit latches and OR gates as a segment.

For Column Finder, the BSRs are first shifted until we find a one, to narrow down the appropriate segment that contains a match (3) in Figure 3.6 (b)). We then copy this segment over to the Reserved Segment (RS) where the final round of shifting happens (4). From this point on, all ETM segments are freed to support the pattern matching for the next k-mer, while the CF works in the background to retrieve the column number (see Figure 3.8 (b)). The shifting of bits in RS is overlapped with the matching of the subsequent k-mer. We point out two details here. First, after the last row activation for a given query k-mer finishes, ETM takes up to 256 DRAM row cycles to flush the pipeline in the worst case, when the one is at the very end. During this time, no new row activation is issued, and the CF operation is stalled until ETM completes. Second, note that each k-mer hit takes up to 4800 DRAM cycles, while the CF operation takes up to 1032 DRAM cycles in the worst-case scenario. Therefore, we observe no contention at the CF, even when there are two consecutive hits in the same subarray.

Sieve Type-2. While Type-2 retains most of the high-level design from Type-3 (ETM, data mapping, matching circuits, etc.), it differs in one key aspect – instead of integrating logic to all subarrays at the local row buffer level, logic is added to a *subarray group* – a subset of adjacent subarrays within a bank (e.g., 1/2, 1/4, 1/8 of subarrays) connected through high bandwidth links (isolation transistors). Each subarray group is equipped with a *compute buffer*, which retains much of the capabilities (k-mer matching, ETM, and column finding) of a local row buffer in Type-3 without its sense amplifiers. Unlike type-3, where k-mer matching is

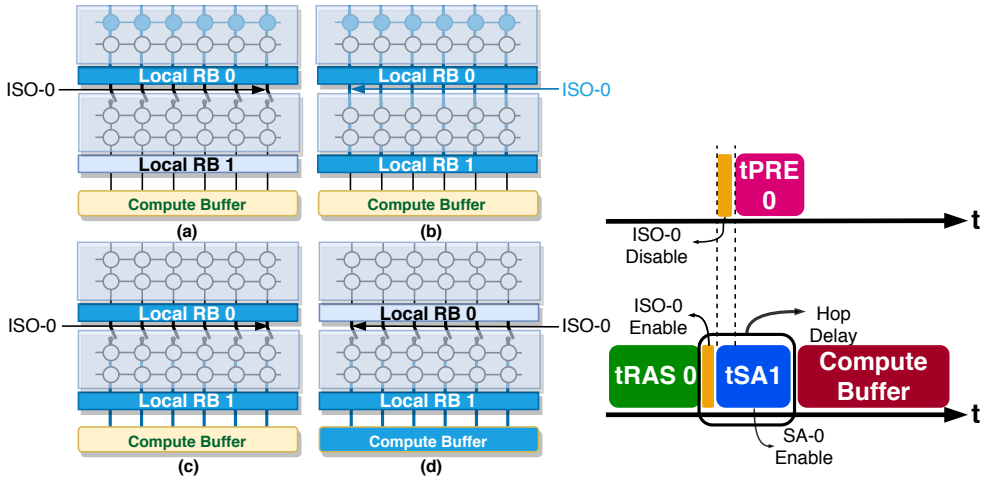


Figure 3.9: **Row-wide data copy across subarrays.**

performed locally at each individual subarray, Type-2 processes k-mer matching inside the compute buffer regardless of the target subarray query k-mers get dispatched to. This involves transferring a row of bits across subarrays to reach the compute buffer at the bottom of the subarray group. To enable fast row copy across subarrays, we leverage LISA [96], albeit adapted to the folded-bitline architecture that Sieve is built upon. We validate the feasibility of our design with a detailed circuit-level SPICE simulation.

Figure 3.9 illustrates the process of transferring a row from the source subarray to its compute buffer – (a) the DRAM row in the subarray 0 is activated and the data is latched onto its local sense amplifiers, (b) when the bitlines of subarray 0 are fully driven, the links between the subarray 0 and subarray 1 are enabled. Due to charge sharing between the bitlines of subarrays 0 and 1, the local sense amplifiers in the subarray 1 senses the voltage difference between the bitlines and amplifies it further, as a result of which, (c) local sense amplifiers in both subarrays 0 and 1 start driving their bitlines to the same voltage levels, and finally, (d) when both sets of bitlines in subarrays 0 and 1 reach their fully driven states, the isolation transistors between them are disconnected and the local sense amplifiers in the subarray 0 are precharged. The process is repeated until the data reaches the computed buffer. Note that – (1) only two sets of local sense amplifiers are enabled at any time in a bank, and (2) as validated in our Spice simulation, the latency of activating the subsequent sense amplifiers (tSA in Figure 3.9 is much smaller ($\sim$ 8X) than activating the ones of the source subarray (tRAS). The latency for one row to cross a subarray (except for the first one) is referred to as "hop delay" which consists of enabling the isolation transistors (link) and the activation of the sense amplifiers.

***k*-mer Matching Walkthrough.** We use Type-3 as an example to illustrate the k-mer matching process. Once a row is selected for activation, both the query and the reference bits are sent to the local row buffer for comparison using the mechanisms described above. The ETM checks all segments and propagates the values of Segment Registers (SRs) to determine if a match is found.Once a match is found, the payload associated

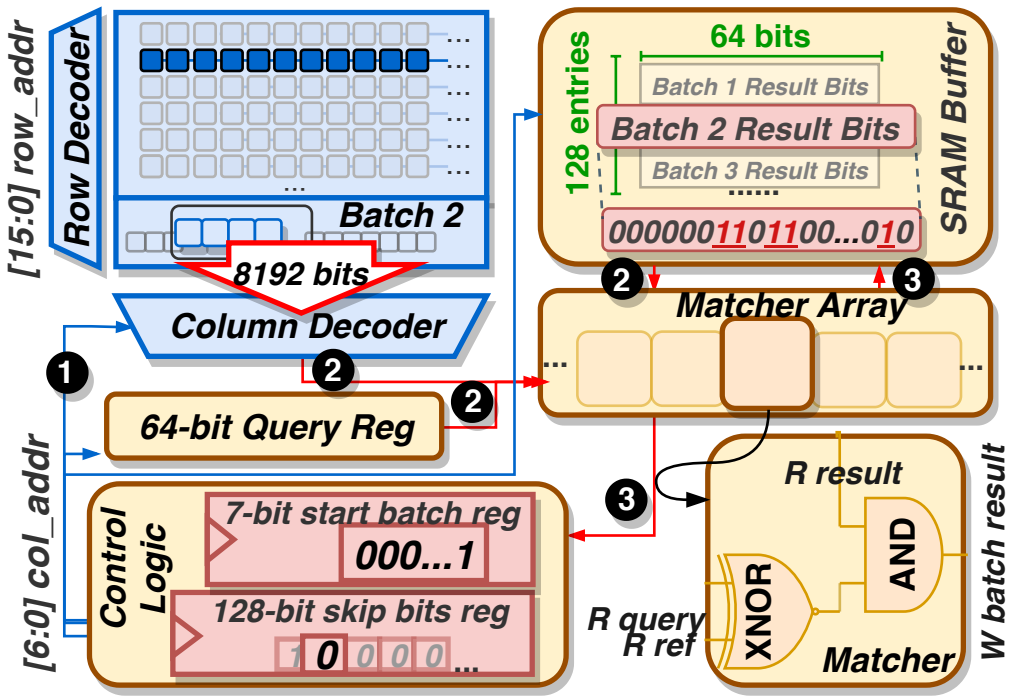


Figure 3.10: **Sieve Type-1. A query k-mer is sent to the Query Register, and a row activation is issued. 1. The controller logic uses the column address to select a batch and indexes into the SRAM Buffer to get the batch result bits entry. 2: The query bit, the reference bits, and the result bits are sent to the Matcher Array. 3: Matchers write back to the result bits entry stored in the SRAM Buffer.**

with that k-mer pattern is retrieved as follows. The CF first determines the segment number by shifting all BSRs. It then gets the column index by shifting all 1-bit latches in that segment until the one is found. The column number is calculated as *segment_number* × *(# of columns / segment)* + *column_index* and sent to subarray controller to index into the payload address offsets.

3.3.2 Sieve Type-1

Sieve Type-1 is not a quintessential in-situ architecture, due to the lack of processing unit embedded in row buffers. However, Type-1 preserves the overall high-level ideas, such as the data layout, ETM, and the matching unit. In addition, Type-1 is the least intrusive implementation of Sieve because it does not change the physical layout of DRAM banks. The bank I/O width is 64 bits, and each row is 8192 bits. Thus, a row is divided into 128 *batches*. A batch is a set of bits retrieved by a DRAM read burst of a read command. Batch size varies depending on the column width, which can be 32, 64, or 128 bits. Next, we introduce each component of Type-1.

SRAM Buffer (SB). SB stores the match result bits, organized in a 2D array. The number of entries is equal to the number of batches, and the entry width is the batch size. Before matching, all batch result bits

are preset to one, and are updated as the matching progresses, again capturing the running match outcome. Figure 3.10 highlights the result of batch two, where zero indicates a mismatch.

Matcher Array (MA). MA consists of 64 matching units. It compares a query bit with the reference bit using an XNOR gate, and updates (writes back) the result bit by ANDing the match result bit stored in SB with the output from XNOR.

Skip Bits Register (SkBR). SkBR is used for ETM. It contains one bit for each batch indicating if we need to process the current batch. All bits in SkBR are preset to one. As the matching progresses, more and more bits in SkBR is set to zero, meaning more and more batches will be skipped. Without SkBR, each row activation is followed by 128 batch comparisons. Since most comparisons result in mismatches, SkBR leads to significant energy and latency reduction.

Start Batch Register (StBR). StBR reduces processing time further. Due to the ETM, Type-1 checks the skip bits to find proper batches to send to the MA. The search time is one DRAM cycle per skip bit. In the worst case where only the last batch is valid, 127 DRAM cycles are wasted to check all the previous skip bits. With the help of the StBR, whose value points to the first batch that needs to be processed, Type-1 can quickly determine the first batch to open.

Column Finder and Payload Retrieval. The control logic first checks the skip bits to locate the batches that contain a one, given the one-to-one mapping between batches and skip bits. A small shifter is applied to get the index of the matched column in the batch. The column number is calculated as (batch index) * (batch size) + (column index), and is then used by the control logic to get offsets and payload.

3.3.3 System Integration

We consider both Dual-Inline Memory Module (DIMM), and PCIe form factors for integrating Sieve into a host. While PCIe incurs extra communication overhead due to packet generation, DIMM suffers from limited power supply. A typical DDR4 DIMM provides around 0.37 Watt/GB [97] of power delivery and 25 GB/s of bandwidth, which is sufficient for Type-1. To satisfy the bandwidth and power requirement, Type-2 needs at least PCIe 3.0 with 8 lanes, and Type-3 needs at least PCIe 4.0 with 16 lanes.

We use a 32 GB Type-2 Sieve to illustrate how Sieve communicates with the host using a PCIe interconnect. Unlike Type-1, which communicates with the host on individual k-mer requests, Type-2/3 uses a packet-based protocol that delivers hundreds of k-mer requests per PCIe packet. A PCIe Type-2/3 accelerator maintains a (*PCIe Input Queue*) and a (*PCIe Out Queue*) for sending/receiving PCIe packets, and a *response ready queue (RRQ)* to hold serviced k-mer requests. The CPU scans the query sequences to generate k-mers, and for each k-mer, it makes a 12-byte *request* that contains the pattern, sequence ID, destination subarray ID,

and other header information. Each PCIe packet contains 340 requests, assuming 4 KB PCIe packet payload size. Each Sieve bank buffers 64 requests. To fully saturate the capacity of a 32 GB Sieve, the depth of the PCIe queue is set to 24 (24 *PCIe packets* $\times$ 340 *requests / packet* $\approx$ 16 *ranks* $\times$ *8 banks / rank* $\times$ 64 *requests / bank*). Sieve removes the PCIe packets from *PCIe Input Queue*, unpacks them, and distributes requests to the target banks. A finished request gets moved to the *RRQ*. Once the *RRQ* is full, a batch of PCIe packets is moved to the *PCIe Out Queue*. Sieve sends an interrupt to the CPU if the packets are waiting in the *PCIe Out Queue* or if there are empty slots in the *PCIe Input Queue*.

The entire space of Sieve is memory-mapped to host as a noncacheable memory region, avoiding virtual memory translation and cache coherence management. Regardless of configuration (DIMM or PCIe), a program interacts with the Sieve device through the Sieve API, which supports calls to transpose a conventional database into the format needed for column-wise access (this can be stored for later use and is thus a one-time cost); load a database into the Sieve device; and make k-mer queries. The API implementation requires a user-level library and an associated kernel module or driver to interface to the Sieve hardware. The exact API and implementation are a subject to future work. k-mer databases are relatively stable over time, so once a database is loaded into the Sieve device, it can be used for long periods of time, until the user wishes to change a database. The same databases are often standard within the genomics community, high reuse can be expected to amortize the cost of database loading.

3.3.4 *k*-mer to Subarray Mapping

Without an appropriate mapping scheme, each query needs to be broadcast across all regions of the accelerator. A naïve mapping scheme would involve looking up an index table that maps queries to banks (Type-1) or subarrays (Type-2/3). Such a scheme would quickly stop scaling, as the size of such an index table increases exponentially with the length of a k-mer. We design an efficient and a scalable indexing scheme, wherein the size of the index table scales linearly with the main memory capacity rather than the length of a k-mer. More specifically, the reference k-mers in each subarray are sorted alphanumerically from left to right, and then each entry in our index table maintains an 8-byte subarray ID along with the integer values of the first and the last k-mers at the respective subarray (identified by the index). Upon receiving a matching request, Sieve first converts the query k-mer to its integer representation, and consults the index table to select the bank/subarray that contains a match. While Type-2/3 exploit different levels of parallelism, they share the same indexing scheme, i.e., if Type-2 only provides the bank address to our indexing scheme, a query needs to be checked against every subarray in that bank. The size of the index table stays well under 2 MB even for Type-2/3 with 500 GB of capacity, which is reasonable for a dedicated bioinformatics workstation.

Table 3.1: Workstation Configuration

CPU Model	Intel(R) Xeon(R) E5-2658 v4
Core/ Thread/ Frequency	14/ 24/ 2.30 - 2.80 (GHz)
L1 (KB)/L2 (KB)/L3 (MB) $	32 / 256 / 35
Main Memory	DDR4-2400MHz
Memory Organization	32GB / 2 Channels / 2 Ranks
GPU Model	Pascal NVIDIA Titan X

Table 3.2: Query Sequence Summary

Query files	**# Sequences**	**Seq Length**	**# *k*-mer**
HiSeq_Accuracy.fa (HA)	1.0e4 sequences	92 bases	6.2e4 *k*-mers
MiSeq_Accuracy.fa (MA)	1.0e4 sequences	157 bases	1.27e6 *k*-mers
simBA5_Accuracy.fa (SA)	1.0e4 sequences	100 bases	7.0e5 *k*-mers
HiSeq_Timing.fa (HT)	1.0e8 sequences	92 bases	6.2e8 *k*-mers
MiSeq_Timing.fa (MT)	1.0e8 sequences	157 bases	1.27e10 *k*-mers
simBA5_Timing.fa (ST)	1.0e8 sequences	100 bases	7.0e9 *k*-mers

3.3.5 Sieve: Putting it all together

For Type-2/3, the host reads the input query sequences and extracts k-mer patterns. For each k-mer, the k-mer to subarray index table is consulted to locate the destination subarray, and a k-mer request is made, as described in Section 3.3.3. A number of k-mer requests that need to be sent to the same subarray is grouped into one "batch". The exact number of k-mer requests per batch is equal to the number of query k-mers in a pattern group (64 in our example). These query batches are placed in a buffer, ready to be shipped to the PCIe device buffer by DMA. PCIe bundles several such batches into one PCIe packet (also described in Section 3.3.3) sent to the Sieve device. Sieve dispatches each batch of query k-mers to the destination subarray, and replaces an already processed query k-mer batch with a new (to-be-processed) batch.

Individual k-mer requests in the same batch potentially complete at different times as (1) they get issued out-of-order (as soon as their bank/subarray becomes available), and (2) each request may involve checking a different number of rows. Thus, response packets may arrive out-of-order at the host, where their sequence IDs and payloads are examined, as part of a post-processing step. Upon completion of all k-mer requests for a given sequence, the accumulated payloads are fed into a classification step. Note that there is no additional reordering step required at the host end as the accumulated payloads are typically used to build a histogram of taxons for a given DNA sequence.

3.4 Methodology

Workloads. We use Kraken2 [98] and CLARK [2] for the CPU baseline, and cuCLARK [99] for the GPU baseline. We use MiniKraken_4GB (4GB), MiniKraken_8GB (8GB), NCBI Bacteria (2785 genomes 6.24GB). The query sequences are summarized in Table 3.2, and K is set to 31.

Baseline Performance Modeling. We report our workstation configurations in Table 3.1. The GPU baseline is idealized because (1) the energy and latency of data transfer from host to GPU are not included, and (2) the on-board memory is assumed to always be large enough to avoid running each query multiple times. The baseline DRAM energy consumption is estimated by feeding memory traces associated with k-mer matching functions, obtained using Hopscotch [100], to DRAMSim2 configured to match our workstation. The CPU energy is measured using the Intel PMC-power tool [101], then scaled down by 30% to exclude the interference from other system components, and the GPU energy is measured using NVIDIA Visual Profiler [102] as it is performed in [103] to characterise the multi-GPU inference server energy efficiency and scaled down by 50% to exclude energy spent on cooling and other operations, consistent with the methodology from DRISA [87].

Circuit-level SPICE Validation. Of all the Sieve components, only the Matchers are in direct contact with the sense amplifiers' BLs. In the presence of the Matcher circuit, the load capacitance on the BL is increased. We use SPICE simulations to confirm that Sieve works reliably. The sense amplifier and matcher circuits are implemented using 45nm PTM transistor models [104]. Because of the relatively small input capacitance of the matcher circuit ($\sim$0.2 pf), in comparison with the BL capacitance ($\sim$22pf), the matcher has a negligible effect on the operation of the sense amplifiers.We find that, after the row activation and when the BL voltage is at a safe level to read, the result of the matcher is ready after less than 1 ns. To validate correct operation of links in Type-2, we use our DRAM circuit model to simulate transfer of data between local row buffers of two adjacent subarrays. In both simulations, the initial charge of the cell is varied across different values to consider the effect of DRAM cell charge variations. Even in the worst case, the matcher and the link between two subarrays cause no bit flips or distortions.

Energy, Area, and Latency Modeling. We estimate the power and latency overhead of each Sieve component using FreePDK45 [105]. Further, we use OpenRAM[106] to model and synthesize the SRAM buffer in Type-1. We use scaling factors from Stillmaker, et al. [107] to scale down results to the 22nm technology node, and use the planar DRAM area model proposed by Park, et al. [108] to estimate area overhead.

Modeling Sieve. We assume a pipelined implementation of Sieve, where the host (CPU) performs pre-processing (k-mer generation, driver invocation, and PCIe transfer) and post-processing (accumulation of response payloads for genome sequence classification), while Sieve is responsible for k-mer matching. Our analysis confirms that the latency of this pipeline is limited by k-mer processing on Sieve. In particular, k-mer matching on Sieve is either comparable to (for Type-3) or slower than (for Types-1/2) both pre- and post-processing steps on the CPU, so the CPU is always able to send enough k-mer requests to Sieve to keep it fully utilized.

We model the pre- and post-processing steps using the baseline CPU described in Table 3.1. We treat the classification step as a separate pipeline by itself, as (1) the algorithm differs for each application, and (2) it is independent of k-mer matching, which is the primary focus of this work. Thus, we forgo modeling the effort required for genome classification and other post k-mer processing. For modeling the k-mer matching itself, we use a trace-driven, in-house simulator with a custom DRAMSim2-based front-end. The simulator also models PCIe communication overhead, using standard PCIe parameters [109]. We use a Micron DDR4 chip (DDR4_4Gb_8B_x16) as the building block for Sieve. DRAM parameters are extracted from the same datasheet and modified to account for the estimated latency and energy overhead of matchers, ETM, column finder, and segment finder.

3.5 Results

3.5.1 Energy, Latency, and Area Estimation

Energy Evaluation. Table 3.3 summarizes the dynamic energy and static power of each Sieve component. Type-3 incurs additional power consumption for each DRAM row activation. However, using formula 10a from Micron's technical documentation [97], we find that Sieve consumes only 6% more energy for each row activation than a regular DRAM, because the area and the load of the extra transistors we introduce is so small compared to the sense amplifier and the bitline drivers. We further break down this energy overhead to understand the effect of the different Sieve components. We find that the Matcher Array (MA) and the ETM dominate the energy consumption, capturing 78.9% and 15.8% of the 6% energy overhead incurred by Sieve, with the energy spent by the Segment Finder and the Column Finder being negligible (less than 5%). Type-1 adds no overhead on top of the regular DRAM row activation because no modification is made to the row buffer, and it is less energy-intensive than Type-2/3.

Latency Evaluation. Table 3.3 shows the latency of each Sieve component. For Type-1, we assume that (1) accessing the SRAM buffer and the Query Register can be overlapped entirely with a column read command (∼15 ns) that retrieves a batch of reference bits, and (2) although the pattern matching and register checking are on the critical path, they add negligible overhead (∼0.5 ns) to the DRAM row cycle (∼50 ns). For Type-2/3, each ETM segment (256 OR gates) meets the timing requirement of completing its operation within one DRAM row cycle. Further, since the segment and column finders are composed of simple shifters, their latency of operation is well within one DRAM cycle (0.625 ns).

Area Evaluation. To estimate area the overhead of Sieve, we use the model proposed by Park et al. [108]. We adopt the DRAM sense amplifier layout described by Song, et al. [110] and a patent from Micron [111]

Table 3.3: Sieve Components Energy and Latency Analysis

Component	Dynamic Energy (pJ)	Static Power (uW)	Latency (ns)
(T1) 64-bit MA	0.867	1.4592	0.353
(T1) QR, SkBR, StBR	1.92	5.28	0.154
(T1) SRAM Buffer	5.12	4.445	0.177
(T2/3) 8192-bit MA	181.683	0.289	0.535
(T2/3) ETM Segment	73.5	56.185	43.653
(T2/3) Segment Finder	2.44	0.294	0.362
(T2/3) Column Finder	20.69	28.16	0.152

for a conventional $4F^2$ DRAM layout. The short side and long side of the sense amplifier are 6F and 90F, respectively. In Type-2/3, for the accommodation of the matcher, ETM, segment, and column finder circuits in the local row buffer, we add 340F in total on the long side of the local sense amplifiers. For Type-2, an extra 60F in long side is added to each sense amplifier for considering the area overhead of the links between the subarrays.

The area overheads for Type-2 with 1, 64, and 128 compute buffers (CB) are 1.03%, 6.3% and 10.75%, for an 8-bank DRAM chip. In Type-3, each local sense amplifier is enhanced with k-mer matching logic, and for enabling subarray parallelism, a row-address latch is added to each subarray [42], resulting in 10.90% area overhead. For Type-1, all components are added to the center strip of our DRAM model. The SRAM buffer of 8 Kbits (128 Rows X 64 Bits) and matching circuit in each bank increase the area by 2.4% and 0.08%, individually.

3.5.2 Kernel Performance Improvement

Comparison Against Row-major In-Situ Accelerators. We simulate an ideal row-major baseline which mimics prior proposals [87, 88, 91] (Row_Major in Figure 3.11), and an improved row-major accelerator based on ComputeDRAM [93]. We measure their speedup over the CPU baseline. We also implement Sieve without ETM (Col-major).

We make the following assumptions for the Row-major, ComputeDRAM-based, and Col-major accelerators. First, their latency for locating and transferring payloads is assumed to be similar to that of Sieve. Second, both architectures are configured to be the same capacity with the same subarray-level parallelism. Third, they share the same indexing scheme. Fourth, we assume that ComputeDRAM has a much shorter Triple-row Activation latency due to the fact that it issues memory commands in rapid succession.

Figure 3.11 shows the results from this experiment. The convention for the workloads on the X-axis is kernel.query.size. The kernel is either Kraken2 or CLARK, the query files are listed in Table 3.2, the sizes are 4GB, 8GB, and NCBI Bacterial reference (6.24GB). We make the following observations.

First, row-major perform similarly to column-major without ETM (slightly worse), but for different reasons. Column-major must activate all the rows that store k-mer data (64 rows if k=32). Row-major and ComputeDRAM stop when it finds a hit, but requires ∼10X more writes to set up the comparison, as each query k-mer must be replicated across the length of the row. Second, ComputeDRAM is able to outperform both the row-major and column-major (without ETM) approaches, owing to its fast triple-row activation. Third, the column-major approach used in Sieve allows it to benefit from our ETM strategy (that provides an additional speedup of 5.2X to 7.2X), in contrast to both row-major and ComputeDRAM designs that lack such an opportunity. We conclude that the chief contribution of column-major layout is therefore 1) in enabling ETM and 2) in amortizing the setup cost across a pattern group of 64 writes. The row-major design performs slightly worse than Type-3 without ETM because, in the event of a k-mer mismatch, both designs on average open roughly the same number of rows (62 8192-bit rows), but the row-major design stops when it finds a hit. We note that, from our evaluation, real sequence datasets are typically characterized by low k-mer hit rates (around 1%), thus favoring Sieve designs.

Leveraging ComputeDRAM to build a column-major k-mer matching accelerator entails solving many challenges. If we populate the query section with the same query, we will need 128×64 write commands per query (630X more than Sieve). Populating the query section with different queries brings more challenges. For example, there could be more than one match, impacting our ability to design an efficient indexing scheme. We note that addressing these challenges while maintaining the performance, efficiency, and cost benefits of these approaches is the subject of future work.

Improvement Over CPU. Figure 3.12 shows the average speedup and energy savings. All results are normalized to CPU measurement. In this experiment, we constrain the memory capacity of all designs to 32 GB. For Type-2, we consider all possible numbers of compute buffers per bank and select the midpoint of 16 (T2.16CB). We present the performance of other Type-2 configurations in Section 3.5.2. For Type-3, we choose the best performer, which supports 8 concurrently working subarrays (T3.8SA). While clearly more energy-efficient, Type-1 offers limited speedup (1.01X to 3.8X) for 8 out of 9 benchmarks, showing that for many workloads, there is significant additional performance potential that can be tapped via an in-situ approach. However, we also point out that Type-1 is likely to outperform CPU/GPU as its memory capacity grows (more banks thus more parallelism and bandwidth), while the similar memory-capacity-proportional performance scaling is hard to achieve in a non-PIM traditional architecture [17], due to the memory wall. Type-3 designs offer a speedup and an energy savings of as much as 404.48X and 55.89X respectively, over the CPU baseline. Note that this is in comparison to a Type-2 design that offers a speedup of 55.49X and an energy reduction of 28.11X over the CPU baseline, clearly showcasing the substantial benefits that can be realized by exploiting finer-grained parallelism at the subarray-level. We also find that Sieve is sensitive to

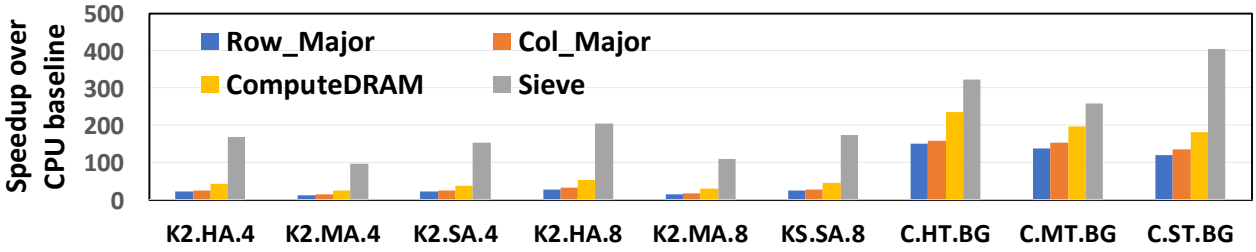


Figure 3.11: **Row-major in-situ vs. Sieve Comparison.**

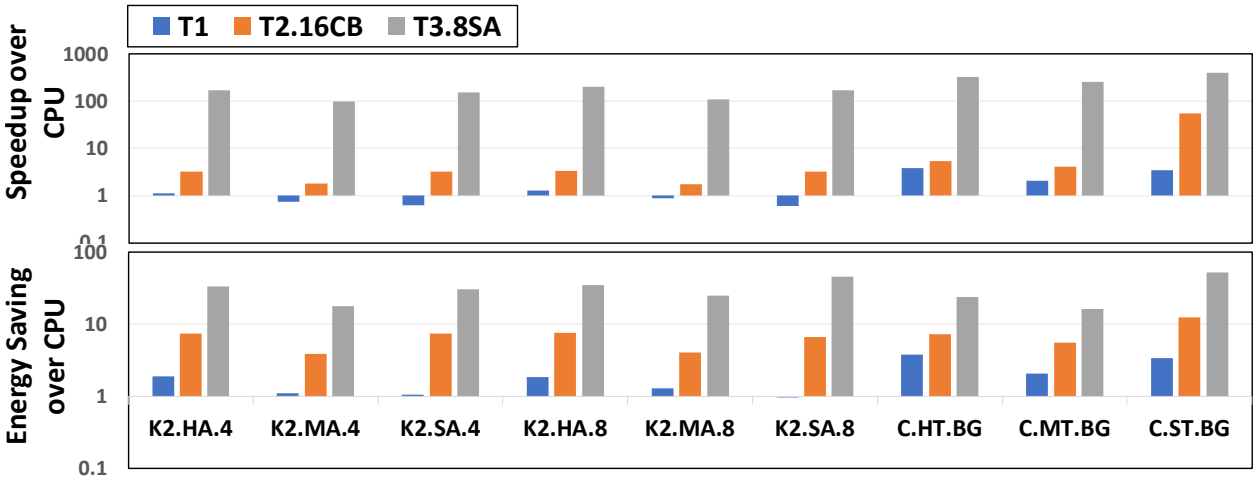


Figure 3.12: **Comparison with CPU baseline.**

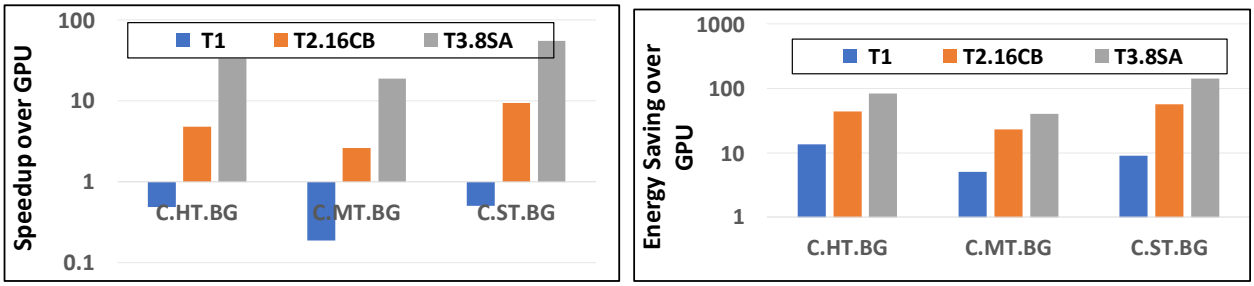


Figure 3.13: **Comparison with GPU baseline.**

the characteristics of the application. For example, the C.MT.BG benchmark performs worse than C.ST.BG benchmark as the number of k-mer matches for C.MT.BG is 3.28X higher than C.ST.BG benchmark, resulting in more row activations, increasing the overall query turnaround time and energy. Furthermore, our early termination mechanism interrupts row activations as soon as we detect a mismatch, minimizing the overall turnaround time and energy consumption for workloads with fewer k-mer matches.

Improvement Over GPU. Figure 3.13 shows the speedup and energy savings of various Sieve designs (32 GB) over the GPU baselines. Type-1 is 3X to 5X slower than the GPU but more energy efficient, and Type-2 is only modestly faster (2.59x to 9.43x). However, as the memory capacity of Sieve and dataset size increase, Type-1/2 are likely to outperform the GPU unless GPU memory capacity scales as fast, because all reference datasets can fit onto Sieve, avoiding the repetitive data transfer from host memory to GPU board.

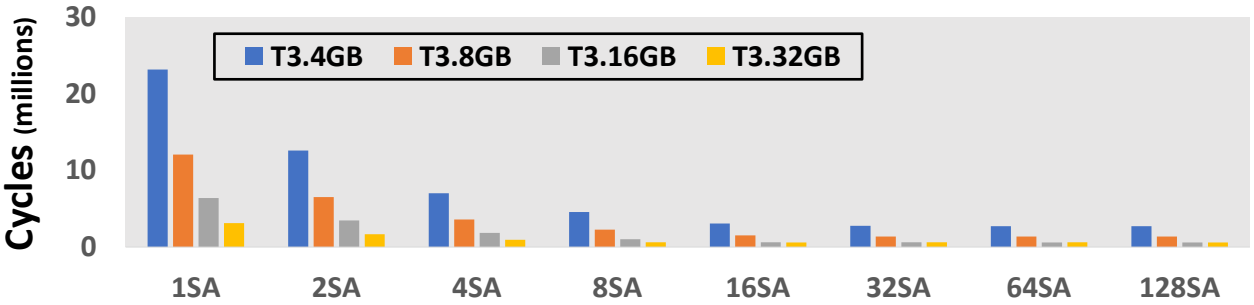


Figure 3.14: **Average cycles spent to process CPU benchmarks.**

Type-3 dramatically outperforms the GPU, because it leverages subarray-level parallelism. Type-3 offers speedups of 33.13X–55.0X and energy savings of 83.77X–141.15x.

Effect of Increased DRAM Bandwidth. Simply increasing bandwidth to DRAM in the CPU and GPU baselines is not sufficient to address the performance bottleneck in k-mer matching, because we find that it is not bottlenecked by bandwidth. While it is memory-intensive (high percentage of loads in the ROB), memory bandwidth is underutilized because each MSHR is unable to serve multiple loads and the available MSHRs are quickly depleted, stalling subsequent loads in the ROB and preventing the bandwidth from being fully saturated. Even if we overprovision those Broadwell cores with enough MSHRs to sustain all outstanding memory accesses, and all loads are served concurrently with a memory latency of 40 ns to reach the same level of throughput as Type-3, the workstation has to be equipped with over 215 cores, not only resulting in a substantial increase in power consumption, but a considerable wastage in DRAM bandwidth as only a small portion of the retrieved cache line is useful. cuCLARK is highly optimized, so we suspect that GPUs are constrained by similar bottlenecks as CPUs, although we have not yet pinpointed the exact set of microarchitectural structures.

3.5.3 Sensitivity Analysis

Number of Subarrays per Bank. We analyze the impact of subarray-level parallelism on performance and energy by comparing various Type-3 design configurations (see Figure 3.14) at different memory capacities and number of subarrays per bank. The results are averaged across all benchmarks. Supporting all subarrays performing k-mer matching simultaneously without increasing the area overhead significantly is not yet feasible, due to power delivery constraints. However, for this experiment, we assume this is not an issue. In any case, although Sieve's k-mer matching throughput increases with more concurrent subarrays, the speedup plateaus after 8 subarrays—probably because most bank-access conflicts can be resolved by a small number of subarrays [42].

Number of Compute Buffers. We explore the performance-area tradeoff of Type-2 designs, by varying the number of compute buffers (shown in Figure 3.15). For reference, we include Type-1 (the left-most bar T1) and Type-3 (the right-most bar T3.1SA) designs without subarray-level parallelism. The middle eight bars represent Type-2 with 1-128 compute buffers per bank. We make the following observations. First, Type-2 with one compute buffer is faster than Type-1 (1.39X to 1.94X) but not by a large margin. For each row activation, in the worst case, Type-1 has to burst read 128 batches to the matchers, which is similar to T2.1CB where the opened row needs to "hop" across 128 subarrays to reach the compute buffer. Since the hop delay (∼4ns) is faster than a burst latency (tCCD: 5∼7ns), and both design are equipped with some forms of ETM, T2.1CB is likely to spend less time on data movement than Type-1 in the average case. However, the chain activation of sense amplifiers in Type-2, which relays the row to the compute buffer, consumes significant energy, making Type-2 with sparse compute buffers less energy efficient. Second, generally speaking, increasing the number of compute buffers per bank also increases the speed and energy efficiency of Type-2. As we have explained previously, adding more compute buffers reduces the activation of sense amplifiers, which in turn reduces the delay and energy consumption. Third, the area overhead scales with the number of compute buffers per bank. Finally, the speedup and energy reduction of T2.128CB slightly trails behind those of T3.1SA, because T2.128CB still requires one hop per row activation. However, Type-3 also has a higher area overhead than T2.128CB for enabling subarray-level parallelism.

ETM. To simulate the adversarial case where every query k-mer has a match, we turn ETM off in Type-2/3, and measure the speedup and energy reduction over CPU/GPU baselines (averaged across all benchmarks). Type-2/3 without ETM are still 1.34x–155.37x faster and 4.15x–36.17x more energy efficient than CPU, and 1.3X–9.54X faster and 6.60X–18.43X more energy efficient than GPU.

PCIe Overhead. We use PCIe 4.0 x16 in our simulation. Overall, PCIe adds 4.6% to 6.7% communication overhead to the ideal case where k-mer matching requests are dispatched to the destination bank/subarray as soon as they arrive, and returned to the host when they complete.

3.6 Related Works

In this section we discuss previous work that shares similar interests concerning Sieve. The concept of PIM dates back to the 70s [112]. Since then, there have been many proposals integrating heavy logic into 2D planar DRAM dies [113, 114, 115, 116, 117]. These early efforts largely remain at their inception stage due to the challenges of fabricating logic using the DRAM process. Recently, the 3D-stacked technology, which takes a more practical approach by placing a separate logic die underneath the DRAM dies, revitalizes the interests in PIM research. To fully exploit the benefit of 3D-stacked architectures, many domain specific

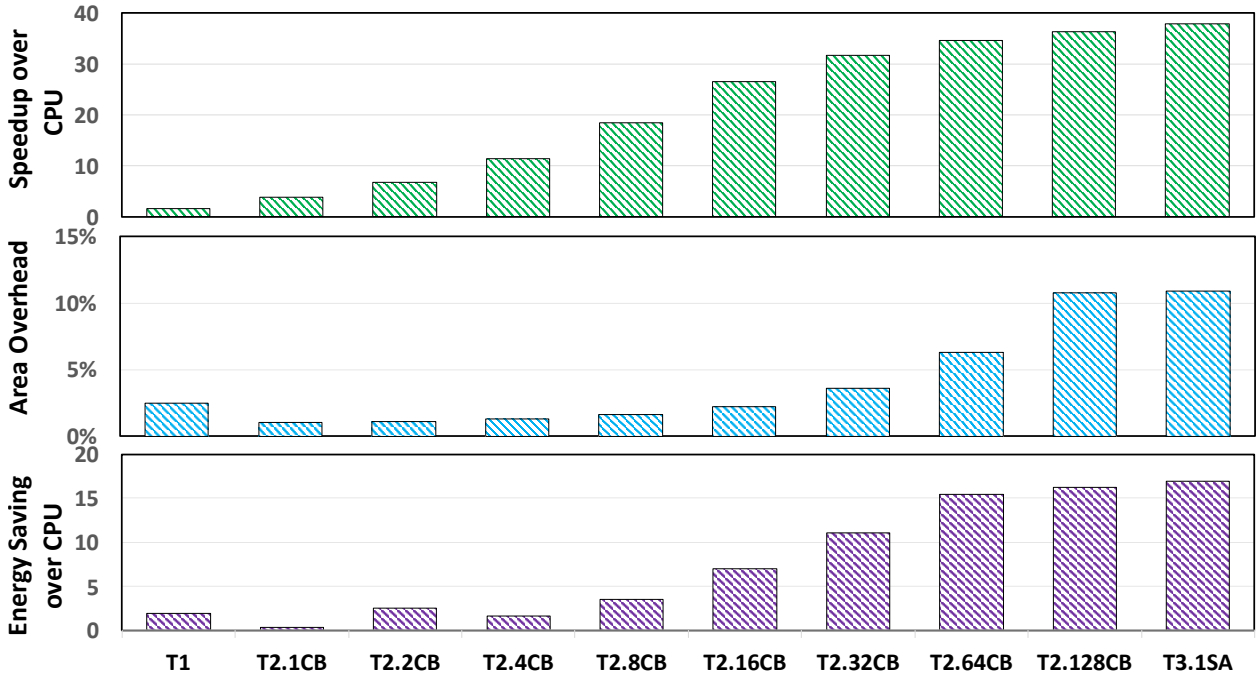


Figure 3.15: **The effect of varying the number of compute buffers. T = Type, #CB = number of compute buffers.**

accelerators for graph processing [17, 48], pointer chasing [118], and data analytics [50] have been proposed. We plan to evaluate Sieve in 3D-stacked context as future work.

Non-DRAM-based In-situ Accelerators. NVM- and SRAM-based in-situ accelerators such as Pinatubo [119] and Compute Caches [120] have been proposed, but we choose DRAM for its maturity and availability, which can lead to quicker development and deployment cycles. Furthermore, SRAM generally has a lower capacity than that of DRAM, a smaller number of subarrays, and shorter row buffers. We plan to evaluate NVM-based Sieve in future work.

PIM-based Genomics Accelerators. Recently, PIM has been explored for several algorithm-specific PIM architectures for genomics. For example, GenCache [84] modifies commodity SRAM cache with algorithm-specific operators, achieving energy reduction and speedup for DNA sequence aligners. Medal [85] leverages commodity Load-Reduced Dual-Inline Memory Module (LRDIMM) and augments its data buffers with custom logic to exploit additional bandwidth and parallelism for DNA seeding. Radar [86] provides a high scalability solution for BLAST by mapping seeding and seed-extension onto dense 3D non-volatile memory. However, these efforts are not ideal for k-mer matching. GenCache has hardwired logic in SRAM to compute Shifted Hamming Distance and Myer's Levenshtein Distance, which are not used for k-mer matching. Medal is highly optimized for FM-index based DNA seeding, which relies on different data structures (suffix arrays, accumulative count arrays, occurrence arrays) than those in k-mer matching (associative data structures such

as dictionaries). Radar binds seed-extension, a stage irrelevant to k-mer matching, with seeding to maximize speedup.

PIM-based Genomics Accelerators. PIM has been explored for several algorithm-specific architectures for genomics. For example, GenCache [84] is an SRAM-based accelerator for DNA sequence alignment. Medal [85] augments the data buffers of commodity DIMM to exploit additional bandwidth and parallelism for DNA seeding. Radar [86] provides a high-scalability solution for BLAST by mapping seeding and seed-extension onto dense 3D NVM. These efforts rely on domain-specific knowledge to achieve maximal speedup for specific algorithms that are not applicable to k-mer matching, but are complementary to Sieve.

3.7 conclusions

In this work, we identify k-mer matching as a bottleneck stage in many genomics pipelines, due to its memory-intensive nature. We propose Sieve, a set of DRAM-based in-memory architectures to accelerate k-mer matching, by storing reference k-mer patterns along the bitlines and enhancing row buffers with a minimal set of Boolean logic for k-mer matching. We optimize Sieve with an Early Termination Mechanism. Type-1 offers limited benefit over CPUs and GPUs. Type-2 offers extensive speedups over CPUs (3.74x to 76.62x) but only modest benefit over GPUs (1.33x to 12.97x). Type-3 offers compelling benefits over both, with speedups and energy savings over the CPU of as much as 389.49X and 93.97X respectively; and 6.05x and 68.74x over the GPU.

Chapter 4

DRAM-CAM: General-Purpose Bit-Serial Exact Pattern Matching

4.1 Introduction

Exact pattern matching is a widely used computation kernel. A common software implementation is a lookup or hash table, but large data sets do not fit into the last-level cache (LLC) and exhibit poor locality. Furthermore, the computation per pattern lookup is also too small to mask the high memory-access latency, resulting in frequent processor stalls [53], making the task memory-bound. An alternative is a coarse-grained index that fits in the LLC, in which a key is mapped to a bucket of potential matches, with linear or binary search within a bucket. However, our prior results [53] show poor temporal locality in which buckets are accessed.

To address these limitations, data-centric architectures leveraging content addressable memory (CAM) have been proposed [121, 122]. This paper describes how to implement CAM functionalities inside *DRAM*, which offers several advantages over non-volatile memory (NVM) and SRAM alternatives. Even a highly compact 3T3R PCM NV-CAM cell is over 3X larger than a DRAM cell, and SRAM is much less dense and more power-hungry.

The proposed architecture, DRAM-CAM, is built on Sieve [53], a recently-proposed processing-in-memory (PIM) key-value accelerator designed originally for massively-parallel k-mer matching (searching for short DNA sequence patterns of size k), but more generally suitable for a variety of key-value applications. Sieve provides an average of 326X/32X speedup and 74X/48X energy savings over multi-core-CPU/GPU baselines for k-mer matching, using a column-wise data layout for patterns, allowing element-parallel, bit-serial matching

Table 4.1: Mapping exact matching kernels onto DRAM-CAM

Benchmark	Index	ETM	PCL	DTU	CLP	Input	Payloads	DRAM-CAM patterns	DRAM-CAM computing
String Match	Yes	Yes	No	No	Yes	Key file	None	Encrypted file	Search keys in the encrypted file
Histogram	No	Yes	Yes	Yes	No	8-bit pixels	None	Image binary pixel values	Aggregate hits for each pixel pattern
Word Count	No	Yes	Yes	Yes	No	Unique words	None	Words from text file	Aggregate hits for each input word
Bitcount	Yes	No	No	No	Yes	32-bit binaries	Num of set bits	32-bit binaries	Retrieve number of set bits
Apriori	No	No	Yes	No	No	Itemsets bit vectors	None	1-hot encoded transactions	Check if transactions contain an itemset

(each bit position is checked across a large number of bitlines, i.e. data items). Sieve and SIMDRAM [123] showed that this offers better matching throughput than a traditional, row-wise data layout. This allows Sieve to integrate low-overhead bit-wise logic inside row buffers, coupled with subarray-level parallelism, to simultaneously compare thousands of patterns in each row cycle without incurring expensive data movement. Although a similar in-situ approach has been explored in prior proposals such as Ambit [88] and SIMDRAM, their multi-row activation-based approach, which relies on charge-sharing, is more energy-intensive and slower than the sequential single row activation and digital comparisons employed in Sieve [53], due to the overhead of row-copy operations involved to set up operand rows in the "Bitwise" group for pattern matching[123, 88]. Furthermore, column-wise data layout and single-row activation allow Sieve to exploit an Early Termination Mechanism (ETM) that prevents unnecessary DRAM row activation if all columns have encountered a mismatch. Therefore, even if the slow multi-row activation mechanism is replaced with rapid timing-constraint-violating DRAM commands that leave multiple rows open to perform fast row-wide logic operations, as described in ComputeDRAM [93], Sieve still performs better by a large margin due to the benefit of ETM, which is not possible in a row-wise data mapping. Furthermore, combining ComputeDRAM with a vertical data layout is unlikely to outperform Sieve, because of the much larger overhead of setting up queries for the target subarrays [53].

In this paper, we add several features that enable a wider range of pattern-matching applications, including population-count logic to count matches (in Sieve, a given k-mer will have at most one match), hardware support for faster transposition of data into the column-wise format, and optimizations for greater parallelism. The evaluation shows that DRAM-CAM provides up to three orders of magnitude of speedup and energy reduction over the CPU baselines, and on average outperforms the closest PIM competitor by 3.7X.

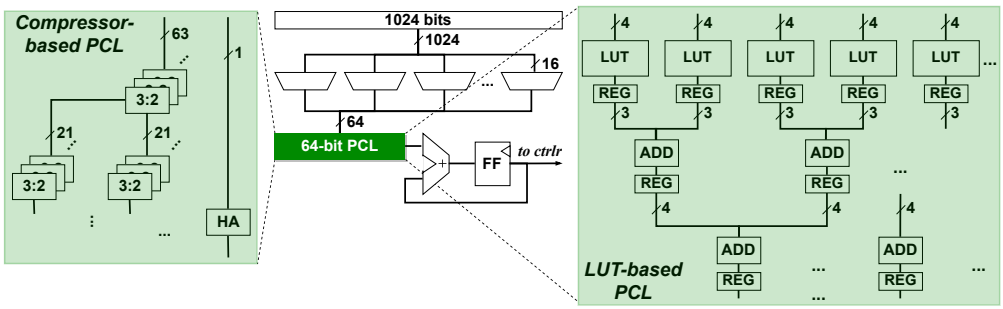


Figure 4.1: Population count logic.

4.2 Architecture

DRAM-CAM retains the core architectural designs of Sieve and serves as a PCI-attached accelerator with an offload model. We introduce several hardware components and runtime optimizations pertaining to DRAM-CAM.

Population Count Logic (PCL). A population count logic (PCL) unit returns the total number of matches for each query. The PCL accumulates the number of ones from the row of latched bits at the subarray level, then aggregated at the controller level for the total number of hits. In many use cases, aggregating hits for each query accounts for nearly the entirety of the workload. Integrating PCL at the subarray level is difficult since it needs to process a large bit vector in a timely fashion with minimal hardware overhead.

Our PCL design, shown in Fig. 4.1, works on 1024 bits by processing chunks of 64 bits. To count the 1s in a group of 64 bits, we explore two options: lookup table (LUT) and Wallace-tree-architecture compressor tree circuit [124]. The first level of the LUT-based PCL requires 16 four-input LUTs that take four bits from the latches and output the number of ones in binary. The remaining levels of this PCL are like an adder tree, aggregating ones from all LUTs. One optimization is to insert registers between levels to form a pipelined PCL, which reduces latency but increases area and power overhead (see Table 4.2). The compressor-tree PCL is based on [124], which uses 57 3:2 compressors and 8 half-adders in ten cascading stages. The 3:2 compressor has the same truth table as a full adder. Each compressor processes 3 bits, outputting the number of ones in its *sum* and *carry* bits as $sum + 2 \times carry$.

Data Transposition Unit (DTU). If the reference patterns are reused across different executions, transposing the data in software is a one-time cost amortized over a long period of use. However, some workloads require input data to be transposed on the fly and written to the DRAM-CAM prior to matching, which places the data transposition operation on the critical path. We integrate a simplified data transposition unit (DTU) from SIMDRAM [123] into DRAM-CAM. The DRAM-CAM DTU requires only one 4KB SRAM transposition buffer. DRAM-CAM's DTU works at a rate of transposing one cache line worth of data (512 bits) in one cycle. We estimate that such hardware DTU is 381.3X faster than a software one (estimated

using a modified DRAMSim2), and adds an insignificant (<0.1%) amount of execution time. Once the CPU with the help of a dedicated runtime environment (future work) instructs the DRAM-CAM device to load the reference pattern sets (e.g., image data for Histogram) from disk using DMA, data first arrive at this SRAM buffer, then transposed row by row by simple custom logic and written into DRAM-CAM using DRAM commands.

Chip-level Parallelism (CLP). Sieve chips in a rank respond to queries in a lockstep manner due to the shared chip select signal (CS), a design carried over from a traditional DDR architecture. Chip-level parallelism (CLP) can be achieved to a certain degree by providing each chip with a dedicated chip select wire. Note this solution does not make each chip truly autonomous, because the data line (DL) still has to be shared inside a rank due to limited high-frequency data pin count, which is prohibitively expensive to scale. DRAM-CAM chips receive their input queries once the shared DL is available, thus only pattern matching is parallelizable, while query input is serialized. The downside of CLP is that the number of entries in the index table will be increased since chips need to be indexed. However, the granularity of the indexing scheme can be adjusted if needed to keep the index within L2 capacity.

Runtime Optimizations. To leverage the parallelism of DRAM, we want to leverage as many subarrays as possible, which reduces congestion and maximizes parallelism. DRAM-CAM starts offloading patterns by choosing a random subarray for pattern storage, and after it is filled with *subarray_width* patterns, randomly chooses the next subarray from a different channel/rank/bank for pattern placement. Further optimization is to replicate small reference pattern sets multiple times by storing them in unused subarrays, which allows applications to use them for greater parallelism. To support this optimization, the main changes occur in the index table, where one additional busy bit for each entry is needed to indicate if the subarray is currently being used or not. When a new query arrives, the index table chooses a subarray whose busy bit is 0 that stores the same reference patterns. If all candidate subarrays are busy, we choose a random one to wait upon. Pattern distribution (PD) offers 22% to 7.4X speedup while pattern replication (PR) offers 4X to 29.4X speedup over an unoptimized pattern storage scheme (Fig. 4.2). PR generally offers better performance than PD, because it allows DRAM-CAM to utilize subarray-level parallelism on top of bank-level parallelism.

Application Mapping. While some kernels map to DRAM-CAM naturally, such as *String Match (SM)* and *Bitcount (BC)*, others are not so straightforward and require algorithmic changes. *Histogram (HG)* and *Word Count (WC)* differ most from their CPU counterparts, where the input images or text files are transposed into DRAM-CAM prior to the matching process. Then a standardized input set such as all 8-bit pixel patterns or unique English words are passed as input to aggregate hits. For *Aprior (AP)*, the entire transaction database is transcribed using one-hot encoding, with each column representing a transaction and each row representing an item. To check if a candidate itemset is a subset of a transaction, the i^{th} row

Table 4.2: Population Count Logic Characteristics

	LUT no Pipeline	LUT Pipeline	Compressor Tree
Area (nm^2)	201	554	148
Delay (ns)	0.76	0.34	0.84
Power (uW)	0.03	0.06	0.02

Table 4.3: Workstation Configuration

CPU Model	Intel(R) Xeon(R) E5-2658 v4
L1 /L2 /L3 $	32 KB / 256 KB / 35 MB
Main Memory	DDR4-2400MHz (32 GB/2 Chan)

corresponding to the i^{th} 1 of the bit vector is opened. Table 4.1 shows more details of mapping each kernel onto DRAM-CAM. One interesting discovery is that the best way to utilize ETM in natural language (e.g., *Word Count*) is to match the patterns backward, due to the significant prefix overlapping.

4.3 Evaluation

The experimental setup and evaluation methodology are identical to those of [53]. The baseline DRAM energy is estimated by feeding memory traces to DRAMSim2, configured to match our workstation. The CPU energy is measured using the Intel PMC-power tool, then scaled down by 30% to exclude the interference from other system components, consistent with the methodology from DRISA [87] For application performance modeling, we use a trace-driven, in-house simulator that has a custom DRAMSim2 as the front end. We use the Micron DDR4 4Gb 8B x16 chip as the building block. We assume a pipelined implementation of DRAM-CAM, where the host (CPU) performs pre-processing and post-processing, while DRAM-CAM is responsible for pattern matching. We use Verilog to implement different versions of the population count circuit. Then, we estimate power/area/latency using Synopsys in 90nm. Finally, we use scaling factors from [107] to scale down results to 22nm. See the original Sieve paper [53] for more methodology details. Table 4.3 reports the CPU hardware setup. We measure the portion that can be offloaded to DRAM-CAM, which is 98.97% for *String Match*, 75.88% for *Histogram*, 92.99% for *Word Count*, 100% for *Bitcount*, and 63.00% for *Apriori*. Table 4.2 summarizes performance characteristics for PCL. The compressor-based PCL has lower area and power, while the pipelined LUT-based PCL is the fastest. We propose to fit PCL in the center strip of each DRAM chip, and each PCL is time-shared among subarrays of a bank. This setup increases the latency slightly. Decoupling CS signals to enable chip-level parallelism requires negligible hardware changes. For the data transposition unit, the primary component is a 4KB SRAM buffer. We estimate its area to be 0.015 mm^2, and it consumes 2.22 uW.

Exact Pattern Matching Workloads. We select the same applications from [121] (Table 4.1), minus *Vortex*, which is deprecated and not open source, and *ReverseIndex*, which does not map well to DRAM-CAM.

String Match processes a key file of strings and a file of hashed (encrypted) strings to find which keys occur in the encrypted file. *Histogram* counts frequencies of pixel values in the RGB channels of a bitmap. *Word Count* generates the frequency for each word in a text file. *Apriori* performs associative rule mining, building a candidate itemset, and counts itemset frequencies in a transaction database. Our results suggest DRAM-CAM's bit-serial nature favors workloads with shorter patterns (several hundred bits or less). The Reverse Index is a "bad fit" because each pattern (URL links) is too long for DRAM-CAM to handle. DRAM-CAM also favors kernels that can issue large batches of pattern search requests to fully leverage parallelism in the DRAM hierarchy.

Performance improvement over CPU. Figure 4.2 reports the speedup and energy saving over a CPU baseline of various DRAM-CAM configurations, including the performance of our unoptimized (UNOPT) setup, which is closest to the original Sieve architecture while enabling these other applications, and the benefit of three optimizations: pattern distribution (PD), pattern replication (PR), and chip-level parallelism (CLP). For applications that need PCL, we model LUT with the pipeline. The optimizations are highly effective when the reference pattern set is small, because it can be distributed and replicated many times to leverage the massive internal parallelism of DRAM. Additionally, chip-level parallelism offers approximately 2.9X speedup when applicable, but it does not help when a query needs to visit all subarrays to aggregate hits. *String Match (SM)* shares the most similarities with k-mer matching and benefits the most from such an accelerator. *Word Count (WC)* only experiences modest speedup. In fact, UNOPT is 1.5X slower than CPU. There are two reasons: (1) long string patterns and high match rates cause frequent and long sequences of DRAM row openings, and (2) a large input set (reference patterns) that limits optimization potential. This is in contrast to *Apriori (AP)*, which also stores large reference sets and long patterns. but only opens a few rows (<10). DRAM-CAM outperforms *Bitcount (BC)* on the CPU, because it stores a much larger lookup table (32-bit vs. 8-bit patterns).

The baseline DRAM-CAM (UNOPT) tends to show the best energy efficiency because the dynamic power consumption of DRAM-CAM depends on the number of banks that are used for pattern matching, and the UNOPT setup uses only a small percentage (0.7% $\sim$ 50%) of banks, resulting in up to 126.4X lower power than the CPU baseline. There is a tradeoff between greater parallelism and higher energy. PD shows worse energy saving than UNOPT, even though it offers better speedup, because UNOPT uses all subarrays of a smaller set of banks, but leverages subarray-level parallelism (SALP) to its full potential, thus making up the performance loss due to increased bank conflicts. On the other hand, PD usually utilizes fewer subarrays from a larger set of banks, resulting in sublinear speedup w.r.t. bank count. PR usually shows better energy saving than PD, except for the SM benchmark, by exploiting more SALP. SM has a small input set, and PD utilizes only two banks (low power). PR offers 16X speedup, but needs 128 banks, However, HG, WC,

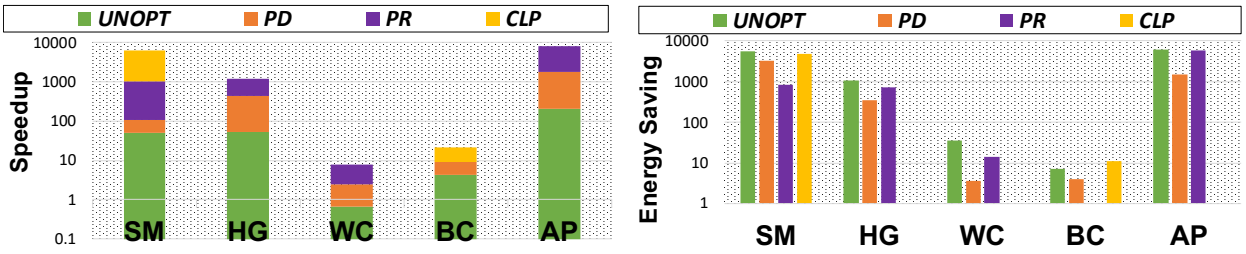


Figure 4.2: Comparison with CPU baseline.

and AP have larger data sets, and PD requires the same amount of banks as PR, meaning they have similar dynamic power consumption. Since PR significantly reduces the execution time of those benchmarks, it offers better energy efficiency for those benchmarks. Finally, CLP increases power consumption minimally, but the performance improvement is significant, so the energy savings approach or surpass UNOPT.

Comparison to alternative PIM designs. We compare DRAM-CAM with several prior DRAM-based in-situ proposals. Fig. 4.3 reports the results, and the performance numbers are normalized to CPU baselines. We assume all indispensable architectural features such as population count logic are enabled for all architectures, even though they are missing from some prior works, and all appropriate hardware and software optimizations proposed in this work are equally applied to prior works. *Ambit* adopts the traditional horizontal data layout (row-major) and triple-row-activation (TRA) based logical operation (XNOR) for pattern matching. *ComputeDRAM-H* reduces TRA latency by half but retains the horizontal data layout. *ComputeDRAM-V/SIMDRAM* switches to vertical data layout (column-major) with TRA. In addition to charge-sharing based in-situ accelerators, we also simulate variations of DRISA, which combines analog bit-line functionality with digital logic in the row-buffer. *DRISA-H* uses horizontal data layout while *DRISA-V* uses vertical, and *DRISA-V-Batch-Input* is *DRISA-V* but utilizes Sieve-style batched queries.

TRA-based pattern matching is inherently slow, even with the modified version proposed in ComputeDRAM. Each row-wide comparison takes multiple DRAM cycles, whereas in-row-buffer logic takes only one. Moreover, exact matching needs to XNOR operand rows, which requires two TRA operations. Second, while for general-purpose computing, vertical data layout has shown better performance, for exact matching, horizontal data layout is better because each query only needs to populate one row, whereas vertical data layout has to populate a two-dimensional block of bits ($subarray\_width \times query\_bit\_length$) for each query to support the bit-serial matching. Third, PIM generally favors short patterns over longer patterns, and this is especially true for column-major layouts. Fourth, DRAM-CAM outperforms DRISA because it has a more efficient way of setting up queries, plus early termination (ETM). Note also that GRIM-filter [125], an HMC PIM for short-sequence DNA alignment, may also support exact pattern matching, an interesting direction for future work.

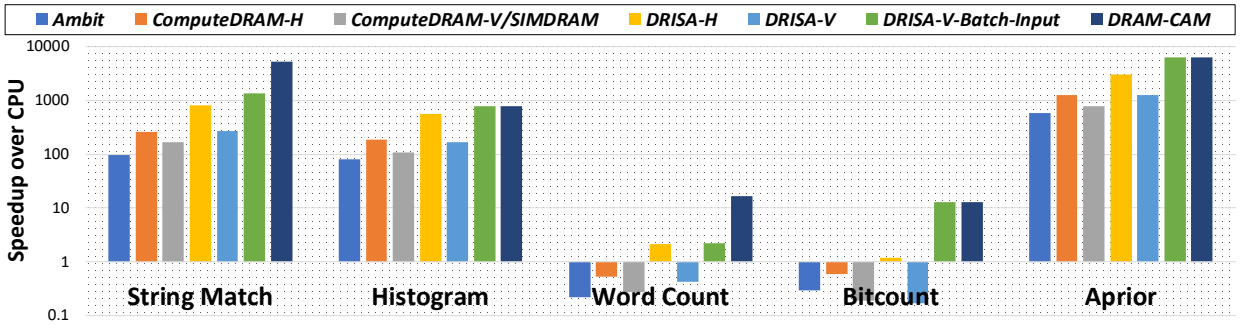


Figure 4.3: Comparison to other in-DRAM accelerators.

4.4 Conclusion

This work develops general CAM functionality inside DRAM, making it capable of accelerating a wide range of exact pattern-matching workloads while achieving significant energy reduction over the CPU, with up to 6217X speedup and 5888X energy savings.

Chapter 5

Membrane: A PIM-based Architecture to Accelerate Database OLAP Queries

5.1 Introduction

Online Analytic Processing (OLAP) systems are critical technologies used by enterprises to unlock the potential of their vast enterprise databases. These systems employ analytic SQL queries to transform database data into visual graphs on live dashboards, generate summary reports depicting the progression of key performance indicators (KPIs) over time, and trigger alerts when KPIs deviate from the norm. In modern enterprise settings, these analytic SQL queries often serve to convert raw data in enterprise databases, often referred to as warehouses, for downstream machine learning (ML) pipelines.

Enterprise databases have consistently grown in size over the past five decades. Despite this growth, the prevailing expectation remains that the underlying OLAP analytic SQL queries will continue to execute quickly and efficiently. Historically, much of this demand has been met by the progressive doubling of performance (both in computation and storage) of the underlying hardware, all while maintaining a near-constant cost from one hardware generation to the next. This phenomenon, a combination of Moore's Law and Dennard's scaling, has fueled this progress. However, it is now evident that this trajectory is no longer sustainable. Indeed, Google recently showed results from profiling its hyperscalar fleet and found that BigQuery, an analytics platform, consumed about 10% total cycles within the fleet, and proposed analytics as a candidate for acceleration. [11]

There is a rich history of enhancing database query speed through hardware innovations. Noteworthy instances of this approach include Oracle's 2009 acquisition of Sun, followed by endeavors to develop database-

specific hardware. Another example involves IBM's acquisition of Netezza, initially intended to capitalize on FPGA-based acceleration, a pursuit that has since been abandoned. In parallel, efforts have also been directed towards enhancing database software to extract greater performance from underlying hardware. This has involved rethinking fundamental mechanisms employed in analytic database engines. Notably, there was a transition from row-store to column-stores, analogous to shifting the underlying data representation from a row-major order to a column-major order. This shift resulted in significant performance improvements, given that analytic queries typically access data in a primarily column-oriented fashion, such as when scanning for records whose fields match certain criteria.

A more recent approach involves additional vertical data shredding at the bit-level using techniques like BitWeaving [126]. This involves transforming the underlying computation into bit-level arithmetic, which can be efficiently evaluated at the circuit level using a concept referred to as intra-cycle parallelism. For example, when evaluating a predicate (such as "columnValue ≥ 5"), data is fetched by the bit position. So, if the memory system is used to fetch 8 bytes of data, what gets fetched is 8 x 8 = 64 bits of the most significant bit (MSB) for 64 consecutive column values. These MSB bits are then compared with the MSB of the value (5), and predicate evaluation proceeds by bit positions but bit-parallel operations like XOR and AND. In many cases, the least significant bits (LSBs) for the column values need not be fetched, reducing the memory fetches and the number of cycles used to evaluate the predicate on the batch of columns (64 in the example above).

Furthermore, the importance of *in-memory* database organizations is growing rapidly for OLAP systems, including in data science and business analytics settings where complex analytic queries are often performed with a human-in-the-loop (a key driver behind the rise of DuckDB) [127]. A PIM-based approach is especially appealing for these workloads because they are often bound by the memory system's performance in conventional von Neumann-style processing systems (which dominates the server landscape on which database systems are deployed). As noted in [128], OLAP applications hit the memory wall [129], and this problem is likely to grow over time as memory densities are likely to grow faster than memory bus speeds (both latency and throughput impact OLAP workload performance) [130]. Furthermore, even when the database does not fit in memory, smart methods of caching data from disk are used by the database management system (DBMS) to keep hot data in memory. Thus the CPU-DRAM level is critical for overall query performance [131].

Our paper explores processing-in-memory (PIM) for analytic SQL queries. Notably absent from the existing efforts in this domain is a comprehensive consideration of both hardware options and software implications. For instance, Ambit [88]) and SIMDRAM [123]) focus on a data layout called BitWeaving-V (where data is stored by the bit positions), while others have considered only traditional column-major layouts,

e.g. [132]. However, no prior work has compared the benefits of these different layouts in a comprehensive way. Moreover, prior research (except for Castle in the SRAM domain [133]) has not thoroughly examined end-to-end query performance across a full benchmark.

In this paper, we show that end-to-end query processing does indeed benefit from PIM and present the following contributions:

1. We concentrate on DRAM-based PIM and explore the hardware design possibilities suitable for data laid out in either the BitWeaving-V or the traditional columnar formats. We introduce two distinct hardware designs: Membrane-V and Membrane-H, based on vertical or horizontal data layouts. These designs highlight the necessity for different architectural elements based on the software approach.

2. We devise query processing mechanisms to comprehensively evaluate a popular SSB database benchmark. Our findings indicate that while our PIM approach accelerates most segments of analytic queries, certain parts, notably aggregation and sorting, are more effectively executed on the CPU. Consequently, we recognize the continued significance of CPUs in analytic query processing.

3. We note that our benchmarking employs a prevalent software-based data acceleration technique called WideTable [134], to convert intricate queries into simple, PIM-friendly scans by joining the data upfront and storing the data in this "denormalized" form. With suitable encoding, the space overhead is modest, approximately 18–22% in our experiments with SSB. In fact, the benefits of eliminating joins and the associated query planning overheads has led to adoption in several commercial products [135, 136, 137].

4. Our investigation reveals that our initial Membrane-V and Membrane-H designs significantly enhance the performance of the most data-intensive component of analytic queries—the table-scan (i.e., filtering) operation. However, a new bottleneck emerges: gathering the selected records for the subsequent query processing phase. To address this need, we show the benefits of a *rank-level unit (RLU)* on the DRAM side of the memory bus for gathering the results of a table scan (i.e., for *early materialization*).

5. Collectively, we analyze the design space encompassing V- and H-based hardware as well as software methods for analytic query processing. Through experimentation on a large 60 GB in-memory SSB database (Scale Factor-100), we demonstrate that our methods yield orders-of-magnitude improvements over existing approaches. As a result, we propose a potential novel avenue for accelerating analytic queries in OLAP systems.

5.2 Background

OLAP Basics There are two main categories of database workloads: *online transaction processing* (OLTP) and *online analytical processing* (OLAP) [73]. These two workloads have starkly different characteristics and

are regularly serviced by two separate database systems. Typically, new data (*e.g.*, orders in an e-commerce application), are first recorded in an OLTP system that uses transactions to safely record the data. Periodically, data from the OLTP system is transferred to a read-mostly, append-only OLAP system commonly known as a *data warehouse* [74]. The OLAP system—the focus of this paper—is used to analyze this historical data.

An OLAP database usually has few fact tables (often just one) and many dimension tables. Fact tables hold transaction records, while dimension tables provide detailed information for specific columns in the fact table records. For example, an e-commerce application may have an orders fact table with one record per purchased item, including the customer ID as a foreign key. A separate customer dimension table would have one record per customer, including detailed customer information and a unique customer ID as the primary key. Other dimension tables may record more purchase details, such as product descriptions. Fact tables tend to be large, while dimensions tables are smaller. OLAP queries often involve selecting and joining dimension table records with fact tables and then aggregating values to produce informative results, such as a list of top products in the last month.

Given the read-mostly and append-only nature of data warehouses, a common method to speed up query processing is to *denormalize* the database schema. This technique folds information from the dimension table(s) into the fact table so that a join is no longer needed to evaluate OLAP queries. In research, it has already become a common requirement for software-based OLAP acceleration methods [134, 138, 88, 126, 139]. Denormalization is now emerging in multiple commercial products as well (*e.g.*, [135, 136, 137]). WideTable [134] is a specific, widely-used style of denormalization. Although denormalization comes at the cost of increasing the database size, dictionary-based encoding can limit this overhead (to 18-22% in our experiments with SSB, and generally small for a wide range of schema) [134, 140, 141, 142, 143].

OLAP queries are data-intensive, involving relatively few processor cycles per byte of input data. For example, when a query asks for all customers in a given zip code, it may scan an entire table while only applying a simple comparison operation on each input record. As CPU speed and memory size have increased faster than both the memory speed and memory bus bandwidth, OLAP query evaluation in main-memory environments (the focus of this paper) is often memory-bound [128].

OLAP is Memory Bound To demonstrate the memory-bound nature of OLAP workloads, we evaluated hand-optimized C++ implementations of the thirteen queries in the Star Schema Benchmark (SSB) [144] on a typical server (two Intel Xeon Platinum 8260 CPUs, each with 24 cores and 376 GiB of memory), with scale factor 100, which corresponds to a ∼60GB database. Our query implementations used explicit SIMD instructions from the AVX-512 (512b), AVX2 (256b), and SSE (128b) instruction sets. Despite the decreased vector width, the AVX2 and SSE implementations were not significantly slower than the reference configuration–less than a 2% difference on average and less than 5% in any case. These results seemed to tell

us that the CPU implementation may be bottlenecked. To inspect this bottleneck more closely, we monitored the memory bandwidth utilization during each SSB query in the SSE configuration What we observed is that, throughout the entire benchmark, memory bandwidth utilization was universally near the experimental machine's maximum memory bandwidth. Thus, these experiments demonstrate the memory-bound nature of this database workload in two orthogonal ways: by showing that the CPU is not the performance bottleneck, and by showing that a single core can saturate the memory interface. This further suggests that, for this task, CPU-side hardware such as Intel's IAA would be unlikely to improve performance.[1]

Materialization Strategies Each query has two key steps: *Filter* and *Aggregate*. The *Filter* scans the columns required to satisfy the `WHERE` clause of each query and evaluates the conditions. The output of the *Filter* is a bitmap that represents the rows that meet the `WHERE` clause's conditions. The *Aggregate* takes the bitmap produced by the *Filter* as input. For each set bit in the bitmap, the *Aggregate* retrieves the corresponding record and performs the remaining work of the query, including any `GROUP BY` and `ORDER BY` clauses.

Late materialization propagates the filter bitmap throughout the entire query, resulting in an aggregation step that must perform a data retrieval operation in addition to the aggregation kernel. The traditional role that PIM has filled in database acceleration, including prior PIM work, has been to act as a filter processor, which relies on the CPU to perform the final fetch and aggregate query component [53, 88]. In contrast, *Early materialization* performs the data retrieval at an earlier stage in the query (when evaluating *Filter* steps), so the aggregation avoids this step. Depending on the query characteristics and PIM architecture, early or late materialization may be better.

We explore the impact of varying the materialization strategy on the aggregation component of the AVX-512 CPU baseline using the SSB benchmark. Our results are presented in Figure 5.1. We measured the proportion of time spent on the aggregation step for each query and each materialization strategy. Figure 5.1 suggests that using PIM to accelerate only the *Filter* step may not be sufficient to achieve significant speedups for some queries, such as Q3.1, which can only be accelerated by less than 25%. Therefore, we propose adding a *rank-level unit* (RLU) to perform early materialization, using the bitmap produced by filtering to gather the appropriate fields of the selected records into a contiguous block. This allows rank-level parallelism in this gather step and also optimizes the traffic to the CPU by using the full width of the interface with useful data, instead of requiring the memory controller to fetch scattered words.

[1]The IAA's primary benefits appear to be in decryption/decompression and offloading streaming-memory tasks from the cores; once the data have been fetched and decrypted/decompressed, the IAA can also perform the scan/filtering on the desired columns, and then fetch and decrypt/decompress any additional desired fields from the selected records.

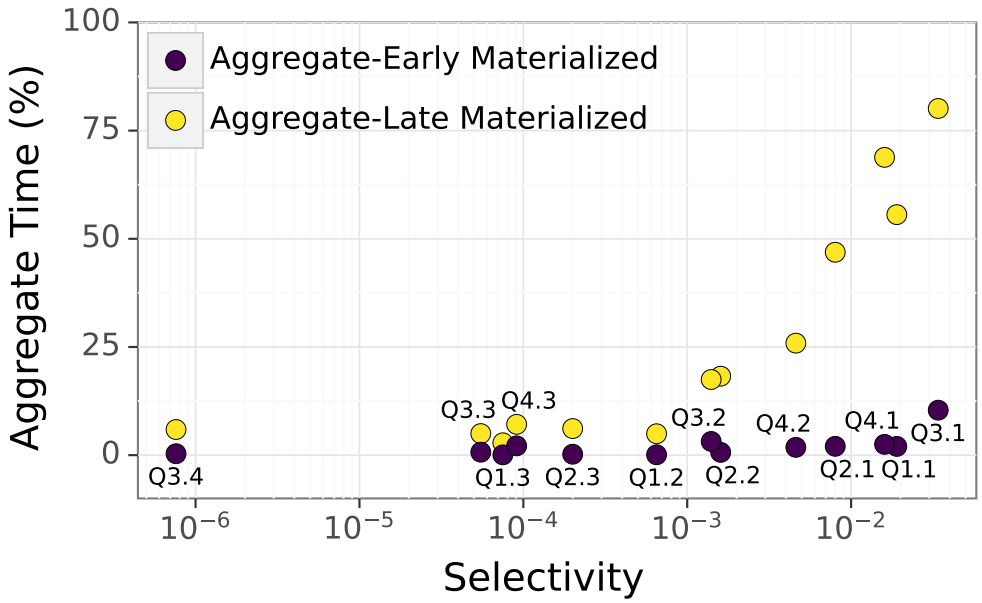


Figure 5.1: The results show how long the *Aggregate* step takes for each query, with early/late materialization. Using PIM may improve the remaining time (for the *Filter* step).

5.3 Architecture

In this section, we present two versions of the Membrane PIM architecture. Membrane-V (Vertical) is a bit-serial and element-parallel architecture that extends the exact pattern-matching capabilities of Sieve [53]/DRAM-CAM-style[39] architectures to process range queries. Membrane-H (Horizontal) is an element-serial bit-parallel architecture based on Fulcrum [132].

For Membrane-V, because records are laid out vertically (column-wise) along a bitline, the retrieval of the selected data would require many row activations to retrieve each bitslice of the selected records. Instead Membrane-V uses a second copy of the WideTable data that is stored elsewhere in memory and in the traditional horizontal format along a subarray row. The memory controller first issues the filtering command and receives the resulting bitmask, and then retrieves the data. For Membrane-H, in which the data is already in a row-wise horizontal layout, the processing unit can gather the data items immediately as they are selected. Section IV presents more details on the orchestration with the memory controller.

5.3.1 Membrane-V

Membrane-V performs query predicate filtering in an *element-parallel* but *bit-serial* manner. This operation can be carried out by laying the data vertically (column-major) along the bitlines of DRAM chips. Filtering is performed as a series of row activations, with each activation processing a given bit position (*bit slice* across many bitlines, i.e., data items. To support database query filtering, only simple bit-wise relational operations are needed. DRAM-based PIM architectures that process data bit-serially, leveraging a vertical data layout,

come in two categories: one utilizes a charge sharing (i.e., analog) triple-row-activation (TRA) operation, which does not require additional logic at the row buffer level, and one that integrates digital logic into the row buffer and can process data using normal DRAM single-row-activation (SRA).

Prior works such as Ambit [88], SIMDRAM [123], ComputeDRAM [93], and DRISA [87] adopt the TRA-based approach. While the TRA-based DRAM-PIM is more area-efficient due to the lack of additional digital logic along the width of the row boffer [88], other than the additional row decoders, it is more energy-intensive and slower than the sequential single-row activation with digital circuits integrated at the row buffer [53, 39]. The TRA overhead is primarily due to: (a) raising each additional wordline increases the activation energy by 22% [88], and (b) there is an overhead of multiple row-wide copy operations to move data operands to the PIM-capable rows [123, 88]. Furthermore, each TRA can only accomplish a bitwise AND/OR/NOT/MAJ operation, and a series of TRA operations must be chained to support more complex operations such as those needed for table scan [123]).

Alternatively, we can integrate bitwise digital relational logic directly into the local row buffers at the DRAM subarray level. For example, Sieve [53] and DRAM-CAM [39] embed a matcher after each sense amplifier for bit-serial exact pattern matching. We can extend this architecture by replacing the matcher hardware with a one-bit comparator that can support inequality to enable table scans (Fig. 5.2).

SRA-based Membrane-V Architecture. Fig. 5.2 illustrates the overall architecture of Membrane-V, including the vertical data layout and the circuit design to enable predicate table scans. The form factor required to integrate Membrane-V into the host system can be flexible. Each Membrane-V chip comprises multiple banks, similar to a commodity DRAM chip. However, within each bank, Membrane-V has an additional layer of hierarchy called *Subarray Groups*.

Subarray Groups. Similar to Sieve Type-2 [53], in Membrane-V, a subset of adjacent subarrays within a bank are connected through high-bandwidth links (isolation transistors) to form a subarray group. No modifications are made to a subarray in Membrane-V, except the last subarray's row buffer in each subarray group is extended with a filter logic array, mainly consisting of a row of 1-bit comparators with auxiliary bit-latches (described below). Subarray groups within a bank can independently scan their stored wide table entries by sequentially activating rows of a subarray, transferring them into the last subarray's filter logic array using the LISA mechanism [96], and performing the bit-serial comparison between attributes of the wide table entries (e.g., d_year) and the target values (e.g., 1994). The LISA mechanism enables the intra-subarray-group row-wide data relay by sequentially activating the local row buffers between the source and last subarray [53]. The latency of making one "hop" from one row buffer to the next consists of enabling the isolation transistors (link) and the activation of the sense amplifiers, which is only 1/8 of a regular row activation [53].

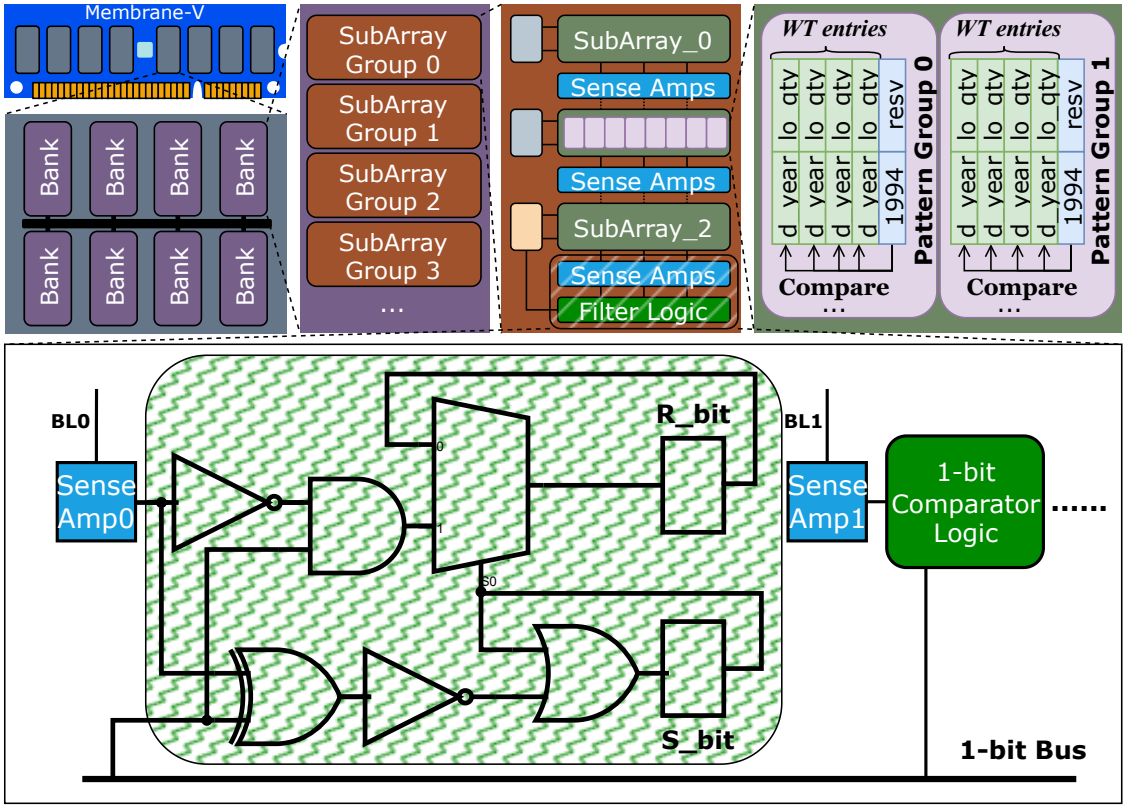


Figure 5.2: Membrane-V Architecture.

Data Layout. To support table scan in Membrane-V, we adopt a column-major bit layout similar to that proposed in [123, 126]. The dictionary-encoded WideTable entries are thus transposed onto bitlines for column-wise placement. Each attribute occupies consecutive bits in one column (i.e., spanning multiple rows) in the order of MSB to LSB. Because the transmission delay of the long wordlines prevents a predicate value bit from being dispatched to all comparators in a subarray row during one DRAM row cycle, the whole subarray is further broken down into smaller groups of columns called *pattern groups*. Within each pattern group, a block of WideTable entries is followed by a column of predicate value populated at run-time. All pattern groups work in a lockstep manner by comparing the predicate value with WideTable entries in that pattern group. This is somewhat similar to how a large logical subarray actually consists of smaller mats, each with a wordline amplifier. After a row is latched into the row buffer of the last subarray through row activation and LISA, the predicate bit is sent to all comparators within a pattern group through a 1-bit bus.

Filter Array. The crux of the Membrane-V architecture is its ability to perform a relational comparison (e.g., =, <, >, etc.) between a block of table attributes (e.g., *d_year*) and a predicate value (e.g., *1994*) bit-serially. This is handled by a filter array integrated at the last subarray in each group. The filter array consists of a row of 1-bit filter logic, which connects to a sense amplifier and a 1-bit bus for inputs. Fig. 5.2 bottom (pre-synthesized gate-level diagram is shown for clarity) shows the hardware logic of the 1-bit

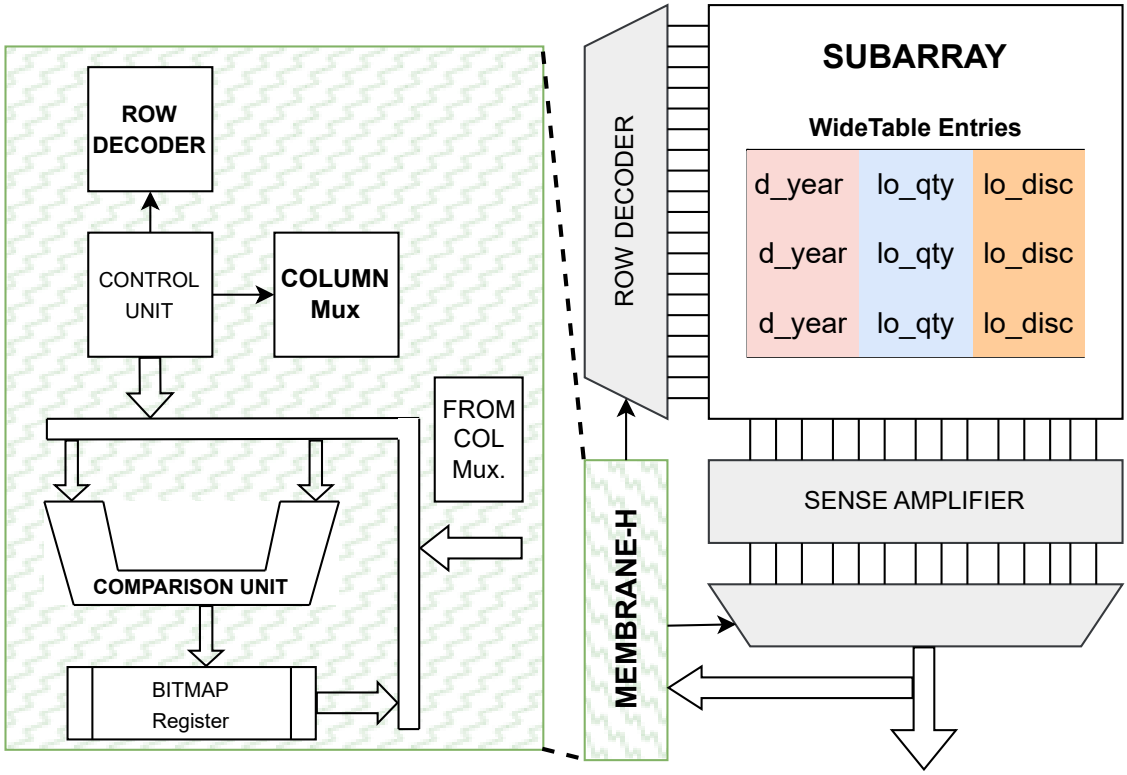
comparator that determines if one attribute stored in a bitline is smaller, equal to, or larger than the predicate value by making a comparison at every bit position and capturing the running result in the S_bit (stop bit) latch and R_bit (result bit) latch. Each subarray row activation delivers one attribute bit (through the sense amplifier) and one predicate bit (through the 1-bit bus) to each filter. If after n sequential row activations, where n is the attribute bit-length, the value of an attribute is $>/</=$ to the predicate value, then the R_bit latch stores 1/0/0, and the S_bit latch stores 1/1/0. The final bits stored in the R_bit latch and the S_bit latch are transferred to the Membrane controller to produce the bitmask. To support ranged predicate filtering, such as *1994*<*d_year*≤ *1997*, the Membrane-V runtime would break it down to multiple relational searches (i.e., *1994*<*d_year*, *d_year*<*1997*, and *d_year*=*1997*), execute them separately and aggregate the final results.

Write Broadcast. We observed that the latency of writing predicate values to the query regions of the subarrays takes substantially more time than the actual row-activation time spent on matching the predicate (Sec 5.5.3.) Since the predicate value bits written into each subarray are identical and are destined to the same row and column addresses, we propose a simple optimization that broadcasts the predicate bits simultaneously by connecting all participant subarrays' local row buffer to the global data lines, allowing them to accept the predicate bits at the same time. The hardware cost to the Membrane chip is negligible because the wires and the routing for the control signals are already in place [42].

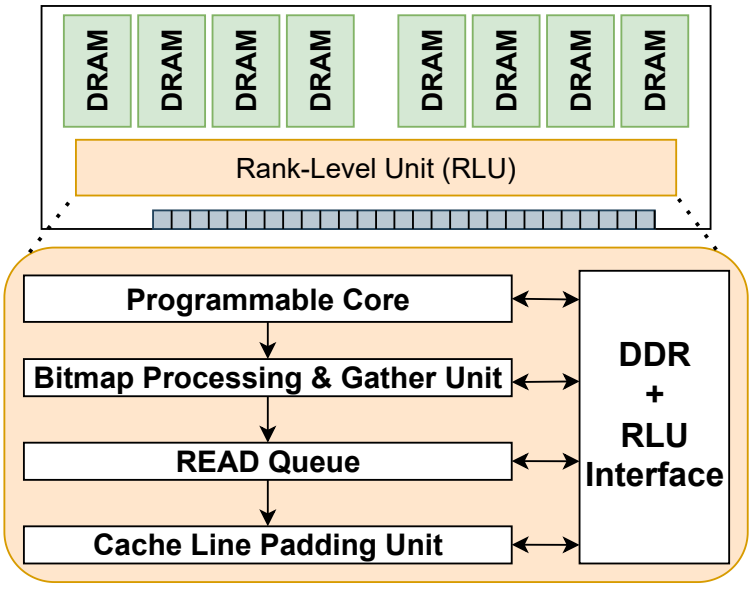
5.3.2 Membrane-H

The Membrane-H design operates on one word at a time in the row buffer with a processing element (PE) or an ALU present at the edge of a subarray optimised to perform scan operations. The Membrane-H unit filters the database attributes held within the row buffer and produces a bitmap, whose length is equal to the number of attributes that can be packed into a single page or row buffer width. Since entire attributes are checked in each row access, Membrane-H is able to pursue early materialization. While database terminology defines the early materialization to be at the granularity of each database record, we focus on early materialization at the granularity of a row buffer worth of records to align with the access pattern of the PIM architecture.

We couple subarray-level processing elements with a rank-level unit (RLU) to perform the gather operation necessary to materialize the result. The RLU is placed on the DIMM module, similar to [145] and [146]. The operation of Membrane-H breaks down into a scan phase performed in parallel across subarrays that produces a bitmap; and gather phase, performed by the RLU to gather the desired fields from the selected records, based on the bitmap. Our early investigations revealed that a single subarray unit performing both the phases would result in a large logic overhead and increased delay at the subarray-level. With the RLU, we not only pipeline scan and gather, but also optimize each unit for its respective task. An advantage of the



(a) Membrane-H Overall Architecture



(b) Rank Level Processor

Figure 5.3: Membrane-H Architecture.

RLU is that it provides rank-level parallelism, compared to the traditional method of having the CPU or a CPU-side accelerator such as the IAA perform the gather.

Subarray-Level Processing Element. These elements are distributed at the subarrays next to their respective row buffers and consist of a comparison unit to perform the predicate operation and shift logic

to feed data in the row buffer, one word at a time, to the PE. OLAP database workloads such as SSB [144], TPC-H [147], etc. generally operate on string values and signed integers, and these can be dictionary compressed, so the comparison unit only needs to operate on integers. A bitmap register is allocated to hold the contents of the resultant bitmap after the predicate operation. Based on the analysis from the SSB workload, we allocate a 32-bit bitmap register per subarray. When a query requires more than 32 bits, it could divided into sub-queries and processed sequentially. A Control Unit associated with each PE is responsible for orchestrating the operations and performing tasks such as iterating over attributes, updating bitmap registers, and interfacing with the RLUs. In case of a limitation related to feeding data sequentially off a row buffer due to mat-level barriers within the subarray, we propose to use a row of latches to store the row buffer contents and process the data. We could avoid the large area overhead by conservatively placing these latches for every 4 or 8 subarrays and bringing data to be processed via the LISA [96] mechanism.

Rank-Level Unit (RLU). The primary job of the RLU is to perform the gather operation and interact with the memory controller. It consists of a small programmable core to orchestrate query execution depending on the data layouts used within the subarrays. It also has a bitmap-gather unit where bitmaps are resolved, and address offsets generated to perform a gather operation. These generated addresses are inserted into an internal READ queue. The RLU also pushes the data gathered from the DIMM into the CPU. This process of transferring data to the CPU can be achieved via a mechanism such as Data Direct I/O (DDIO) [148] that features in most server-class Intel processors, which allows devices external to the CPU, such as network interface cards, to push data directly into the cache [149, 150]. DDIO is built on top of Direct Cache Access (DCA) [151] that allows I/O devices to provide prefetch hints to the CPU. Several works such as [152, 153, 154] have studied the benefits and optimization techniques related to DCA to seamlessly integrate it with the OS stack to get better performance. These mechanisms can be leveraged in the PIM context for pushing the data from the Membrane-H DIMM to the CPU. This is discussed further in the following section.

5.4 System Integration

Membrane's system integration consists of two levels of abstraction. First, a small set of Membrane-specific ISA instructions expose the PIM functionalities that can be used by a programmer. Second, logic in the DRAM controller is used to further decompose these new ISA instructions to DRAM-specific commands for each Membrane (V/H) architecture.

Membrane ISA. There are three main types of Membrane ISA instructions: *predicate*, *data movement*, and *configuration* instructions. Predicate instructions include equality and inequality flavors and take the predicate value, the type of predicate (e.g., equal, not equal, less than or equal, greater than or equal, between,

etc.), and the target column to which to apply the predicate. Data movement instructions are typically used offline by the DBMS to orchestrate loading the WideTable into the PIM. (Since generating the WideTable is considered a fixed cost that is paid once, we decided not to add dedicated hardware to accelerate it.)

In order to maintain a reasonably user-friendly interface of the Membrane ISA instructions and to keep the instructions' encoding lightweight, we add eight internal column-operand names $c_0 - c_7$ that can be used by the PIM instructions.[2] Similar to existing vector-length agnostic ISAs which maintain the state of their vector register names using Control Status Registers [155], we augment the memory controllers with bookkeeping logic to keep track of the start address, size, layout, and bitwidth of each column operand. Then, data movement and predicate instructions addressing a specific column operand will read these data structures to generate addresses for the required DRAM commands.

A DBMS would utilize the Membrane instructions as follows, assuming that space has been allocated in row-aligned blocks where each row spans all channels/banks/ranks—see below for more detail on this issue—and virtual-to-physical mappings have been established. 1) Use data movement instructions to load the WideTable into Membrane. This is a one-time cost for an analytics session. 2) Use configuration instructions or a system call to map specific columns from the WideTable that are used in the query to column operands $c_0 - c_7$ by specifying the starting virtual address, size, layout, and bitwidth of the data elements. This column setup step performs translation so that the memory controller knows the starting physical address of the column—see below for more detail on support for virtual memory. These column setup operations are broadcast to all memory controllers. 3) Call Membrane predicate instructions as part of the query execution to execute the selection operators in Membrane. For example, suppose we want to filter on d_year and return lo_date. We would set d_year's starting address, length, bitwidth, and data layout with a configuration instruction and map it to c_0, and do the same with lo_date and map it to c_1. Finally, we would execute a predicate instruction on c_0 and project the masks on c_1. If the RLU is present, the projection command is sent to the RLU. If not, the projection requires the memory controller to fetch the bitmap produced by the predicate operation on c_0 and use it to fetch the selected items. Again, these predicate instructions are broadcast to all memory controllers.

Virtual Memory. Modern DBMSs run on top of conventional operating systems and use the OS virtual memory management techniques. Traditional virtual memory has considerable flexibility in mapping virtual pages to physical frames in DRAM. But for subarray-level PIM, to maximize subarray parallelism during a scan, data used by the DBMS should be mapped in a "breadth-first" fashion across all the subarrays being used for PIM acceleration. Furthermore, to allow the memory controller to simply broadcast PIM commands to all channels/ranks/banks and subarrays, rather than sending a separate command to each subarray, we need to

[2]When a query operates on more than 8 columns, the query is divided into sub-queries.

ensure that for any PIM operation, all subarrays participating in that PIM operation are working on the same row. The typical address interleaving helps with both these considerations: it spreads successive cache-lines across channels, ranks, and banks before returning to the next column positions in a given row of a given subarray. However, when a given row "position" is filled up across all the channels/ranks/banks, interleaving typically moves on to the next row in the same subarray "position" across the channels/ranks/banks. This suggests that a system performing PIM acceleration would benefit from a slight change in the interleaving so that when one row is filled up across the channels/ranks/banks, the next row is in a different subarray. If the PIM architecture uses subarray groups, the interleaving should prioritize subarrays with the filter array, to avoid the overhead of the LISA data movement. Changing the interleaving in this way should not affect performance of regular applications. It is also desirable that allocations for PIM should be in aligned units that fill an entire row across all the channels/ranks/banks. This may require padding the end of the WideTable with some canary values.

In Membrane, the PIM operation is driven by the memory controller in with PIM commands that operate on one row of DRAM at a time, where a row spans all channels/ranks/banks. This means that rows not used for PIM can be used for regular (non-PIM) data in the same or other processes.

The PIM support in the virtual memory system consists of an allocation system call for the WideTable, with a descriptor for each column's size. The allocator returns a descriptor with the starting virtual address for each column, and ensures that each column is placed in a contiguous, row-aligned region of physical memory. If a satisfactory allocation cannot be made, the allocator can return an error or move other processes' data to free up sufficient space (this configuration choice is up to the system administrator.) The allocator records each column's virtual and physical starting address in a PIM mapping structure, so that when a column is specified as part of a PIM computation (see "Membrane ISA" above), the specified starting virtual address can be quickly mapped to a physical start address and the size can be checked. If no valid mapping is found, a segmentation fault occurs. This mapping can be a system call or—preferably—supported at user level with a new form of TLB for PIM, the PIM-TLB, that contains the column mappings. If the WideTable has too many columns to fit in the PIM-TLB, a TLB-miss will raise a PIM-TLB-fault that checks the table in the OS.

With the RLU, gathering and packing the data is performed on the DIMM side of the memory bus and benefits from rank-level parallelism. To simplify the task of fetching the data from the RLU, a mechanism such as Intel's DDIO [148] approach could be extended to allow the RLU to push data into the last-level cache. In this way, the memory controller does not need to wait for the RLU to have a block of data ready. DDIO involves using a descriptor table as a means of specifying physical addresses that the I/O devices can write to within the last-level cache, thereby avoiding expensive DMA transfers. Once an I/O device

performs a write, it is indicated to the CPU by an interrupt mechanism or polling (which would be more appropriate for Membrane). The CPU in turn updates the descriptor table to receive the next READ request while processing the just-read data. We extend this mechanism in the context of Membrane by provisioning additional PIM-specific descriptor tables, allowing the RLU to push data from its memory to the CPU's last-level cache at preconfigured addresses. Furthermore, because multiple RLUs share the interface to the CPU, they can pipeline their writes to the CPU and hide delays at individual RLUs, e.g. because the data being gathered are sparse.

5.5 Evaluation

5.5.1 Power, Latency, and Area Evaluation

Energy Evaluation. Table 5.1 reports area, delay, and power of each Membrane-H (M-H) ALU and the Membrane-V (M-V) filter logic (i.e., 1-bit comparator). The dynamic energy of entire Membrane-V filter array is calculated as the energy of the 1-bit comparator multiplied by the subarray width, and total dynamic energy consumption on the table scan for each query can be estimated as the number of times the filter array is accessed multiplied by energy cost of accessing the filter array once. The total dynamic energy consumption for Membrane-H is calculated as the ALU energy cost multiplied by the times the ALU is used for predicate filtering plus the energy consumed by RLUs. Our RTL simulation indicates that each bitmap processing and gathering Unit consumes 417.38 μW and with presence of a programmable core, we have approximated the overall power consumption of the RLU to 0.5 W. Membrane's peak power usage depends on its subarray-level parallelism as more active subarrays lead to a significant rise in static power.

Area Evaluation. To estimate the area overhead of a Membrane chip, we first obtain the area breakdown of the DDR4 chip (Micron_8Gb_x8) that is used to build the Membrane using Cacti-3DD [156]. Each subarray contains 512 rows, and there are 128 subarrays per bank. For Membrane-V, the logic can only fit along the sense amplifier's long side as indicated in [53]. We adopt a DRAM sense amplifier layout described by Song et al. [110] and a patent from Micron [111] for a conventional $4F^2$ DRAM layout. The short side and long side of the sense amplifier are 6F and 90F, respectively. We estimate 990F needs to be added on the long side of the local sense amplifiers to fit the 1-bit comparator logic. To support the LISA mechanism, an extra 60F on the long side is added to each sense amplifier for considering the area overhead of the links between the subarrays. For Membrane-H, since the ALU does not need to be pitch-matched to the bitline, the area overhead is much smaller. We estimate fitting 64 additional ALUs (shared by 128 subarrays) per bank, along with the overhead of enabling SALP and LISA, incurs only 5.26% area overhead. The Rank-level Unit incurs

Table 5.1: Membrane Hardware Characteristics.

	M-H ALU (16-bit)	M-V Filter (1-bit)
Area (μm^2)	301.08	11.36
Delay (ns)	0.36	0.47
Power (μW)	25.75	1.14

negligible area overhead (Table 5.1) compared to the overall chip area. Additionally, RLU would not affect the storage density of the Membrane since it is integrated into the DIMM module. Overall, the area overhead for Membrane-V with 1, 2, 4, 8, 16, 32, 64, and 128 filter arrays (i.e., SALP-1 – SALP-128) is 2.35%, 2.71%, 3.42%, 4.84%, 7.75%, 12.50%, 25.00%, and 46.0% respectively, and the area overhead for Membrane-H with 1, 2, 4, 8, 16, 32, and 64 ALUs (i.e., SALP-1 – SALP-64) is 2.02%, 2.04%, 2.07%, 2.14%, 2.28%, 2.55%, 3.10%, and 5.26% respectively. For both Membrane-V and Membrane-H, we observe that enabling a modest amount of subarray-level parallelism (SALP=2 or 4) achieves the best performance per watt, shown in Figure-5.7. These results and a sensitivity study regarding the area efficiency w.r.t. subarray-level parallelism are also shown in Figure 5.7(a) and Section 5.5.4. Additionally, introducing a row of latches to overcome the mat-level organization within the subarray structure to store and feed data sequentially into the Membrane-H ALU would introduce only a 1.9% area overhead to the DRAM cell-array area.

Latency Evaluation. The main latency of the filter array logic in Membrane-V is on the write broadcast in the Membrane-V path of the row activation. However, it adds negligible overhead (∼0.47 ns) compared to the DRAM row cycle (35∼50 ns). For Membrane-H, the ALU latency is also on the critical path. While filtering on one attribute takes an insignificant amount of time (∼1 ns), the total ALU latency to check all relevant attributes in a row can add up to a similar latency as a DRAM row cycle, depending on the data layout. Still, it is sufficiently hidden by the data materialization cost. We show more insights on the latency breakdown of Membrane-V/H in Section 5.5.3.

5.5.2 Overall Membrane Performance

Figure 5.4 shows the overall *end-to-end* performance improvement of Membrane over the CPU and two other possible PIM-based OLAP solutions (SIMDRAM [123], and RVU [157]) in log scale. We select SIMDRAM as an alternative DRAM-based bit-serial processing technology to the Membrane-V. SIMDRAM differs from Membrane-V with its charge-sharing (analog) triple-row-activation-based computing. The RVU is a 3D-stacked near-memory-processing solution comparable to the Membrane-H because it employs an element-parallel but bit-serial columnar data layout. We assume all PIM architectures only accelerate the table scan portion of the workloads. The host cooperates with the PIM using the bitmask, which can fully leverage the CPU and PIM processing potential. We compare the Membrane integrated system with the handcrafted optimized AVX512 C++ query implementation. Membrane-V/H and SIMDRAM are configured to be eight

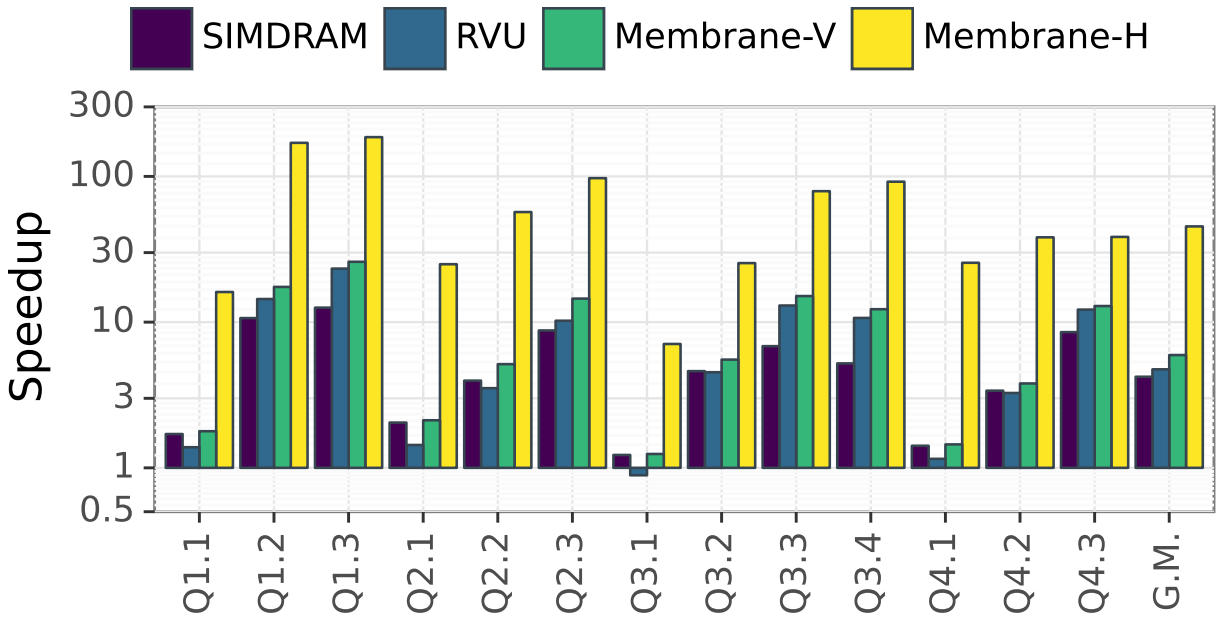


Figure 5.4: End-to-end SSB performance of various PIM techniques, compared against a hand-optimized AVX-512 CPU baseline. Results are depicted as a speedup against the CPU baseline. Aside from Membrane-H, which uses the RLU, all techniques are evaluated using a late materialization strategy.

channels with two ranks per channel using Micron's DDR4_8Gb_x8 DRAM chip as the building block. For the RVU, we assume 16 HMCs (configuration in Table. 5.1) connected with zero communication overhead to make up the same capacity as other architectures. We choose a moderate degree of subarray-level parallelism, namely, four subarray groups (SALP=4) in Membrane-V and SIMDRAM[123] and four concurrently working ALUs in Membrane-H. For Membrane-V, we also enable the write broadcast feature by assuming each write command updates the query regions of two subarrays. For both Membrane-V and Membrane-H, we report the performance of the row-major data layout.

First, the Membrane architecture consistently outperforms the CPU baseline in all queries. Membrane-V and Membrane-H offer 1.26×/25.97×/5.94× and 7.20×/185.75×/45.39× min/max/geomean speedup respectively. Second, Membrane achieves the best performance gain when the query selectivity is high (i.e., few database records passed matched to the predicate), such as in Q1.3, Q3.3, Q4.3. This characteristic is because Membrane can save data movement overhead by performing an in-place table scan, significantly eliminating the overhead of fetching data into the CPU. However, for queries with low selectivity, data retrieval for aggregation becomes the larger bottleneck. For example, Q3.1 selects the most data to aggregate; hence, the portion of the workload that benefits from Membrane-V's late materialization is limited. Third, Membrane-H almost doubles the performance of Membrane-V because H

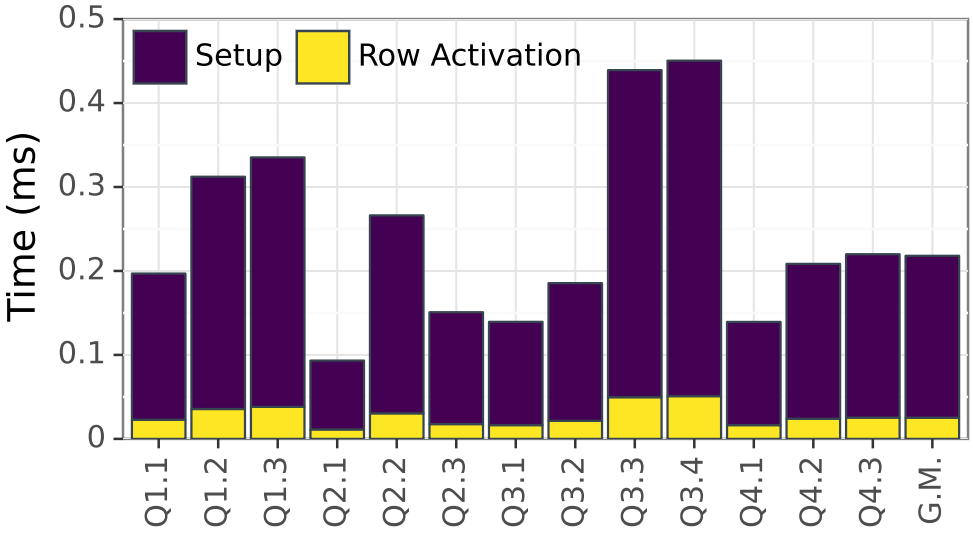


Figure 5.5: Membrane-V SSB query breakdown.

works well with the RLU and thus provides densely packed data for the host to consume, saving the cost of processing the bitmask and return trips from the CPU to memory for data retrieval.

Compared to SIMDRAM, Membrane-V is, on average (geomean) 34% faster. We notice that while SIMDRAM is 15% faster than Membrane-V to set up the predicate due to its ability to leverage the DRAM burst write feature better; it suffers performance loss in the actual predicate scan. Each triple-row activation is 6.8X slower than a single-row activation, and additionally, SIMDRAM spends 3X more operations at each bit location than Membrane-V. Compared to RVU, Membrane-H is, on average (geomean) 72% faster. While both RVU and Membrane-H scans horizontally laid out data, RVU has a much smaller throughput because of the narrow row buffer in HMC compared to 2D planar DRAM banks. Each row activation allows the Membrane to process more database records than RVU. Additionally, the computing logic of RVU is integrated at the logic layer; therefore, it is limited to DRAM bank-level parallelism, contrary to Membrane, which further exploits subarray-level parallelism.

The energy reduction (not shown in a figure) is highly correlated to the execution time of the query. Both Membrane-V and H versions achieve the best energy reduction for Q1.3 and the least for Q3.1. Specifically, Membrane-V offers a min/max/geomean of 2.01×/48.09×/10.05× energy reduction, while H offers a 8.46×/456.73×/71.40× energy reduction.

5.5.3 Membrane Performance Breakdown

Figures 5.5 and 5.6 show the execution time breakdown for both Membrane architectures (using the same configuration that is used in Figure 5.4). In the case of Membrane-V, write broadcast is enabled. Figure 5.6

also shows the effect of three different data layouts, namely, row-major, hybrid, and columnar, on Membrane-H. We make several key observations. First, the actual time spent activating rows for table scans (Filter) is relatively low for both Membrane-V and H. Membrane-V's execution cycles are mostly spent on populating the subarray query regions. Each query, i.e., predicate value, must be written vertically along the bitlines. The predicate values are several hundreds of bitlines apart, which is a highly inefficient way to access the DRAM. Furthermore, the write latency associated with setting up the queries in Membrane-V cannot be sufficiently reduced by the subarray-level parallelism because column commands within a bank must be served serially unless the write-broadcast hardware design is enabled (Figure. 5.7(b)). Membrane-V favors queries that filter on fewer attributes or attributes with low cardinality. Membrane-H spends roughly the same amount of cycles on row activation and processing the latched bits in the local row buffer using the ALU, regardless of the data layout. A key aspect of Membrane-H is the benefit of the RLU in performing data retrieval at a rank-level. The increased parallelism and bandwidth available with RLU helps in reducing the data retrieval time by 2.66x across the benchmark. More in-depth analysis of the effect of data layout on Membrane-H's performance is in Section 5.5.4.

5.5.4 Sensitivity Study

Effect of Data Layout

Figure 5.6 illustrates how different data layouts affect the performance of the Membrane-H architectures. The row-major layout activates the same number of rows across different queries because each row activation brings a whole row of complete database records to filter, and all database records must be checked. The hybrid layout is a combination of row-major and column-major layouts. Only rows containing attributes used in the table scans are activated in the hybrid and columnar layout. The performance of the columnar and hybrid data layouts improves as fewer attributes are scanned. The hybrid data layout improves the row activation time (Row-Act bars in Figure. 5.6) of eight queries by 1.4× and five queries by 2.6×. The columnar data layout improves the row activation time of three queries by 1.4×, seven queries by 1.9×, and three queries by 2.7×.

The hybrid data layout favors queries that scan attributes concentrated in a few subarray rows. We spent considerable time trying to find the optimal hybrid layout using the Louvain method for community detection [158]. The data layout choice also affects the latency spent on the ALU. The amount of time the ALU spends on each activated row depends on how many attributes in that row are participating in the table scan.

We observed marginal differences in performance between different data layouts across the query suite. The hybrid approach demonstrates that in the presence of prior knowledge related to the queries to be

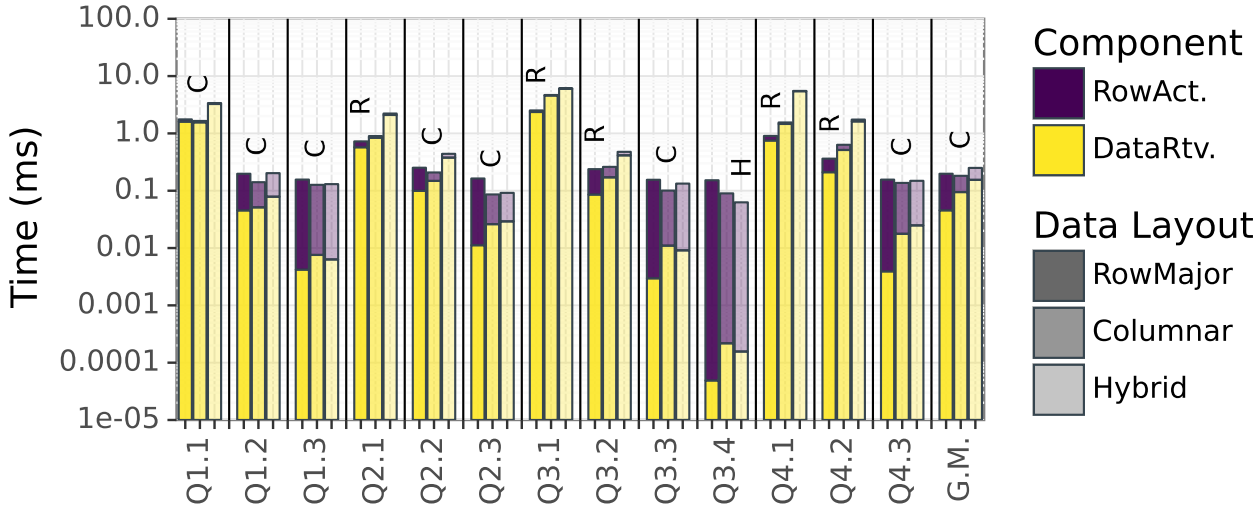
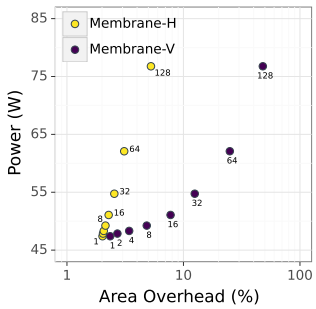
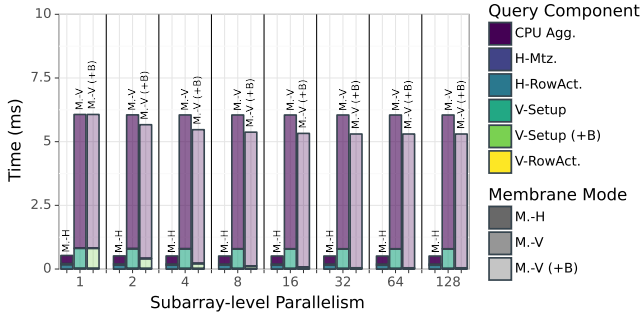


Figure 5.6: Membrane-H time to materialize each SSB query, by layout. For each query, the data layout with the shortest time to materialize each query is labeled above.



(a) The Membrane chip power and area overhead at varying levels of subarray-level parallelism.



(b) Query execution time breakdown in multiple Membrane modes, by query component. H/V-: Membrane-H/V, RowAct.: Membrane latency on row activation, Materialize (Mtz.): Membrane-H latency to materialize data, and +B: write broadcast.

Figure 5.7: Effect of subarray-level parallelism on area, geometric mean query power consumption, and performance.

executed, the data layout could be optimized for better performance. However, this is highly dependent on data schema and the queries being run themselves. 5.6 demonstrates the empirical results observed from our experiments with the data layout modifications on SSB.

Subarray-level Parallelism

Effect on Power and Area. Figure.5.7(a) illustrates the area (X-axis) and power (Y-axis) overhead that is associated with subarray-level parallelism (SALP). Note the area and power overhead of the Membrane-H RLU (integrated on the DIMM) are not included in the analysis since it is not affected by subarray-level parallelism. Membrane-H chip stays low in area overhead even if we aggressively increase the potential subarray-level parallelism. The ALU logic does not need to be pitch-matched to the sense amplifiers (Section.5.5.1),

unlike Membrane-V. For low SALP values, such as SALP=2 and 4, Membrane-V and H have a similar area overhead (∼2%). The geometric mean power consumption difference of Membrane-V and Membrane-H is small regardless of the SALP values, but both grow rapidly with higher SALP values. The peak power consumption associated with SALP=128 incurs over 60% overhead compared to SALP=1.

Effect on Performance. Figure 5.7(b) shows the geometric mean performance breakdown of query processing with varying degrees of SALP. Increasing the SALP 1) reduces the overhead of inter-subarray data movement (LISA) incurred by copying rows from non-PIM subarrays to the PIM-enabled subarrays, and 2) overlaps multiple row activations that go to the same bank but different subarray groups/ALUs. Doubling subarray-level parallelism adds 10% to 15% additional performance for Membrane-H and Membrane-V w/o write-broadcast optimization. The subarray-level parallelism is most beneficial when the write broadcast mechanism in Membrane-V is enabled, significantly reducing the predicate value setup time in Membrane-V. However, the benefit of subarray-level parallelism plateaus after SALP=4 (without Membrane-V broadcasting) or SALP=8 (with Membrane-V broadcasting) for both Membrane-V and Membrane-H for several reasons. First, bank-level access conflicts can be eliminated with a moderate number of concurrently working subarrays. Second, subarray-level parallelism can only parallelize the row activation of retrieving data from different subarrays. However, for Membrane-H, the column commands are still performed serially. Finally, the data set is not large enough to saturate the capacity of the Membrane device. From the above analysis, we conclude that SALP=4 balances the performance and overhead the best for both Membrane-V/H.

5.6 Related Works

Prior works in the database field such as BitWeaving [126] exploited the "intra-cycle"/bit-level parallelism of processors to accelerate the scan and filtering kernels. SIMD-scan [159] aimed to perform the same by utilizing on-chip vector processing units with SSE instructions. The BitWeaving-V/H flavors inspired this work to perform a similar PIM design-space exploration with Membrane-V/H, although Membrane-H uses the traditional columnar database layout using a row-major placement in the DRAM.

Processing In Storage Solutions. With database machines [160], there were attempts in the 1970s and 1980s to push query computation closer to where the data resided — at that time, spinning disks. This shows that database processing was important enough to warrant specialized hardware even 40–50 years ago. However, these efforts were abandoned as the resulting custom storage package was expensive to manufacture and commodity microprocessors were seeing exponential growth in performance. In the end, database machines were unable to match the price and performance trajectory of traditional servers. However, with the slowing of Moore's Law, there is a need to revisit ideas for specialization in today's context.

Pinatubo [119] and SmartSSD [161] are examples of other works that have proposed pushing query processing into the storage device. These designs, however, are limited by the storage I/O interface and suffer from higher latency and lower degrees of parallelism.

DRAM-Based PIM Designs. As mentioned previously, several prior works such as Ambit [88] and SIMDRAM [123] propose a triple-row activation design to perform logical operations at the subarray-level that could be leveraged for processing OLAP queries, but these approaches require multiple row activations per bit-level operation. JAFFAR [162] is a DIMM-level design that focuses on the scan operation by operating on the I/O buffer present on each DIMM. Although it gains by reducing data that travels over the memory bus, the amount of parallelism available in the I/O buffer is limited. The Reconfigurable Vector Unit [157] proposes to implement vector processing units at a vault-level in an HMC design. Our approach would extend to an HMC or HBM memory architecture but provides strong results even with more the more commoditized DIMM architecture. Most of these solutions do not evaluate end-to-end query processing pipelines or explore optimization techniques such as data layouts within the memory. Polynesia [163] is another work that aims to accelerate the analytical portion of HTAP database workloads using vault-level processing elements on a 3D stacked DRAM design. Membrane differs significantly from [163] in that it thoroughly explores the design space at the subarray-level.

Alternative Architectures. Prior works such as [164] accelerated the filtering step on the GPU but omitted the data-retrieval portion, which we have shown will often consume a large portion of query processing time. Crystal [165] was another recent work to accelerate analytical queries on the GPU. Ibex [166] and [167] implemented query processing on FPGAs. However, GPUs and FPGAs are limited by PCIe bandwidth, which is lower than typical memory-bus bandwidths. They also suffer from the limited scalability of onboard memory compared to the main memory addressable by the CPU. Papaphilippou and Luk [168] provides a comprehensive survey of works investigating acceleration of database systems using FPGAs and arrives at similar conclusions.

GPUs should be compatible with the Membrane approach. CPUs, discrete GPUs, and similar processing units can utilize Membrane PIM memory for scans and transfer intermediate results efficiently to CPUs or GPUs for further operations. Note, however, that Nvidia's A100 GPU thermal design power is 5.2/4.3× larger than Membrane-V/-H peak power points with SALP=128, respectively. For the scan operation, Membrane V/H outperformed Nvidia's A100 GPU by 3.3 × /8.5× for SSB. We further tested potential peak scan throughput of a P100 GPU not bounded by memory bandwidth. We populated the L1 cache with a vector of required length and tested the Scan operation implemented using [169]. We found that Membrane-H would still outperform by 3.73× even compared to this idealized, L1-resident baseline.

5.7 Conclusion and Future Work

Our proposed PIM architecture, Membrane-V/H is shown to have a geomean end-to-end speedup of 5.94x/45.39x over a highly optimized reference CPU codebase implementing the SSB Benchmark while achieving an energy reduction of 10.00x/70.77x. Membrane-V/H are also evaluated to have a speedup of 90.26x/359.21x over a state-of-the-art analytical database engine—DuckDB [170]—while having an area overhead of 3.42%/2.07%. We have conducted a detailed design space analysis of not only the PIM architecture but also describe the system integration and end-to-end performance. Membrane demonstrates the potential of subarray-level processing-in-memory architectures to accelerate analytics-based OLAP workloads, and shows the value of PIM for accelerating table scans. This work also shows the value of the H layout with early materialization using Rank-level Units (RLUs) to perform data-retrieval related gather operations. The growing memory wall, combined with rapid growth in data volumes, motivate processing-in memory architectures such as Membrane to accelerate data-intensive workloads such as OLAP queries.

Chapter 6

DRAM-BitSIMD: Exploring the Design Tradeoffs and Opportunities in DRAM-based Bit-Serial Vector Computing

6.1 Introduction

For applications with large datasets and low computational intensity (ops/byte), today's computer systems are bottlenecked by memory access bandwidth [171, 13]. These observations have motivated periodic attempts over the past several decades to place computational capabilities inside the DRAM, e.g. [172, 173, 174, 175]. More recently, the slowdown in Moore's Law and the vast difference in data bandwidth accessible to the processor compared to that inside the DRAM has motivated a renewed look at DRAM processing in memory (PIM).

One research direction has been to leverage the bit-level parallelism available in the local row buffers in each subarray. A typical DRAM access reads an entire row of 4K–8K bits from the selected subarray in each chip, multiplied by the number of chips in a rank (typically 4-8). Implementing some computation capability bit position enables massive bulk bitwise parallelism. This approach is often called *in-situ* PIM.

The design space for in-situ PIM architectures involves jointly optimizing the capabilities of the per-bit digital logic while imposing minimal overheads in area and power to leverage best the massive parallelism offered by the subarray. The goal of this work is to explore this complex design space.

The bit-serial vector computing paradigm allows massive bitwise data-level parallelism to be realized by laying out data in a vertical column-major fashion. This means that operating on an entire word requires a series of bit-serial steps. Prior work on bit-serial computing in DRAM leverages charge sharing on the bitlines, in which two or more operand rows are activated, and the charge sharing performs a simple Boolean computation [34, 36, 33, 38]. This analog approach is sometimes called processing using memory (PUM). While bit-serial computing requires a series of DRAM row activations, each row activation can operate on an entire row's worth of bits. These *bit slices* are 4-8K bits per chip, multiplied by the number of chips in the rank, enabling massive parallelism that dwarfs the small number of steps required to complete a full-word (e.g., 32-bit) computation. In addition, up to one-half of the subarrays in each bank and in each chip can be activated simultaneously.[1] With 32–64 subarrays per bank, subarray-level parallelism (SALP) substantially increases the computing throughput, although activating several subarrays simultaneously requires more power than traditional DRAM chips and system interfaces are designed to support.

If applications can indeed benefit from such high degrees of parallelism, new PIM-enabled memory products support higher power draw. Until then, we envision that the in-situ PIM design space broadly divides into two markets—"memory-first" and "accelerator-first" PIM. Memory-first designs focus on adding PIM features with minimal area/power overhead so that the resulting product fits in existing memory-system design constraints and has minimal impact on memory capacity. This limits subarray-level parallelism (SALP) and other PIM features. Moreover, memory-first designs require supporting PIM computation while simultaneously satisfying conventional memory accesses, entailing important system design considerations. First, the memory allocator must ensure that physically contiguous memory regions are always available for PIM computations, potentially necessitating periodic defragmentation. Second, although address interleaving is somewhat configurable in most modern systems, individual 8- or 16-bit chunks in a cache line are typically spread across the chips in a rank of DRAM, allowing for efficient retrieval of cache lines, but this means that a memory-first deployment with a conventional row-major data-layout cannot assume that the bytes of an individual word are even in the same DRAM chip.[2] For vertical data layouts, this may require that data transposition is implemented on the DRAM module or in the memory controller, which can fetch the bytes from the appropriate locations and then transpose, reuniting the bytes of a word into a single column

[1]The limitation of one-half comes from the way that sense amplifiers (SAs) are typically laid out for pitch-matching purposes; for more detail, see Section 6.2.

[2]Even adjacent bits may not be physically adjacent in the SAs; see Section 6.2.

within the subarray. Further, note that successive cache lines from a physical memory page are spread across channels and could also be spread across ranks and banks.

Accelerator-first PIM seeks to design the best data-parallel accelerator and uses DRAM as an implementation technology to achieve this without the constraints of the traditional memory interface. Data in an accelerator-first architecture can still be read and written by the processor, for example, via CXL [16], but data capacity and host read/write bandwidth would be lower and device power higher than what a traditional memory interface supports. For example, in this paper, we explore the degree of SALP. For purposes of this paper, we assume such an accelerator would be deployed as a separate accelerator board attached to the PCIe bus, very much like a discrete GPU. This allows it to draw much considerably more power, even as much as a GPU. Our exploration shows that the accelerator-first approach can outperform state-of-the-art GPUs by 5X for memory-bound data-parallel tasks, with much lower power and, thus, much better energy efficiency.

In addition to the deployment models, the complexity of the bit-serial logic embedded into the DRAM itself is another key axis we explore in this work, as it has important performance implications. First, we explore the number of bit registers that could be accommodated within the bit-serial logic to avoid a "register spilling" effect, where extra row accesses are needed to store intermediate results. Second, we explore various configurations of a *bit-serial logic unit* (BLSU) that differ in the number and types of operations they can support, offering interesting power-performance-area tradeoffs. In particular, we explore: 1) A NAND-only version as the minimum logic complete design, 2) A MAJ3 + NOT design as a digital point of comparison to charge-sharing triple row activation, 3) An XNOR + AND + SEL design that performs search and conditional update primitives of Associative Processing, and 4) A much more capable design adding XOR/OR.

We also observe that a sequence of logic operations on local registers in the BSLU can operate at a higher frequency than subarray reads/writes. Logic operations are limited by the latency of propagating control signals to all columns, modeled as *tCCD*, a timing parameter describing the latency between two DRAM column commands. This can be 5-10x faster than a regular row access cycle involving row activation and precharge. This decoupled execution model is unique to digital in-situ PIM solutions and infeasible for charge-sharing based solutions which tie the PIM computation to row access and is another novel contribution over prior bit-serial PIM approaches such as Micron's IMI architecture [18].

The main contributions of this paper are therefore:

- Our paper explores the design space for digital bit-serial in-situ techniques across several key axes, including deployment models (memory-first vs. accelerator first), bit-serial logic complexity (simple vs. complex operation sets and number of bit-serial registers) and evaluates the relevant power-performance-area tradeoffs of the various designs in a detailed technical evaluation that compares against CPU, GPU, and analog PUM (using SIMDRAM [34]) baselines.

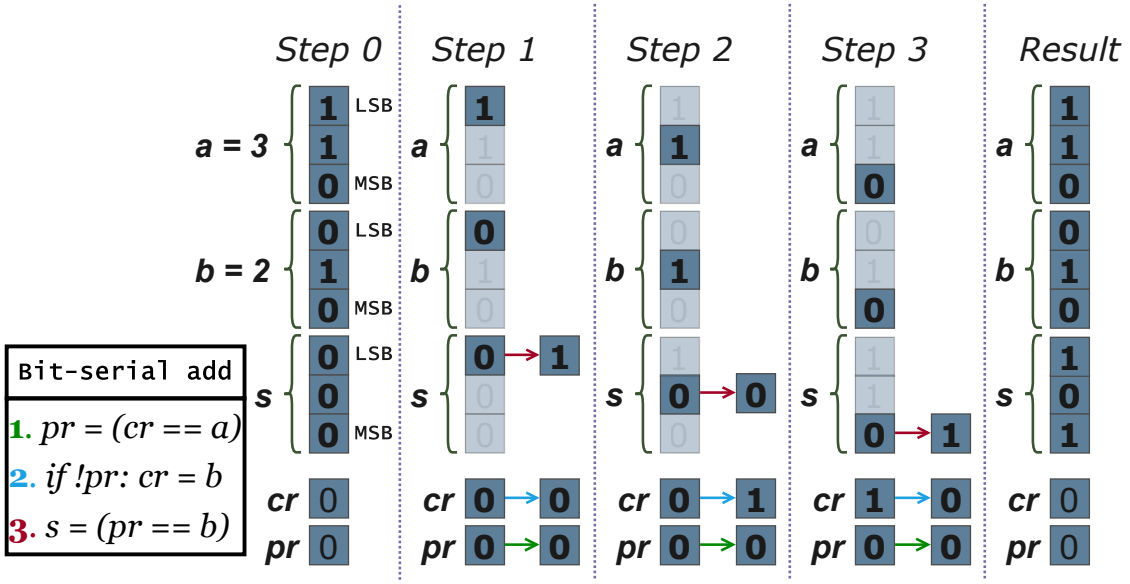


Figure 6.1: Bit-serial addition (a + b = s) example.

- Our exploration also evaluates the programmability aspect of in-DRAM bit-serial computing with the help of a lightweight but comprehensive bit-serial ISA that includes memory row read and write, bit-serial logic operations, and depending upon the design configuration, may support one or more local bit registers, as well as register manipulation operations such as move and set. This allows us to support 30+ high-level operations, including arithmetic (integer and floating-point), logical, relational, etc., implemented using a microcode approach.
- We also propose a *rank-level processing unit (RLU)*, which serves two purposes. First, it acts as a controller to issue commands to the DRAM to run a PIM computation kernel. Second, the RLU can access memory in a conventional way, thus enabling it to support any tasks that require cross-column data access, allowing it to implement tasks such as reductions, shifts, and transpositions. The RLU avoids the need to implement them in the memory controller.
- We also describe the system integration of various bit-serial designs with the host system, including interaction with the memory controller, integration into the address space, mechanisms for the host CPU to launch compute kernels and retrieve results, and support for concurrent utilization of PIM-enabled memory between different PIM-enabled processes as well as between PIM- and non-PIM processes.

6.2 Background

Bit-Serial Computing. Figure 6.1 illustrates the bit-serial addition of two 3-bit words. The result is computed sequentially from its LSB to MSB, and each bit position is computed by applying three logic

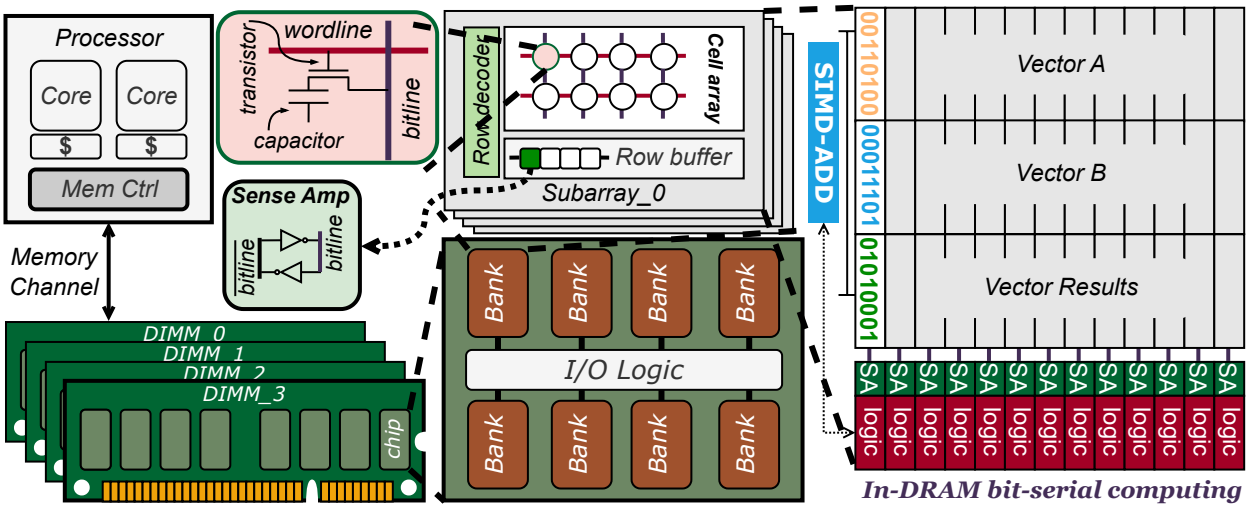


Figure 6.2: In-situ bit-serial computing in DRAM.

operations. This particular example leverages two additional registers, namely *cr* and *pr*, for storing carry and intermediary bits respectively. Other arithmetic, relational, and logical operations can be synthesized similarly by executing different logical steps at each bit position. While traditional (i.e., bit-parallel) computing can compute results in one shot, bit-serial computing can outperform it by simultaneously operating on a large vector.

In-DRAM Bit-Serial Computing. Bit-serial computing in DRAM involves operating on the values either (1) on the bitlines, with the result bit captured by the sense amplifier, in the case of analog PIM, or (2) in the local row buffer, in the case of digital PIM, with the operand(s) coming from either/both the local row buffer and/or a designated one-bit register, and the result either written back to the local row buffer or written to a designated bit register (or two registers, in the case of arithmetic, where a carry bit is also needed).

Prior work [37, 18, 34, 36, 33, 38] has demonstrated the benefits of leveraging a vertical data layout to perform massively parallel bit-serial SIMD-style processing. The key idea is to treat each bitline as a vector lane and align the source and destination data elements vertically on top of each other, as shown in Figure 6.2. A series of subarray row activations perform the computation sequentially at each bit position. The vertical layout allows each activation to access a bit slice across a row of vector elements (i.e., bitlines or vector lanes). Two additional advantages of the vertical layout are that it enables arbitrary bit access within the operands (e.g., left or right shifting within each word is cheap) and it supports flexible operand size, without having a word spread across multiple chips.

Limitations of Analog Approaches. Many prior architectures leverage DRAM's analog property by connecting three DRAM rows to the sense amplifiers, AKA triple-row-activation (TRA), to force charge-

sharing at the row buffer, equivalent to performing a row-wide bitwise logical operation. More complex operations, such as arithmetic, can be synthesized as a sequence of logical operations. However, analog-based bit-serial DRAM computing has the disadvantages of high latency and energy overhead [12, 39]. First, sustaining the activation of each additional wordline has been shown to require 22% more energy [33]. Second, there is a substantial latency in setting up operand rows in a designated compute region (a group of 16 DRAM rows with an additional row decoder) and copying the result row back to the regular data storage region in the DRAM subarrays. Moving operand rows to and from a dedicated compute region is needed for analog in-DRAM computing because (1) charge-sharing destroys the values in the original rows, and (2) selecting three arbitrary rows to activate requires a large row decoder [33].

Design Space of Digital Bit-Serial PIM. An alternative approach is integrating digital logic to each sense amplifier [18]. In this case, the sense amplifier and 1-bit compute logic are pitch-matched, and an arbitrary operand row can be selected and latched into the local row buffer for subsequent computing. For a single 8-Gbit DDR4 chip (8 banks/chip) with 16K bitlines per bank, there would be 128K 1-bit processing elements. The degree of hardware parallelism can be further increased with subarray-level parallelism, although the degree of SALP is limited by power delivery. Digital bit-serial processing significantly reduces the latency and energy spent on the intra-subarray data movement and only requires traditional, single-row activation. There is a design space to be explored by varying the capability of the integrated bit-serial logic to get different power, latency, area, and performance profiles while achieving varying degrees of flexibility, versatility, and programmability. This work highlights key design considerations and discusses the tradeoffs of different bit-serial PIM designs for massively data-parallel computing.

Bit-Serial Computing Performance. To illustrate the performance potential of an in-DRAM bit-serial architecture, we provide a simple back-of-the-envelope calculation below. In-DRAM bit-serial computing relies on cycling through operand rows for processing. For integer addition ($a + b = c$), a performance-optimized design (Section 6.6.1) requires two DRAM reads (fetch ith bits of a and b into row buffer logic) and one DRAM write (writeback ith bit of c to DRAM row) at each bit position. One DRAM row cycle takes a minimum of ~ 40 ns ($tRAS + tRP$). Therefore, adding two 64-bit integers requires a total of $64 \times 3 \times 40 = 7,680$ ns. In contrast, a modest CPU core clocked at 2.5GHz can perform a 64-bit integer addition in one cycle (0.4 ns), which is $19,200$ times faster.

However, DRAM bit-serial PIM is optimized for throughput. To break even with the CPU's performance on a vector operation, the PIM only needs to achieve 19,200-way parallelism $\times n$ cores to beat an n-core CPU—for example, the PIM would need to achieve 614,400-way parallelism to beat a 32-core CPU. A DDR4 8Gib_x4 chip has $16,384$ bitlines (i.e., vector lanes) per subarray, and a rank of such chips can process $262,144$ bits in a SIMD manner, outperforming the CPU by a factor of $81\times$. This means that even without SALP,

PIM's performance advantage over the CPU is large enough to accommodate the additional overheads in end-to-end execution, such as data transposition and the cost to launch a PIM computation and return the result to the CPU. Moreover, multiple DRAM subarrays can operate in parallel due to the rank-, bank-, and even subarray-level parallelism, achieving the effect of an extremely large vector machine. For example, a 256 GB bit-serial processing enabled DRAM system (16 ranks of 8Gib chips without subarray-level parallelism) can sustain a $16,777,216$ bits/DRAM row cycle peak processing rate, translating to 2.2×10^{12} 64-bit integer addition per second. That means a total of $9,166$ aforementioned CPU cores are needed to achieve the same level of parallelism.

6.3 Related Work

Charge Sharing Based Solutions. A key direction for DRAM in-situ PIM solutions is based on charge sharing, which activates multiple rows simultaneously and performs a simple Boolean operation on them. This approach minimizes DRAM circuit modification and area overhead. Examples include Ambit [33], bit-serial addition [37], SIMDRAM [34], DRISA [36], and ELP2IM [38]. However, charge-sharing-based solutions still require row decoder modification to activate multiple rows and often need dual-contact cells to achieve the NOT functionality. ComputeDRAM [35] demonstrates the possibility of multi-row activation with unmodified DRAM by intentionally violating DRAM command timing constraints, but it also requires storing the negation of all data due to the lack of NOT functionality. It works with some current-day DRAM products, but not all. These solutions often require multiple row copies, both because multi-row activation can destroy the original row contents, and to place the operands into special rows designated for computation. Furthermore, the reliability of charge-sharing-based solutions can be impacted by process variations [176, 38, 34]. The PIPF-DRAM work demonstrates that bit-serial operations can be done based on precharge-free DRAM (PF-DRAM) [177, 178]. The main idea of this architecture is to activate multiple rows consecutively rather than simultaneously, and the charge sharing happens among a sequence of activations. However, this solution faces the same challenges as other charge-sharing-based solutions: a limited set of supported operations and the need for extra row copies.

Other Digital Bit-Serial Solutions. Micron's In-Memory Intelligence (IMI) [18] demonstrates the potential of attaching bit-serial logic to SAs, even though it was ultimately not brought to market. DRISA-1T1C-mixed solution attaches XNOR/NOT gates to SA as a complement of charge-sharing based AND/OR. The exploration undertaken in this work significantly expands the scope of these works by considering a more complete and versatile microcode ISA with bit registers and proposing novel performance optimizations through decoupled execution of memory access and bit-serial operations.

Associative Processing Solutions. Associative processing is a bit-serial technique based on search and update operations. The search only requires comparison, and the update writes new values based on a bitmask (typically produced by the comparison). This approach can leverage content addressable memory (CAM) and lookup-table (LUT) PIM features. CAPE [179], pLUTo [180] and LAcc [181] are examples of this style. However, arithmetic beyond simple integer add/subtract can be expensive, and prior work has not implemented a floating-point. Sieve [12] and DRAM-CAM [39] are designed for accelerating pattern matching with vertical data layout, with pop count peripheral circuits for result reduction, but lack generality to support other types of computation.

PIM with Bit-Parallel Processing Units. Several proposed architectures place processing units that can operate on full words in one step, at subarray, bank, or rank level, without modifying the subarray itself, such as BLIMP-V [182]. Fulcrum [132] is an in-situ solution for 3D-stacked memories such as HBM and implements scalar, bit-parallel processing units at the edge of each pair of subarrays. However, it requires three local row buffers to hold the operand rows and the destination row, and support for left/right shift. An advantage of a fully featured processing unit is that they are not limited to data-parallel operations; for example, they can support conditionals, reductions, etc. However, they require changing the address interleaving, thus affecting regular memory transactions.

Commercial products such as AiM [183] and Aquabolt [184] introduce low-cost multiplication and addition units to accelerate specific deep-learning tasks. However, such solutions lack flexibility and cannot exploit the massive subarray parallelism.

PIM Compiler Support. Devic et al. [182] introduce a compilation framework based on LLVM [185] for the BLIMP PIM architecture, which features a bank-level design incorporating a general-purpose RISC-V processor. They assume that the host CPU would stall until the PIM application completes execution. Vadibel, et al. [186] develop a compiler framework that employs polyhedral optimization techniques [187]. Wang, et al. [188] focus on a PIM compilation framework based on the TensorFlow model. Both impose restrictions on the underlying data representation, limiting applications to matrix operations. The techniques described in this work, in contrast, impose no such constraints. Hadidi, et al. [189] implement a compilation framework for instruction-based PIM offloading, where individual instructions are offloaded for PIM processing. They identify instructions beneficial for PIM execution during compile-time. In contrast, our accelerator-first approach adopts a kernel-based offload model.

6.4 Design Space Exploration

The design space of in-situ bit-serial PIM architectures is characterized by several key parameters, including deployment models, power and area constraints, hardware design limitations, programmability aspects, and performance considerations. This section examines each of these in detail and enumerates potential design options.

6.4.1 Deployment Models

Memory-First Deployment. In the memory-first model, area overhead and memory capacity becomes key consideration as we seek to integrate PIM features into conventional DRAM designs. We also need to split memory space for regular usage and PIM computation and consider system integration details such as virtual/physical addresses and memory paging. Thus, the PIM computation capability can be installed in a few subarrays of the DRAM at most, so that area and power overhead are small. In our evaluation, we explore configurations that fit within an area/power overhead budget of 5% or less, and discuss potential system integration solutions in Section 6.6.

Accelerator-First Deployment. In this model, the PIM computation capability can be installed in a large portion of subarrays, providing us with the flexibility to explore designs that offer varying degrees of subarray-level parallelism (SALP). Although the chip organization such as channels, ranks, and banks can be adjusted or enlarged as a stand-alone accelerator, we follow the traditional DRAM organization for simpler analysis. The area overhead of bit-serial logic introduces tradeoffs of performance and capacity given fixed chip area. Because sense amplifiers (SAs) are shared by two adjacent subarrays, up to 50% subarrays can be activated simultaneously and perform PIM computation, while the rest subarrays can be used for storing data or supporting another PIM context in a time-sharing manner.

6.4.2 Complexity of the Bit-Serial Logic

The level of complexity of the bit-serial logic not only affects programmability but has important performance considerations. First, keeping the bit-serial logic simple implies that the number of bit-serial operations required to realize high-level arithmetic and logic operations would increase. Second, and more importantly, it could impact the number of row accesses required for storing intermediate results. Note that row accesses are more costly than logic operations as each memory-row read or write takes a full row activation and precharge cycle, typically 30–50 ns. On the other hand, bit-serial logic operations that only use the value in the local row buffer and local registers can operate faster, at a cycle time determined by the control signal propagation latency across all columns, which is modeled as *tCCD*, i.e., the delay between consecutive column

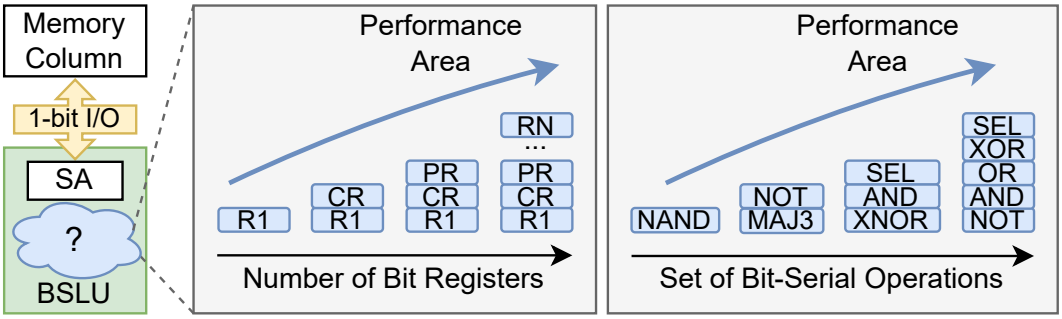


Figure 6.3: Bit-serial logic unit design space.

commands, typically 5-10X faster than a row access cycle, so performance is largely dominated by row access. The running time of a bit-serial program is the sum of the execution time of all row accesses and bit-serial operations in this program. This measurement is slightly pessimistic because some bit-serial operations can potentially overlap with row accesses with proper control sequence or pipelining technique.

We explore the design space of bit-serial logic units (BSLU) based on 1T1C DRAM architecture. In a PIM-enabled subarray, each column has a BSLU pitch-matched and attached to the SA. With vertical data layout, a row read operation can read a bit slice from the memory array to the local row buffer, and a row write operation can write all bits stored in the local row buffer to a specific bit slice—i.e., row—in the memory array. All the BSLUs operate in a lockstep, SIMD style. Figure 6.3 shows the model of a BSLU, where row accesses can be abstracted as 1-bit I/O, and SA can be considered as a 1-bit register. The design space of the bit-serial logic then includes (a) the number of bit registers and (b) the set of bit-serial operations. Since row accesses largely dominate performance, the goal of designing the logic within BSLU is to reduce the number of row accesses at a low area cost.

Number of Bit Registers

Introducing one or more additional bit registers can reduce the number of row accesses by leveraging computation locality within BSLU and avoiding the "register spilling" effect. At the same time, more bit registers require more area and register addressing logic. We analyze how different numbers of bit registers affect bit-serial computing as follows.

0-Reg: Ignored due to incapability of performing two-operand Boolean operations.

1-Reg: With one additional bit register besides SA, the BSLU can perform two-operand Boolean operations. With a logic complete set of bit-serial operations, the BSLU can compute complex tasks. But the performance is limited due to the need of storing all temporary values in memory rows.

2-Reg: By adding one more bit register, the BSLU can store a temporary bit value locally which can significantly reduce the number of row accesses, such as the carry bit during integer addition. In addition, the BSLU can support three-operand operations such as conditional selection (SEL).

3-Reg: Adding three bit registers provides more room to store temporary values during complex tasks such as integer multiplication and floating-point arithmetic. Although all BSLUs operate in SIMD style, complex tasks often require column-specific operations based on a condition. For example, for integer vector multiplication $A \times B = Prod$, we may read out a bit slice of A, and use it as a condition to determine in which columns we need to shift and add B to $Prod$. Thus, there is a need to have three bit registers to store the second operand, the carry bit, and the condition bit, shown as R1/CR/PR in Figure 6.3. This is our preferred architecture.

N-Reg: More efficiency can be gained with additional registers, but the logic overhead becomes difficult to pitch-match.

Set of Bit-Serial Operations

Due to hardware cost, a BSLU can only support a small set of native bit-serial logic operations. We study the following representative set of bit-serial operations and analyze performance and area trade-offs. More bit-serial operations result in better performance but higher hardware costs.

NAND-only – Minimal Logic Complete Design: This BSLU supports a single universal NAND operation which is logic-complete. It requires the 1-Reg architecture (i.e., the SA plus one bit-register).

MAJ/NOT – Digital Version of Triple Row Activation: This BSLU supports three-input majority (MAJ3) and NOT operations. The MAJ3 operation implements the same computation steps as triple row activation analog PIM by serially reading in three bit operands and computing the majority. NOT is for logic completeness. This BSLU uses 2-Reg.

XNOR/AND/SEL – Associative Processing Style: This BSLU supports XNOR, AND and SEL operations. The XNOR operation can check the equality of two bits. Combined with AND, this BSLU can serially match memory data with specific input patterns bit by bit, and use the AND operation to determine if all bits are exactly matched. The SEL operation is for supporting conditional write, so the BSLU can operate in an associative processing style using search + update primitives. This BSLU uses 2-Reg.

NOT/AND/OR/XOR/SEL – A General Purpose Setup: We consider this set of Boolean operations as a good balance point between hardware cost and general-purpose computation functionality and performance. The BSLU can use 2-Reg or 3-Reg. The latter is much more efficient for floating-point and integer multiplication. This is our preferred bit-serial ISA.

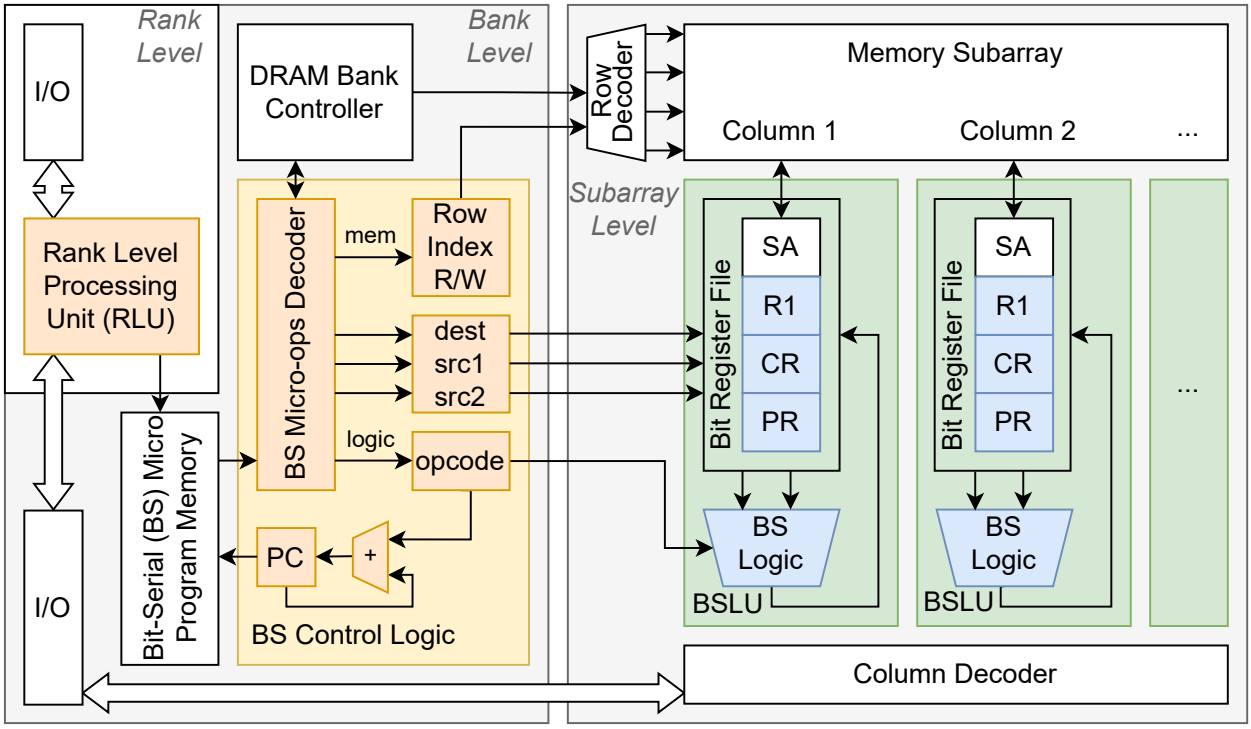


Figure 6.4: DRAM-BitSIMD architecture.

6.5 DRAM-BitSIMD Architecture

The high-level DRAM-BitSIMD architecture we use to evaluate the design tradeoffs discussed above is shown as Figure 6.4. It consists of the following components.

Subarray-Level Bit-Serial Logic Unit (BSLU). This is the bit-serial processing element per subarray column. It includes logic circuit to perform various bit-serial operations, bit registers, and register addressing logic. In PIM computing mode, within each subarray, BSLUs associated with each column are operated in lockstep.

The bit-serial ISA of each BSLU variant includes a unique set of bit-serial logic operations described in Section 6.4.2, common register move/set operations, and regular memory row read/write.

Bank-Level Bit-Serial Control Logic (BSCL). At the bank level, there is a BSCL module for decoding bit-serial micro-ops and sending control signals to all BSLUs within the bank. For memory read/write operations, the control logic decodes the row index and sends the signals for reading a memory row to the SA or writing the SA to a memory row. For bit-serial logic operations, the decoder decodes opcode and source and destination registers, then sends control signals to the BSLUs to perform the computation. The control logic also updates its PC for fetching next bit-serial micro-ops. The BSCL has a small instruction buffer

to store the program. If the program is too large for the buffer, the computing task must be broken into multiple compute kernels.

Rank-Level Processing Unit (RLU). The RLU is a microprocessor that sits on the DIMM and can send commands to each chip and bank, and thus can perform cross-column computation that the subarray-level BSLU does not support, such as reductions. The RLU is also responsible for translating the RISC-V instructions into low-level bit-serial microprograms, using a lookup table indexed by the RISC-V opcode. The subarray row indices in instruction encoding need to be updated based on actual row allocation.

6.6 System Integration

This section describes the software and hardware features that enable interaction with the host system. In this work, we adopt a kernel offloading model, where programmers manually partition the workload to ease the system integration effort. For now, we manually program the PIM kernels; we envision that in the future, a vectorizing compiler with #pragma pim commands could replace much of that effort and that, eventually, the pragma would not be required. We adopt this simplified approach to the programming aspect because this work is focused on architectural exploration and tradeoff analysis.

6.6.1 Programming and Compiling

Bit-Serial Microcode

Because the bit-serial architecture uses only a small number of elementary logic elements, writing the microprogram for a bit-serial operation benefits from logic synthesis tools, which can identify the sequence of operations using these hardware elements and any intermediate values. Figure 6.5 demonstrates how to map a high-level computation task to a NAND-only bit-serial microprogram. Given a task (a) for 1-bit addition, a NAND-only circuit (b) is created by synthesis tools, then the circuit is converted into a bit-serial NAND sequence with temporary variables (c). Depending on the architecture's actual number of bit registers, some temporary variables may be spilled to rows with extra reads and writes. Compiler backend techniques such as instruction scheduling and register allocation can help to reduce the number of temporary rows.

High Level Operations

We implement a rich set of high-level operations, compatible with a typical vector instruction set, as shown in Table 6.1, including bitwise logical, integer arithmetic, integer relational, floating-point arithmetic, and other supporting operations. With the decoupled row access and bit-serial operation execution, we collect

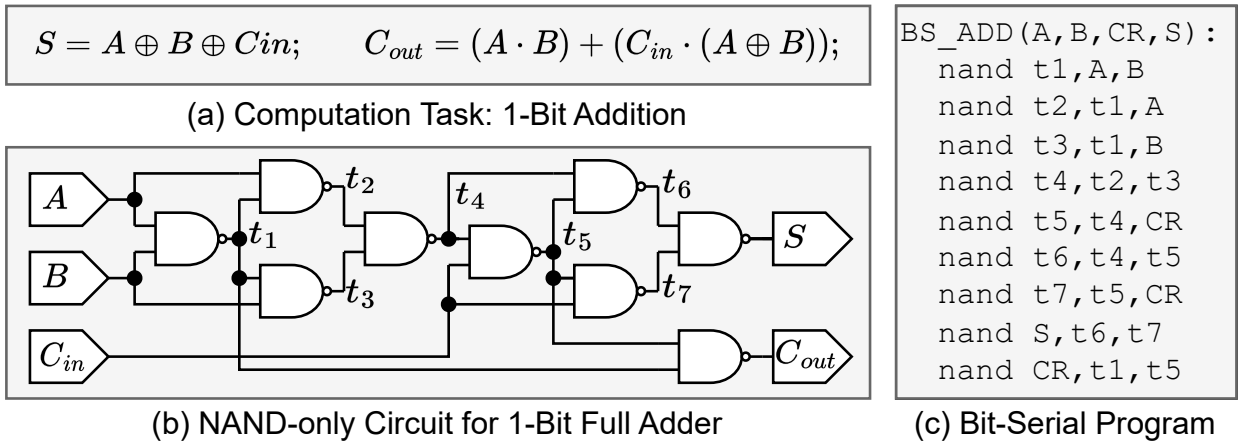


Figure 6.5: Bit-serial programming process for 1-bit addition on the NAND-only architecture.

the number of row read, row write, and logic operations separately in the table from a functional simulator for measuring performance. Note that the bit-serial microprograms can likely be optimized further, and the architecture can potentially support more data types and operations.

Table 6.1: Cycle Count for n-bit High-Level Operations on DRAM-BitSIMD 3-Reg Design

Category	Operation	Complexity	#Read	#Write	#Logic	#R/W/L	Description
Logical	int32 not	Linear	n	n	n	32 / 32 / 32	$dst_{32} = \neg src$
	int32 and,or,xor	Linear	2n	n	2n	64 / 32 / 64	$dst_{32} = src1\ \&, \mid, \oplus\ src2$
	int32 nand,nor,xnor	Linear	2n	n	3n	64 / 32 / 96	$dst_{32} = \neg(src1\ \&, \mid, \oplus\ src2)$
Arithmetic	int32 add,sub	Linear	2n	n	3n+1	64 / 32 / 97	$dst_{32} = src1+, -src2$
	int32 abs	Linear	n+1	n	4n+2	33 / 32 / 130	$dst_{32} = abs(src)$
	int32 min,max	Linear	4n+1	n+1	4n+3	129 / 33 / 131	$dst_{32} = min, max(src1, src2)$
	uint32 mul	Quadratic	$\sim 1.9n^2$	$\sim n^2$	$\sim 3.5n^2$	1940 / 1095 / 3606	$prod_{64} = src1 * src2$
	uint32 div,rem	Quadratic	$\sim 3n^2$	$\sim 1.7n^2$	$\sim 4n^2$	3168 / 1712 / 4257	$quo_{32}, rem_{32} = src1/src2$
Relational	uint32,int32 gt,lt	Linear	2n	1	2n+2	64 / 1 / 66	$dst_1 = src1 >, < src2$
	int32 eq	Linear	2n	1	3n+2	64 / 1 / 98	$dst_1 = src1 == src2$
FP	fp32 add,sub	Quadratic	$\sim 1.3n^2$	$\sim 0.7n^2$	$\sim 1.6n^2$	1331 / 685 / 1687	$dst = src1+, -src2$
	fp32 mul	Quadratic	$\sim 1.8n^2$	$\sim n^2$	$\sim 3n^2$	1852 / 1000 / 3054	$dst = src1 * src2$
	fp32 div	Quadratic	$\sim 2.7n^2$	$\sim 1.4n^2$	$\sim 4n^2$	2744 / 1458 / 4187	$dst = src1\ /\ src2$
Misc	uint32 copy	Linear	n	n	0	32 / 32 / 0	$dst_{32} = src$
	uint32 search	Linear	n	1	3n+2	31 / 1 / 98	$dst_1 = src == pattern$
	int32 if_else	Linear	2n+1	n	2n	65 / 32 / 64	$dst_{32} = cond_1\ ?\ src1 : src2$
	int32 ReLU	Linear	n+1	n	n+1	33 / 32 / 33	$dst_{32} = src > 0\ ?\ src : 0$
	uint32 bitcount	Log-Linear	$\sim 4n$	$\sim 3n$	$\sim 1.3nlogn$	114 / 90 / 218	$dst_8 = bitcount(src)$
	uint32 var_l,rshift	Log-Linear	$\sim 2nlogn$	$\sim 1.2nlogn$	$\sim 2nlogn$	326 / 192 / 299	$dst_{32} = src1 \ll, \gg src2$

Integer Arithmetic, Relational, and Logic Operations. Figure 6.6 shows the microprograms for integer addition and subtraction. They minimize subarray row accesses to just reading out each bit of the two operands and writing back the results by taking advantage of the bit registers (CR/PR) and complex bit-serial operations (XOR/SEL) in the 2-Reg or 3-Reg model. Integer relational operations are implemented

```
int_add(A, B, S):
  set CR, 0
  for i in 0...31:
    read A + i
    xor PR, SA, CR
    read B + i
    sel CR, PR, SA, CR
    xor SA, PR, SA
    write S + i
```

```
int_sub(A, B, S):
  set CR, 0
  for i in 0...31:
    read A + i
    xor PR, SA, CR
    read B + i
    sel CR, PR, CR, SA
    xor SA, PR, SA
    write S + i
```

Figure 6.6: Bit-serial int add/sub microprograms on DRAM-BitSIMD.

based on subtraction, with the necessary sign bit check. If not directly supported by the logic gates in the BSLU, Boolean logic operations are implemented using a short microprogram.

The basic shift-and-add approach for integer multiplication has $O(n^2)$ complexity. We implement unsigned integer multiplication with one level of Karatsuba recursion [190, 191], then fall back to the shift-and-add approach. We implement the shift-and-sub approach described in [192] for division. Bit-serial addition or subtraction can be done on two ranges of bits, i.e. row indices, without the need for shifting the data.

Floating-point Arithmetic. One of the main challenges with FP arithmetic is that mantissa alignment and result normalization require data-value-specific shifting steps, which contradicts the principles of SIMD. We implement the variable shifting in log-linear complexity by performing conditional shifting with 2^i stride, similar to the algorithms in [191].

Miscellaneous Operations. We can also effectively search for an exact pattern among data elements in all columns by encoding the pattern as part of a bit-serial microprogram. The bit-serial ISA supports bit population count (pop count) and variable shift in log-linear complexity.

Application Development

We assume a kernel-offloading model and envision that the kernel code can be written in two ways. If the logic is simple, such as a single *for* loop with no inter-loop conditional or data dependencies, the user can annotate the loop with a pragma, similar to the OpenMP parallel-for, and the compiler would generate vectorized code. An LLVM auto-vectorization routine without user intervention is also possible. In this work, we assume an expert programmer manually identifies kernels to offload and rewrites the applications using custom macros.

Figure 6.7 shows two pseudocode snippets (some details omitted for brevity) comparing the baseline CPU program, and the equivalent DRAM-BitSIMD accelerated code for *Histogram* [193]. The Histogram kernel computes the frequency of each 8-bit R/G/B pixel pattern in an input image. The input image is a PNG file

```
1 for (int i = img.start; i < img.end; i+=3) {
3         blue[img[i]]++;
4         green[img[i+1]]++;
5         red[img[i+2]]++;
6 }
```

CPU Histogram Pseudocode

```
1  ... // make a struct rlu_arg to set up RLU kernel arguments
2  rlu_args.img = *img_slice; rlu_args.num_RLU = ALL_RLUS;
3  bitSIMD_alloc(&rlu_set, ALL_RLUS)
4  bitSIMD_launch(&rlu_set, &rlu_args, rlu_kerne=rlu_hist_kernel)
5  void rlu_hist_kernel() {  // RLU kernel code
6      char gid = 0, vl = rlu_arg.img_slice.size/rlu_arg.num_RLU, bl = 8;
7      ... // make a struct rlu_res to store intermediary results
8      int blue[256]; rlu_res.blue = *blue; ... // also for green and red
9      ... // declare variables: img_v, b_v, counters, etc...
10     bitSIMD_alloc(*img_v, gid, rv, bl);
11     ... // bitSIMD_alloc for other vectors
12     bitSIMD_vld(&img, img_v, vl, bl); // loading input image to vector
13     bitSIMD_vfill(100, b_v, vl); // blue mask pattern 100100...
14     ... // 010010... for g_v, 001001... for r_v
15     for (int i = 0; i < 256; i++) { // search all 8-bit pixel patterns
16         bitSIMD_brdcst(0, res_v, vl, bl=1); // init paral search res
17         bitSIMD_brdcst(i, key_v, vl, bl=1); // init search pattern
18         bitSIMD_vxnor(res_v, key_v, img_v, vl, bl); // parallel match
19         bitSIMD_vand(tmp_v, res_v, bm_v, vl, bl=1);
20         bitSIMD_pcl(&b_cnt, tmp_v, vl, bl=1); // accumulate blue hits
21         ... // repeat for green and red pixels
22         rlu_res.blue[i] = b_cnt;
23     }
24 }
25 bitSIMD_transfer(RLU_TO_HOST, rlu_res, ALL_RLUS)
26 for (int i = 0; i < ALL_RLUS; i++) { ... // host merges RLU results }
```

DRAM-BitSIMD Histogram Pseudocode

Figure 6.7: Compare CPU and DRAM-BitSIMD histogram kernel.

with interleaved blue, green, and red pixels. The key idea of implementing the *Histogram* in DRAM-BitSIMD is to perform a parallel bit-serial search of each pixel pattern from 00000000 to 11111111 and accumulate hits. The host first sets up the kernel by allocating RLUs and distributing a slice of the input image to each RLU (lines 1 to 4). Each RLU runs the same kernel code and generates three counter arrays independently. The

data is transferred to the host (line 25) post computation. Note no cross-RLU communication is needed. The host handles the final data merging (line 26).

Compilation. As previously described, DRAM-BitSIMD uses two levels of ISAs for programming and execution. The first level (Section 6.5) is the DRAM-BitSIMD bit-serial micro-ops ISA. The second level (Section 6.6.1 and Table 6.1) consists of extensible DRAM-BitSIMD high-level operations, aka macros, that manipulate vectors (analogous to RISC-V Vector Extensions). The DRAM-BitSIMD macros are exposed to the programmer as API functions. Each macro is a fixed functional routine (comparable to NVIDIA PTX) agnostic of DRAM-BitSIMD architectural details, which is implemented as a microprogram of low-level BitSIMD ops in the BitSIMD controller at the bank level (Figure 6.4). This decouples the backend code generation from being tied to a specific PIM architecture, leaving room for future hardware and software improvement and providing an abstraction for application and compiler developers.

DRAM-BitSIMD kernel and host codes are compiled separately. The kernel is compiled into sequences of high-level DRAM-BitSIMD instructions (Section 6.6.1) mixed with RLU-compatible instructions (e.g., RISC-V) since the kernel execution is handled by both RLU and bit-serial logic at the subarray. The compiled kernel code is stored in the memory and fetched into the RLU instruction cache at run-time.

6.6.2 Virtual Memory and PIM-kernel Launch

Virtual Memory. Unlike prior PIM work, of which we are aware, our goal is to make BitSIMD designs work with existing OS virtual-memory systems with as few changes as possible. Each bitSIMD_alloc command allocates a data structure to a contiguous virtual memory region. Each data structure can be described with a simple base and size. This does not necessarily map to a contiguous region of physical memory, as we explain below. The allocation fails if the requested allocation is too large for the PIM-enabled memory capacity. Large data structures cannot be allocated a single, contiguous region of physical memory (more on this below), so if the OS cannot allocate the necessary physical-memory regions as needed the allocation also fails. This may motivate OS support for defragmenting memory to support PIM, but that is left for future work. To try to keep space available for PIM operation, the OS's strategy for allocating physical memory to non-PIM processes should try to keep blocks of space free for as long as possible. Another option is to reserve space in systems with high utilization of PIM.

Vertical data layout requires us to allocate n rows together for n-bit words; we call this a *word batch* of rows. In traditional interleaving, successive physical addresses rotate among channels, ranks, banks, etc. but stay within a given row position until that row is filled and then move to the next row in the same "horizontal set" of subarrays. This works well to accommodate vertical data, but SALP requires that once a

data structure has filled a word batch of rows across a horizontal set of subarrays, the next allocation to a word batch should be in a different subarray so that a data structure is spread across as many subarrays as possible to maximize SALP.

We also require the ability to align operands so that operands that are part of the same kernel are mapped in the same way to the same subarrays. This requires the OS memory allocator to understand that a group of operands is related, which is specified by the groupID in the alloc call, as well as word batches and the address interleaving so that once A is allocated, B and S can be allocated to a physical address at appropriate offsets that will align with A.

To achieve these goals, the OS maintains a table mapping base addresses for PIM allocations in virtual memory to base addresses in physical memory. Both the OS and the RLU must agree on how to partition a data structure into chunks that fill a horizontal set of subarrays and then place successive partitions at appropriate offsets in the physical address space so that a data structure is indeed spread across different horizontal sets of subarrays to maximize SALP. If only some subarrays are enabled with PIM, the allocation should ensure that data for PIM computation are only placed in PIM-enabled subarrays. This is deterministic so that the set of allocated regions can be determined by the OS and the RLU simply from the base address and size. These allocated regions of physical memory are pinned and marked non-cacheable. They are also removed from the OS-free list. When the data structure is later freed or the PIM process exits, these allocated regions are released.

This means that once a data structure is successfully allocated, CPU operations on the PIM data structure (loading data or launching a PIM computation kernel) only need to specify the base address and size. This is checked in the mapping table to find the physical base address, so translation and permission checking is very low overhead.

Data allocated in PIM memory are not accessible by regular loads and stores. They may only be accessed through translation functions that load regular data into the PIM in vertical format, retrieve a block of PIM data and convert it back to a traditional layout. For data previously computed by the CPU and where a large portion of the data may reside in the last-level cache, a version of these functions should exist that checks the cache. A streaming version should also exist that bypasses the cache, reading/writing data between traditional and PIM vertical layouts. Both require the involvement of the RLU to perform the appropriate sequence of row accesses to fetch the vertically laid-out data.

Kernel Launch. A PIM computation kernel is invoked with the virtual base address and size for the PIM program and the virtual base address and size for each argument. The program must be smaller than a traditional OS page; its physical address is found using the page table. The kernel calls first invoke the OS. The data arguments are checked in the OS PIM-mapping table, producing the physical base addresses

for the arguments. These are passed with the structure sizes to the RLU by writing them into a descriptor in memory, along with the specific command to be performed (loading data, kernel execution, etc.). Then launching the kernel is performed with a *jpim* instruction that transfers control to the memory controllers and stalls the CPU core. There is no communication between channels during a PIM operation, so it is sufficient for the CPU to broadcast the *jpim*; the memory controllers do not need to coordinate. However, the memory controller should not reorder memory operations across a PIM operation. Initiating the PIM operation on the RLU only requires a 1-bit "go" signal per rank from the memory controller. The RLU fetches the program into its instruction buffer and then begins executing. Each PIM operation is sent to one or more bank control units. The bank control unit understands how PIM structures are mapped to a vertical layout and how they are partitioned across subarrays so that a single PIM command can leverage SALP. Because traditional address interleaving means PIM operations use all channels, ranks, and banks (depending on data size), regular memory read/write (from any process, including the PIM process) must stall until the RLU indicates the completion of a PIM program.

When the RLU signals the completion of the PIM program, which requires an additional 1-bit signal, the memory controller transfers control back to the CPU. The *jpim* instruction completes when all the memory controllers have returned. And the CPU can retrieve the results with a command to read the appropriate data from the PIM, which the RLU services. Note that this approach means this core is not interruptible, and a PIM kernel is atomic; it cannot be interrupted.

Memory Controllers. Prior work such as SIMDRAM [34] adds decoding and execution logic for each PIM instruction at the memory controller (MC). However, direct PIM support in the MC may not be optimal for scalability and backward compatibility; future PIM products with new functionalities (e.g., instructions) require a new MC design. In this work, the host CPU delegates the MCs to oversee the overall DRAM-BitSIMD kernel execution, which ensures proper execution and synchronization of the kernel among participant RLUs. The RLUs decode and execute instructions that perform the actual DRAM-BitSIMD operations. The DRAM-BitSIMD compatible memory controllers must support PIM and interface with the RLU. However, this only requires a few extra signals and some modest logic to schedule memory operations, whether PIM operations or regular read/write. In fact, a typical system contains multiple memory controllers servicing multiple channels. Finally, we adapted a Data Transposition Unit (DTU) design from SIMDRAM that converts input-output data from vertical to horizontal layout and vice versa if needed. We place the DTU in the memory controller so that it has access to any data that are cached in the CPU.

Table 6.2: Selected Benchmarks (Notation: I: integer, F: floats, *: not executable in the stock version of SIMDRAM)

Benchmark	Input-Output Dataset
VA: Vector Addition (I/F) [195]	3.3×10^8 (32 bit)
MV*: Matrix-Vector Multiply (I/F) [195]	20481×8192 (64 bit)
SEL: Select (I/F) [195]	1.2×10^{19} (64 bit)
BS: Binary Search [195]	$2.0 \times 10^5 + 1.6 \times 10^7$ (64 bit)
RED*: Reduction [195]	1.05×10^9 (64 bit)
BC: Bitcount [194]	2.88×10^8 (64 bit)
MLP*: Multi-layer Perceptron (I/F) [195]	61444×8192 (64 bit)
SM: String Match [193]	5 million words, 110710 keys
LR*: Linear Regression [193]	1.5×10^9 (8 bit)
HG*: Histogram [193]	4.7×10^8 (24 bit)
PCA*: Principle Component Analysis (I/F) [193]	6.5×10^7 (32-bit)
MML*: Matrix Multiply (I/F) [193]	$2 \times 500 \times 500$ (32 bit)
KM*: Kmeans [193]	1×10^9 (32 bit) points, 20 centroids

6.7 Methodology

Workloads. We select a wide range of applications from three benchmark suites [194, 195, 193] to evaluate DRAM-BitSIMD's performance. Table 6.2 list all 13 workloads and their respective input data sets. We modify the Binary Search kernel for DRAM-BitSIMD using massively parallel brute-force matching, and replace the Euclidean distance with Manhattan distance in Kmeans to avoid computing square roots.

CPU and GPU Baselines. Our CPU baseline is a 24-core Intel Xeon operating at 2.4 GHz with 128GB 8-channel DDR4 memory, and our GPU baseline is NVIDIA Titan V.

RTL Synthesis of BSLUs. We implement five BSLU variants in RTL and synthesize them with Synopsys Design Compiler and a 14-nm SAED library. Area and power numbers per BSLU are collected from the synthesis tool. We project synthesized results to DRAM by first calculating the number of transistors by dividing the synthesized area by transistor area, where the transistor area is one-fourth of the minimum buffer area in this library (0.0666 um^2). Then we assume each transistor can be implemented on DRAM in the area similar to a 1T1C memory cell, e.g. typically $4F^2$ in the unit of Fundamental-Feature Square for scaling among technology nodes.

Area Evaluation To estimate the DRAM-BitSIMD chip area, we first use Cacti-3DD [156] to obtain the area breakdown of the DDR4 chip (Micron_8GB_x4) that is used as the building block. We adopt a DRAM sense amplifier layout described by Song et al. [196] and a patent from Micron [111] for a conventional $4F^2$ DRAM layout. As suggested in [12], the BSLU is fitted along the sense amplifier's long side. Our RLU is a 1 GHz RISC-V core operating at 9V that consumes 4.86mm^2 [197].

Energy Evaluation We assume the background power of a DRAM-BitSIMD chip is always equivalent to the peak power consumption of a DDR4 chip (worst-case assumption). We add $0.45\mu W$ for each additional activated local row buffer to account for the subarray-level parallelism, as described in [12, 42]. We calculate the dynamic power consumption of the subarray-level BSLU processing elements for each operation using

parameters from our circuit-level modeling. The overall energy consumption in a DRAM-BitSIMD integrated system also includes the power consumption of the host, estimated using the PMC-power tool [198], and the main memory, calculated by Micron's DRAM power calculator [97]. We estimate each RLU incurs 0.5W additional power.

Functional and Performance Modeling. We implement an in-house simulator for functional verification and performance modeling. Our simulator can calculate the exact number of DRAM read/write and digital logic operations for each design configuration we explore. To model the application-level speedup, we first vectorize selected benchmarks using a set of DRAM-BitSIMD API calls to emulate kernel execution and then map each API function to DRAM-BitSIMD hardware resources for optimal performance. Since DRAM-BitSIMD adopts an offloading execution pattern where the host is responsible for the resource (DRAM-BitSIMD compute units and memory) allocation, data transferring, and kernel launching, the end-to-end benchmark performance is calculated by adding the host pre-/post-processing time with the DRAM-BitSIMD kernel time. We account for the data preparation latency and energy cost by including (1) the time of data movement between the host memory region and the PIM-eligible region before and after the kernel execution and (2) the data transformation latency. The cost of input-output data movement is modeled using Ramulator [199], and the data transformation cost is modeled using parameters from SIMDRAM [34].

For modeling DRAM-BitSIMD performance, we adopt the same approach as [200, 19] by building a detailed analytical model for all DRAM-BitSIMD vector API functions that consider input characteristics (data type, vector length, etc.) and hardware characteristics (PIM parallelism, micro/macro operation complexity, etc.), and uses the bit-level simulator to drive the timing calculation, adding time to account for RLU and host operations. DRAM parameters are extracted from Ramulator [199], and the logic operation latency is extracted from our RTL circuit-level modeling. The latency and energy of PIM computing depend primarily on row accesses and the logic complexity of the high-level operation (i.e., add, sub, FP, etc.) at each bit position. We estimate the latency to latch a row of bits into the BSLU registers to be $tRCD + tRP$ ($\sim 30ns$), and the latency to write back from BSLU registers to the memory row to be $tWR + tRP$ ($\sim 30ns$). The latency for BSLU logic is conservatively clocked to match $tCCD$ ($2 \sim 5ns$). We plan to open-source all code and analytical models.

6.8 Evaluation

BSLU Area and Power. The area, dynamic power, and leakage power of each BSLU variant are shown in Table 6.3. Area results are projected to DRAM in F^2 unit.

Table 6.3: Area and Power of BSLU Variants, per 1-bit BSLU

BSLU	Area	Dynamic Power	Leakage Power
BSLU-NAND-1Reg	508 F^2	3.95 uW	4.40 nW
BSLU-MAJ-2Reg	776 F^2	6.23 uW	5.46 nW
BSLU-AP-2Reg	788 F^2	7.06 uW	6.47 nW
BSLU-BitSIMD-2Reg	924 F^2	6.13 uW	6.31 nW
BSLU-BitSIMD-3Reg	1052 F^2	7.07 uW	8.43 nW

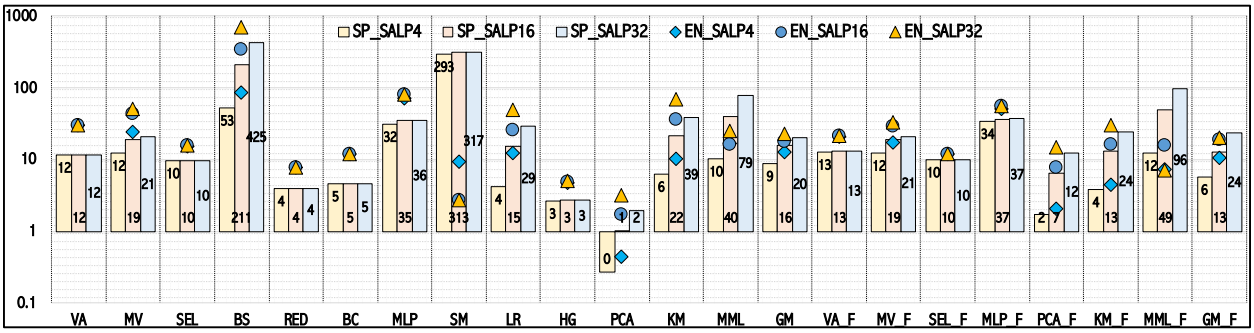


Figure 6.8: DRAM-BitSIMD-3Reg speedup and energy saving over CPU. Bars=speedup (SP); data points=energy reduction (EN).

Insights from our Design Exploration. Figure 6.8 reports the speedup (SP) and energy reduction (EN) of three DRAM-BitSIMD-3Reg versions with varying degrees of SALP against the CPU. The SALP increases the area and power overhead (See Section 5.5.4) but improves performance significantly. A 4-way/16-way/32-way SALP design incurs 3.2%/12.8%/25.7% chip area overhead. With only 3.2% area overhead, the 4-way-SALP configuration is a candidate for a **memory-first** deployment. The most aggressive design (DRAM-SALP32) outperforms the CPU baseline by 2X/425X/20X and reduces energy consumption by 3x/693x/20x (min/max/geomean) and is our best **accelerator-first** design.

We observe that some benchmarks are not sensitive to the increasing SALP level. For VA, MV, SEL, and BC, the data movement between host memory regions and PIM-eligible regions dominates the execution time (> 80%). For HG, the execution time is bounded by the rank-level data aggregation (population count or reduction sum). We leave the exploration of optimal reduction logic placement and strategy for future work. For RED, since all DRAM-BitSIMD variations share the same reduction logic at the rank level, there is no performance difference across different SALP configurations. Accelerating PCA is difficult because PCA requires all input-output vectors to be placed in the same bank due to the lack of support for massively internal data movement across banks, limiting the parallelism potential of bit-serial techniques. We also notice DRAM-BitSIMD achieves comparable speedup and energy savings for floating point vs. integer computation. Finally, energy reduction is highly correlated to the execution time.

Figure 6.8 reports the Geo-mean speedup and energy reduction over the CPU of five proposed BitSIMD designs and SIMDRAM that only implements MAJ/NOT. The results are normalized to that of SIMDRAM,

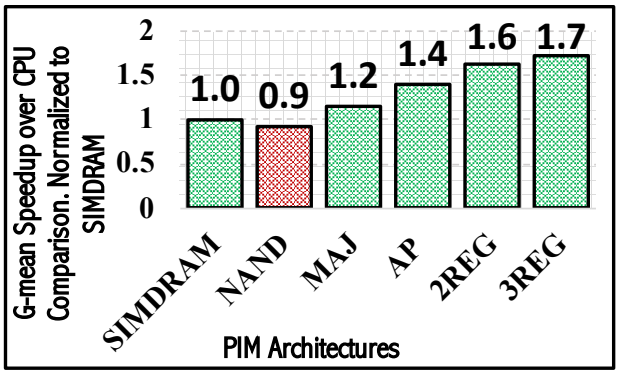


Figure 6.9: BitSIMD energy savings over SIMDRAM.

and all architectures share the same SALP level of 32. Note some of the benchmarks (indicated by '*') cannot be handled by SIMDRAM due to the lack of cross-column reduction logic and floating-point implementation. We assume an updated SIMDRAM has these features for a more fair comparison. SIMDRAM only outperforms the NAND-based DRAM-BitSIMD, showing the performance advantage of supporting a larger set of bit-serial operations (see Section 6.4.2). The energy advantage of digital DRAM-BitSIMD is even greater (Figure 6.10) because the SIMDRAM TRA, and its need for more row access per compute step, incur higher peak power and latency.

Combining Figure 6.8 with the BSLU areas in Table 6.3 also shows that the BitSIMD-2Reg and 3Reg designs are best in terms of raw performance as well as area- and energy-efficiency. The smallest BitSIMD technique that seems viable is AP and is a viable option if area overhead is the overriding concern. But 2-Reg and 3-Reg provide significantly better performance per unit area. We focus on 3-Reg because it provides 1.7x higher multiplication performance even though it is slightly worse (14% BSLU area) than 2Reg in area efficiency.

Comparison against GPU. We compare DRAM-BitSIMD designs to GPU using both the same set of 16 compute primitives (32-bit operands) from [34] (Figure 6.11 and 6.12), as well as several PrIM benchmarks [195]. For a fair comparison, we normalize DRAM-BitSIMD's performance to the GPU silicon area ($815mm^2$) and power consumption ($250W$), and we exclude data transfer in both cases.

Figure 6.11 and 6.12 show that DRAM-BitSIMD has better power efficiency than area efficiency compared to GPU. DRAM-BitSIMD provides better throughput for logical, relational, and non-quadratic arithmetic (e.g., addition/subtraction) operations than GPU but performs worse for division and multiplication, which has quadratic complexity for bit-serial implementation. This explains the performance degradation for MV and MLP benchmarks, which are multiplication-intensive, and good speedup and energy saving for VA, SEL, BS, and BC workloads, dominated by pattern matching and integer ops. RED is limited by rank-level reductions.

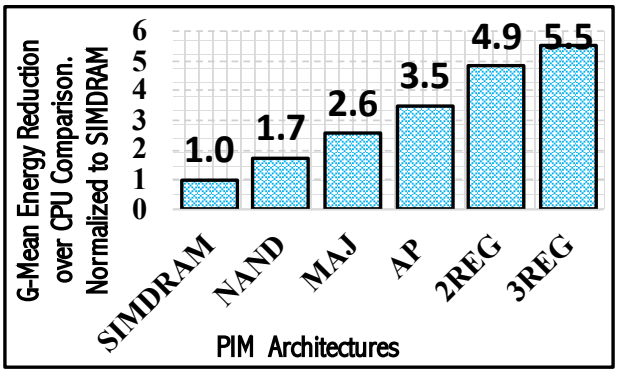


Figure 6.10: BitSIMD energy savings over SIMDRAM.

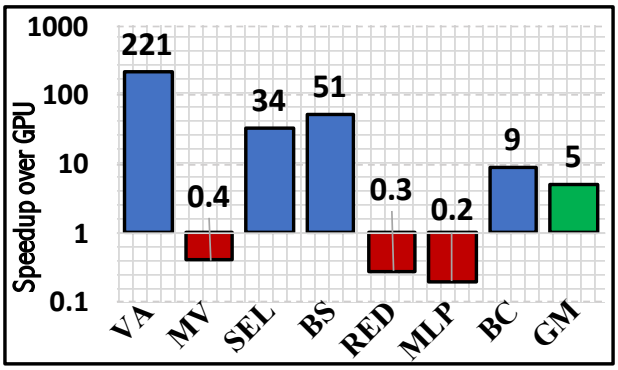


Figure 6.11: BitSIMD-3Reg speedup over GPU.

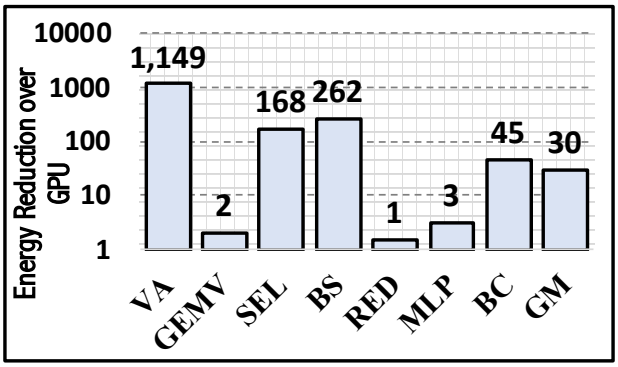


Figure 6.12: BitSIMD-3Reg energy savings over GPU.

6.9 Conclusions

This paper explores the design space for subarray-level, bit-serial PIM, including the design space for digital bit-serial logic, for both memory-first (low PIM overhead) and accelerator-first (optimized for PIM) deployment scenarios. We also introduce a rank-level unit (RLU) as a PIM controller, offloading the memory controller and orchestrating the PIM computation at the rank level, and the RLU also performs reductions and other tasks that are not strictly data-parallel. We show that our best bit-serial architecture, the 3-register

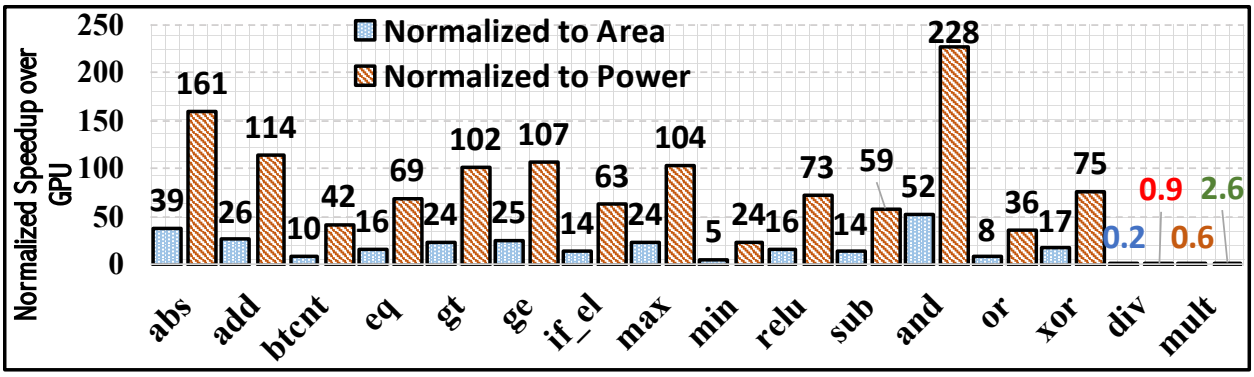


Figure 6.13: DRAM-BitSIMD 3-Reg speedup over GPU. Results are normalized to GPU silicon die area and power.

NOT/AND/OR/XOR/SEL, outperforms the CPU by 20x, GPU by 5x, and SIMDRAM by 1.7x, and is substantially more energy- and area-efficient.

Chapter 7

Ultra Efficient Acceleration for *De Novo* Genome Assembly via Near-Memory Computing

7.1 Introduction

Next Generation Sequencing (NGS) has revolutionized genomics due to the high volume and low cost of sequencing [201, 202, 203, 204]. A typical NGS system can generate 10TB of short DNA reads (100-300 base pairs) in a single run [205, 206]. For most sequencing experiments in which a high-confidence reference genome is known, the standard workflow is to align these reads against the appropriate reference genome [68, 65, 207]. However, the reference genome is not always available, especially when analyzing unknown species, such as a new virus or bacteria [208, 209, 210], or meta-genome that is sequenced from diverse environmental microbiomes [209, 208, 210]. Even when the reference genome is available (e.g., humans), the reference genome may be missing rare genomic variants of biomedical interest [211, 212, 213, 214]. In these contexts, we must assemble our reads *de novo* (without a reference genome). State-of-the-art *de novo* genome assemblers use the reads to construct a de Bruijn graph (DBG) and subsequently find all maximal non-branching paths of the DBG to produce *contigs* (contiguous segments of the assembled genome) [66, 71, 67, 68]. DBG-based assemblers are both time- and memory-intensive, due to a large amount of sequence data and the explosive number of nodes in the graph, posing significant challenges on conventional computing systems.

Although most DBG-based *de novo* assemblers [71, 69, 70, 67, 215] adopt parallel algorithms to improve performance and scalability, they are always memory-latency bound—the memory access takes up the most portion of execution time. Furthermore, the memory bandwidth requirement of DBG assemblers constantly increases at a linear rate as the degree of parallelism increases, which makes DBG assembly also memory-bandwidth bound in a parallel environment.

Accelerating DBG processing is of paramount importance for several reasons. First, DBG processing is the de facto *de novo* assembler for both large (mammalian-sized) or small (e.g. E.coli) single-cell genome analysis [214, 67, 215, 68, 70, 71], as well as metagenomic studies where a large (up to TBs) mixture of bacterial, viral, and fungal microbiome genomes obtained directly from a human body or an environment needs to be assembled [216, 217]. Second, although primarily developed to assemble the 2nd gen (a.k.a. NGS) reads, DBG processing retain its relevance as the foundation of assembling reads generated by the 3rd gen sequencers [218, 219]. Third, DBG processing is on the critical path of many time-critical genome analysis tasks. In the emerging precision medicine domain, a patient's sample is first sequenced on the NovaSeq instrument in under 48 hours, producing 6 to 12 TB microbiome and human DNA/RNA data. This raw sequence data is then passed through various stages, including the DBG assembly (∼3600 CPU hours) [82]. Finally, the rate of genomes been sequenced is vastly outstripping Moore's law [56], For example, the data volume of unassembled bio-sequences surpasses that of astronomy, particle physics, and websites such as YouTube and Twitter [220, 10].

Near-data processing (NDP) is an emerging memory-based approach that can provide scalable parallelism and memory bandwidth by integrating massive cores in memory devices [46, 221, 17, 48, 222]. In this work, we exploit NDP technology to accelerate DBG assembly. We design near-data parallel algorithms for graph construction and graph traversal that exploit the hardware parallelism by distributing data and operations in different memory locations. The near-data parallel implementation enables different NDP cores to process different portions of data simultaneously for performance scaling.

However, naive NDP implementation faces several issues caused by data communication among NDP cores. The graph construction phase requires intensive data movement among NDP cores, because the input sequence and the intermediate data structures are distributed with different strategies. Furthermore, during graph traversal, building a contig requires a series of accesses to k-mers (DNA strings of fixed length k) located in different NDP cores. Our evaluation shows that such inter-core data communication can take up to 60% to 75% of the execution. To reduce these overheads, we propose several optimization techniques, based on domain-specific knowledge on genome assembly. In the graph construction phase, we shuffle the distribution of DBG data structures based on the distribution of addresses for copy, using a greedy algorithm to reduce the number of inter-core data movements. Furthermore, we propose message buffering and k-mer

compression to reduce the size of data communicated. For graph traversal, we design a speculative contig expansion which parallelizes traversal operations in each core.

We summarize the contributions of this paper as follows:

- This is the first in-memory accelerator for DBG-based *de novo* genome assembly.
- We propose several optimization techniques based on domain-specific knowledge of DBG assembly to reduce the data communication overhead in NDP systems.
- We improve upon a state-of-the-art DBG assembly on a NDP system, and we evaluate our design using a application-level architecture simulator. We compare the performance of the proposed design with the software baseline with real genomes. The results show that the proposed optimization techniques can lead to 33-fold and 16-fold speedup over the software baseline for graph construction and graph traversal, respectively. The performance gap between our NDP-based DBG assembler and a conventional one is expected to grow even wider given larger genome size, as demonstrated in our evaluation. Furthermore, the proposed NDP-based DBG assembler scales well when increasing the system size.

7.2 Key Ideas

We propose our NDP-accelerated DBG assembler by modifying the widely used MEGAHIT tool [70] (Figure 7.1).

7.2.1 DBG Assembly Pipeline

We reuse the interface in MEGAHIT to support the NDP functionality in a general DBG assembly pipeline, which includes read loading, graph construction, contig assembly, etc. We replace the implementation of graph construction and contig assembly, which are performance bottlenecks in the pipeline, with our proposed NDP method.

MEGAHIT uses several intermediate data structures for transitions between different pipeline phases. We do not change these intermediate data structures used in MEGAHIT to keep the general pipeline intact in our implementation. Specifically, the NDP graph construction takes in the binary sequence data generated from the MEGAHIT read loading program, which supports general input formats of genome assembly, including single-end and double-end reads using different sequencing technologies [201, 63]. The NDP-based graph construction generates the sorted k-mers and writes them into files that can be processed by the graph cleaning program in MEGAHIT. Then, the NDP-based contig assembly distributes the cleaned DBG (sorted k-mers) in the NDP system and traverses the DBG to build contigs using the proposed techniques. The

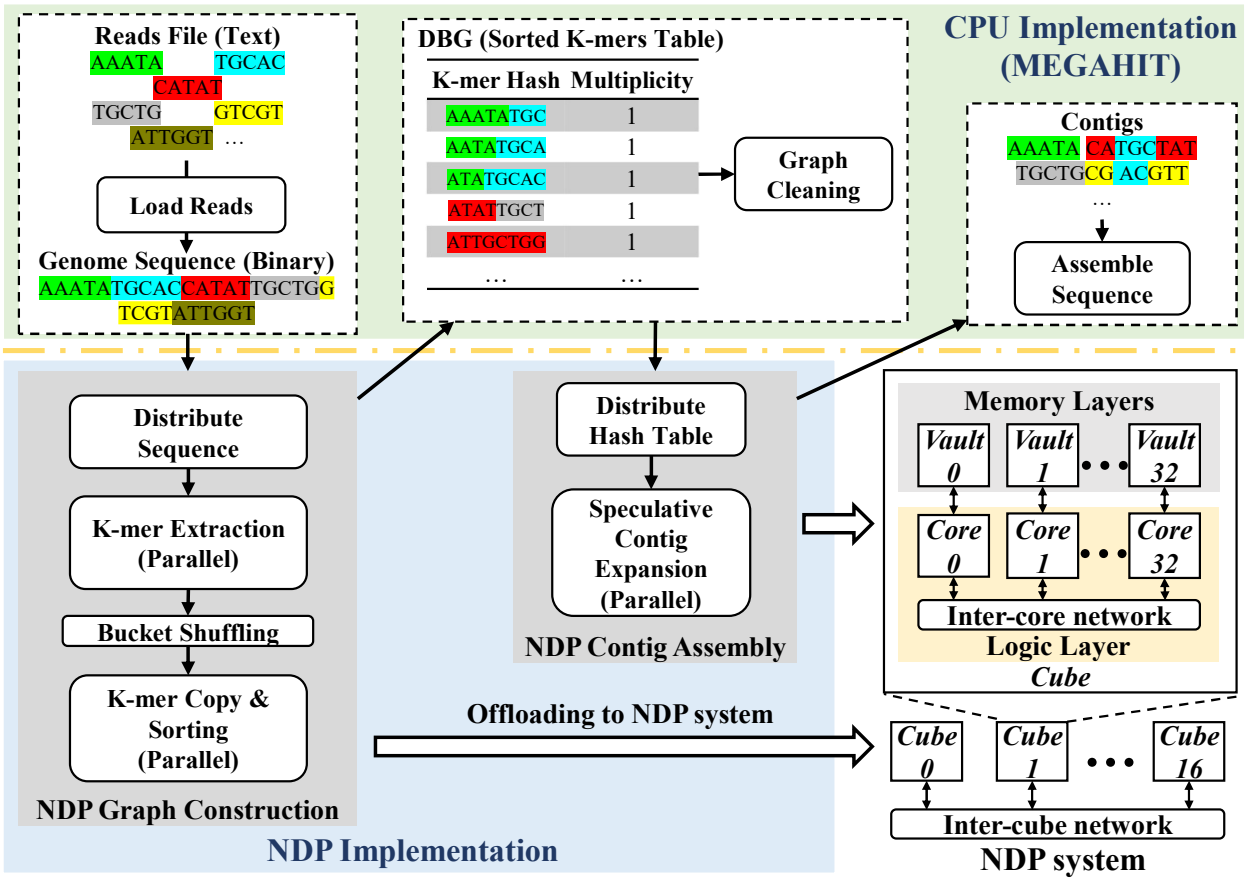


Figure 7.1: **The overview of NDP-accelerated DBG assembly.**

NDP-based contig assembly generates the contig graph that will then be assembled by the original MEGAHIT implementation for the final sequence.

7.2.2 NDP Acceleration

NDP is a type of architecture where the data processing and storage units are co-located in a single module. Emerging 3D-stacked DRAMs, such as hybrid memory cubes (HMC) and high bandwidth memory (HBM), are popular platforms to enable NDP functionality. A 3D-stacked DRAM integrates a logic layer in the memory die, which features highly parallel compute units to leverage the low access latency and large internal memory bandwidth. Take the HMC as an example. Each HMC cube has multiple vertical slices–vaults. The memory layers and the logic layer communicate through fast through-silicon vias (TSVs). There have been various HMC-based NDP systems [17, 48, 49, 50] where a small per-vault core (referred to as a NDP core) is embedded at the logic layer, re-purposing a vault to a near-memory compute unit. The NDP system can scale out by connecting multiple cubes using high-speed serial links to form a network of NDP cores. Scaling out the NDP system simultaneously increases the memory capacity, parallelism, and aggregated memory

bandwidth, which is ideal for parallel genomics workloads with a large memory footprint and high bandwidth demand. In this work, we evaluate the effectiveness of proposed designs in the context of HMC architecture, which provides concrete parameters accessible to researchers. However, our optimizations may also be applied seamlessly to other 3D-stacked memories like HBM, which shares a similar parallelism and partition strategy (e.g., channels v.s. vaults) [43, 44].

The NDP architecture consists of multiple Hybrid Memory Cubes (HMC), and each HMC connects to the others using an inter-cube network [50, 223]. Each cube's memory is divided into several vertical memory vaults, and each vault is coupled with an integrated processing core which is connected to a memory controller for local vault access. We can schedule parallel applications on NDP systems by exploiting massive NDP cores. NDP system supports remote function calls based on message passing to handle inter-core communication without expensive coherence management [17]. In this work, we propose the implementation of graph construction and graph traversal (contig assembly) on the NDP system with optimization based on domain-specific knowledge.

7.3 NDP-based DBG Construction

Figure 7.2 illustrates the flow of parallel DBG construction, and Algorithm 1 shows the pseudo code. The DBG is represented as a series of "buckets" distributed among all NDP vaults. The distributed DBG is built through the following steps: (1) Reads distribution. (2) Bucket allocation. (3) k-mer address scan. (4) k-mer extraction. (5) Post processing. We design an efficient bucket distribution procedure and message buffering and compression to improve the performance by reducing the inter-core communication.

7.3.1 NDP parallel graph construction

Input reads are first distributed to all NDP vaults. Then the NDP system sets up several buckets for cores to collaborate without interfering with each other. Building a DBG can be abstracted as putting k-mers into different buckets based on their hash values. Each bucket is divided into N non-overlapping partitions (lines 1 to 7), where N is the number of NDP cores. When an NDP core visits the bucket, it is confined to its partition, making concurrent bucket accesses among different cores possible. Then the buckets are assigned to NDP cores (line 8). The distribution of buckets is crucial to the performance of graph construction. Thus we design an optimized bucket mapping scheme, which is described in 7.3.2.

Next, a batch of buckets are selected in each iteration, and NDP cores fill those buckets with k-mer addresses by scanning its local reads (line 10 to 14). This is because decomposing reads into k-mers inflates the size of the input dataset by a factor of (n-k+1)*k/n (n = read length, k = K-mer size), processing all

```
input  : Distributed raw read data - reads[num_reads]
         → cores[num_cores].seq_from
         → cores[num_cores].seq_to
input  : num_bucket
output : de Bruijn graph table - dbg[num_kmers]
/* Calculate the size and partition for each bucket                              */
#ndp_parallel_for
for c ← 1 to num_cores do
    for seq ← cores[c].seq_from to cores[c].seq_to do
        for kmer : seq do
            b = hash(kmer)%num_buckets;
            cores[c].bucket_size[b] + +;
            buckets[b].size + +;
        end
    end
end
/* Distributed buckets to NDP cores                                             */
assign_buckets(buckets, cores);
/* Copy k-mer addresses into buckets                                            */
#ndp_parallel_for
for c ← 1 to num_cores do
    for seq ← cores[c].seq_from to cores[c].seq_to do
        for kmer : seq do
            b = hash(kmer)%num_buckets;
            buckets[b].addresses.add(&kmer);
        end
    end
end
/* Copy k-mers from address                                                     */
#ndp_parallel_for
for c ← 1 to num_cores do
    for bucket : cores[c].buckets do
        for kmer_addr : bucket.addresses do
            target_core = find_core(kmer_addr);
            target_core.copy(kmer_addr, bucket.kmers);
        end
    end
end
/* Bucket post-processing:  sorting, remove redundancy, calculate multiplicity, etc. */
#ndp_parallel_for
for c ← 1 to num_cores do
    for bucket : cores[c].buckets do
        post_process(bucket);
        dbg.add(bucket);
    end
end
```

Algorithm 1: Pseudo code for building distributed De Bruijn Graph on a NDP system.

buckets simultaneously results in peak memory explosion. After addresses are filled for all buckets, each NDP core takes the ownership of its buckets by copying k-mers based on the addresses gathered from the previous step (lines 15 to 20). The two-pass creation of the buckets for DBG construction is superior than pushing the k-mers directly into the buckets for several reasons: the algorithm iteratively selects a batch of buckets to process, cores may have unbalanced amounts of k-mers belonging to the current buckets. If a single-pass paradigm is adopted, some cores will be busy "pushing" K-mers into the network to the destination buckets while other cores are idle. Furthermore, the "pushing" has a sequential-reads/random-writes pattern, incurring low cache locality. Since K-mer addresses (8-byte) are smaller than the actual K-mers (32-byte to 128-byte), "pushing" addresses incurs a smaller penalty. In the second-pass, cores fill their buckets with K-mers, and since buckets are roughly the same size in each batch, cores have balanced workloads. The second pass has a sequential-reads/sequential-writes pattern.

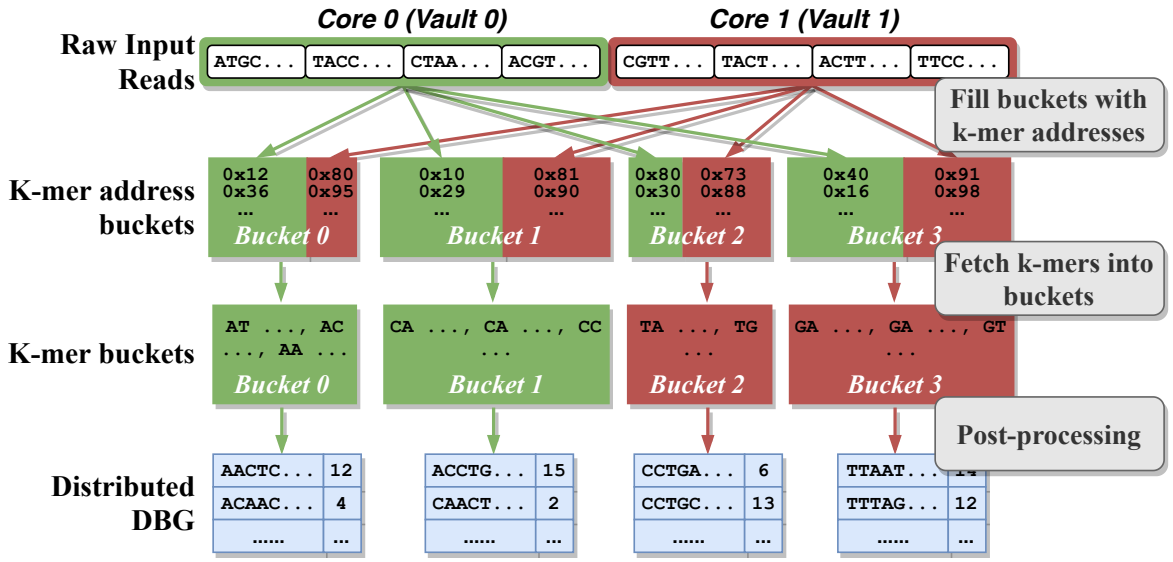


Figure 7.2: **The overview of NDP-based DBG construction.**

The k-mer may be stored in a remote core that requires a remote function call to copy the data. Since we evenly distribute sequences to NDP cores in the original order, we can easily locate the target to send the remote function. This step suffers from massive fine-grained communication overhead since many k-mers are from reads distributed in remote vaults.

Finally, a post-processing stage (line 21 to 25) is involved to reduce bucket size, since many k-mers (as many as 80% [224]) are redundant due to the deep sequencing coverage and repeat patterns in genomes [224]. A common practice is to sort the k-mers in a bucket, allowing us to obtain the multiplicity (number of occurrences) of each k-mer as a helpful by-product.

7.3.2 Bucket Distribution

To design a good bucket distribution scheme, one needs to consider the origins of k-mers in a bucket. Figure 7.3 shows an example of two buckets and three NDP cores (vaults). A large portion of read partition 0 (red) is hashed into bucket 0; thus, co-locating bucket 0 with read partition 0 can significantly reduce the number of remote k-mer fetch requests. Similarly, bucket 1 has a high concentration of k-mers from the read partition 1 (green), so it is more suitable to be assigned to the vault 1. There is anywhere between 29% to 40% reduction of messages over a naive random bucket mapping if the origins of k-mers are considered. One possible explanation for such a phenomenon is that real genomes often contain many regions of repeat patterns. For example, about 8% of the human genome consists of so-called tandem repeats, which are low complexity short sequences that occur multiple times in a row (e.g. "CAGCAGCAG...") [225]. The

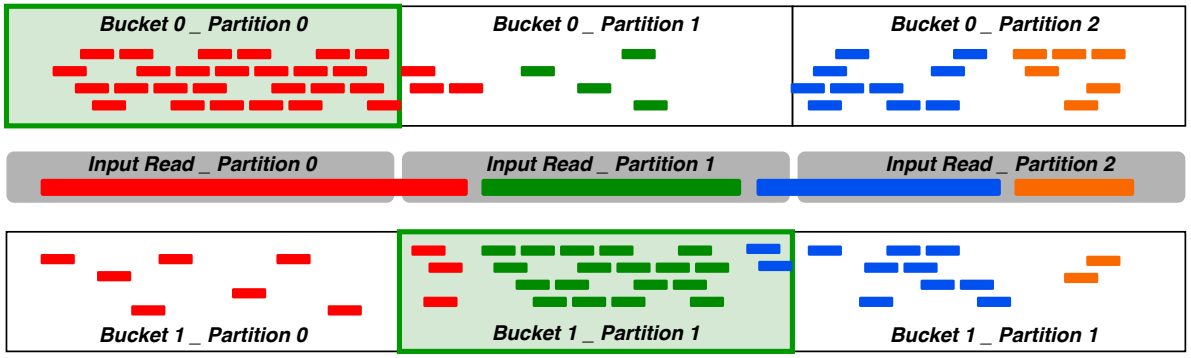
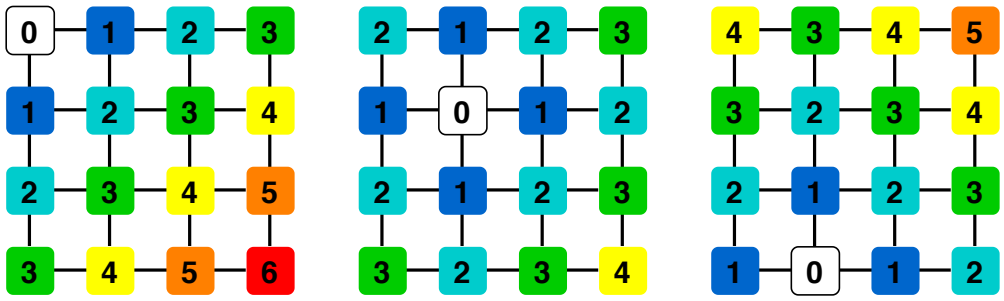


Figure 7.3: **Bucket shuffle based on the origins of k-mers.**



a. **Bucket at vault 0** b. **Bucket at vault 5** c. **Bucket at vault 13**

Figure 7.4: **Hop distance from the source core (in white) to different remote cores (in color).**

commonly adopted hashing schemes that operate on the binary form of a k-mer pattern will inevitably try to fit k-mers obtained from these repeats into a small group of buckets.

However, simply reducing the number of messages passed among the NDP cores may not be the optimal solution, as it fails to consider the non-uniform latency of switching a packet in some networks. For example, in a mesh-style network, the latency of switching a packet is correlated to the distance between two nodes, since a packet arrives at its destination through a series of hops, and each hop adds a certain amount of additional latency. Figure 7.4 illustrate a situation where a message-reduction-based bucket shuffling strategy does not work well. Suppose buckets have roughly equal amounts of k-mers that need to be fetched from each remote vault in an NDP system with a mesh NOC. The total amount of remote messages generated is the same regardless of the bucket location. However, when the hop count per message is considered, it is a poor choice to put this bucket at the four corner vaults. For example, if each remote vault contributes 10 k-mers into the bucket, then a bucket generates $10\times1\times2+10\times2\times3+10\times3\times4+10\times4\times3+10\times5\times2+10\times6\times1=480$ total message hops at vault 0, 320 at vault 5, and 400 at vault 13. Therefore, total message hops should be considered if we strive to reduce inter-core communication costs.

The slowest core limits the run time of the parallel graph construction, and the inter-core communication takes the majority of the execution time. Thus the optimal bucket mapping is the one that generates the

least communication for the slowest core. For an NDP core, the communication cost of processing its share of buckets can be approximated as the total message hops needed to fetch all k-mers. However, finding such optimal bucket mapping is infeasible. Let's consider a simpler case where the number of buckets mapped to each core is the same. With 65536 buckets and 512 NDP cores (vaults), the number of possible mappings that need to be checked is $\binom{65536}{128} \times \binom{65408}{128} \times \binom{65280}{128} \times ... \times \binom{128}{128}$. A naive heuristic that selects the least amount of communication cost for each bucket can easily suffer from workload imbalance. We describe below a greedy solution that addresses both the run time concern and the imbalance concern.

After each bucket's size is obtained (line 11), all buckets are ranked in descending order based on their sizes and put into a list. The bucket distribution logic runs in a loop in which each iteration selects a batch of n buckets from the list, with n being the number of NDP cores. For an NDP system with fully connected networks, each bucket is assigned to a vault based on its largest partition. A bucket will be randomly selected if two or more partitions have the same size. The vault that has been assigned with a bucket in this iteration will not be assigned with another one. When a bucket needs to be assigned to an occupied vault, the bucket is assigned to a vault with the second-highest k-mer contribution (second highest partition). This process repeats until all buckets are assigned. For an NDP system without a fully connected network topology, the bucket shuffling step is the same as the above procedure with minor tweaks. Instead of choosing a winning vault for each bucket based on its partition sizes, the bucket is assigned to the vault that generates the smallest hop count. This shuffling implementation adds an insignificant amount of overhead (<1%) and works well in our evaluation.

Each NDP core is provisioned with a table that indexes buckets to their owner vaults/cores. The number of table entries is equal to the number of buckets, which is 65536 in our evaluation. Each entry has $\log_2 65536 = 16$ bits to represent bucket IDs, and additional bits to represent core IDs (9 bits for 512 cores). The table adds 1.2% total storage overhead per HMC cube. Searching this table is a constant time operation since the bucket index is the hash value of a k-mer.

7.3.3 Message Buffering and *k*-mer Compression

Message Buffering

In the graph construction step, each NDP core copies a k-mer from a remote vault by sending that vault's owner (an NDP core) an extraction request wrapped in a message. The remote NDP core responds to the request immediately by sending the k-mer back in another message. This is inefficient since each message's payload can fit multiple k-mers, and each request has no dependency on each other. An obvious optimization

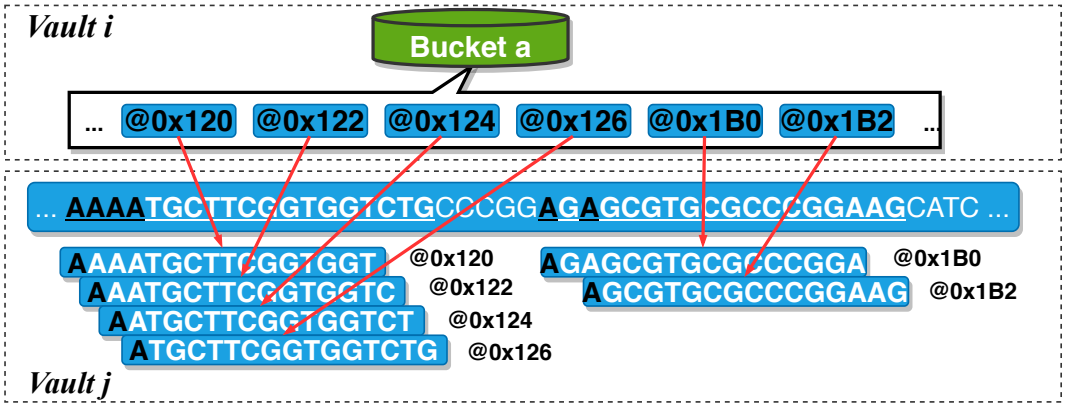


Figure 7.5: **Message compression by leveraging the overlapping bases of consecutive *k*-mers.**

is to delay the responses and aggregate multiple k-mers into one message. At each vault, we provide $n-1$ buffers corresponding to the rest $n-1$ remote cores.

k-mer compression

This compression technique is used in conjunction with the message buffering to improve a message payload density. The key observation is that since the k-mer addresses are put into a bucket by sequentially sliding a window on input reads (with variable stride lengths), there is an opportunity for data reuse when copying k-mers pointed by those addresses. Figure 7.5 illustrates this idea. A naive way of sending k-mers from Vault j to Vault i is to lay them out exactly in the message payload one by one. Suppose the message payload size is 64 bit and 2-bit/base. The uncompressed format allows two k-mers to be sent through one message. A more compact representation of those k-mers is to copy the entire sequence from the first base of k-mer at 0x120 to the last base of k-mer at 0x126 (19 bases) and provide a small array of offset pointers to distinguish each k-mer. This allows the same message payload to fit four k-mers.

The compressibility of k-mers in a packet depends on several variables: the number of buckets, size of k, hash function, genome repeat patterns, etc. Deriving a formula to predict the effectiveness of packet compression accurately is out of this project's scope. We empirically evaluated this idea using an E.coli DNA sequence and realistic DBG assembler settings: k=22, 65536 buckets, and the first four bytes of each k-mer are hashed. We find that over 20% of consecutive k-mer pairs in a packet are overlapped, and the proposed compression technique trims away more than 10% of redundant bases. We also analyze how many bases every overlapping k-mer pair shares. The distribution is summarized in Figure 8. The result suggests that each pair of overlapping k-mers have a high chance of sharing more than half of their content.

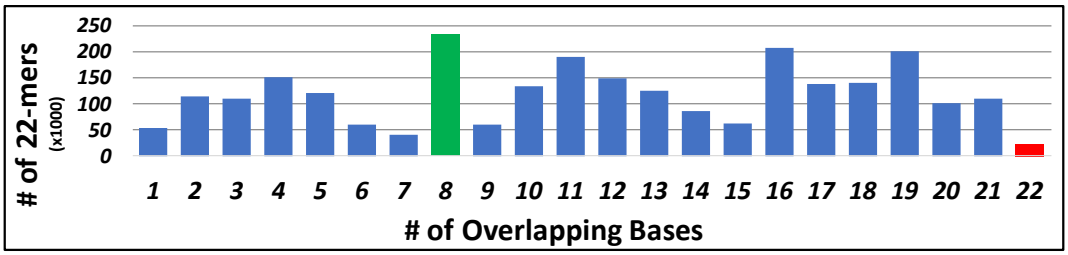


Figure 7.6: **The distribution of the number of bases that are overlapped for each consecutive 22-mer pairs.**

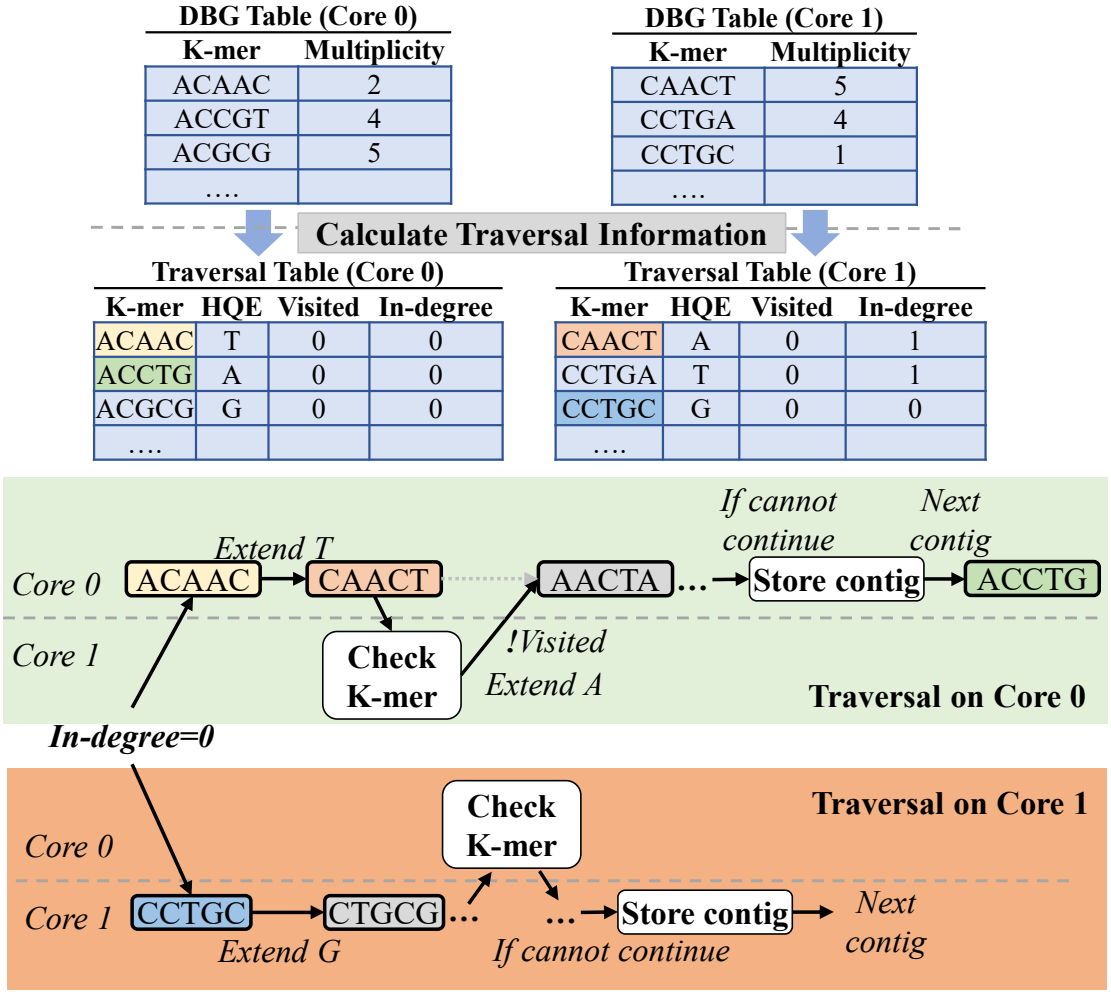


Figure 7.7: **The overview of NDP-based graph traversal.**

7.4 NDP-based DBG Traversal

This section introduces the NDP-based graph traversal. We exploit the NDP system's parallelism to construct contigs and use a speculation mechanism to accelerate contig expansion.

```
input  : Distributed DBG table - dbg[num_kmers]
         → cores[num_cores].kmer_from
         → cores[num_cores.kmer_to
output : Contigs built by traversal - contigs
/* Calculate k-mer information                                              */
#ndp_parallel_for
for c ← 1 to num_cores do
    for kmer ← cores[c].kmer_from to cores[c].kmer_to do
        /* Find the high-quality extension (HQE)                            */
        for c: ['A','T','G','C'] do
            kmer.HQE = max_multiplicity(kmer[1 :] + c);
        end
        /* Update the in-degree of HQE                                      */
        target_core = find_core(kmer.HQE);
        target_core.increament(kmer.HQE.in_degree);
    end
end
/* Parallel contig assembly                                                 */
#ndp_parallel_for
for c ← 1 to num_cores do
    for kmer ← cores[c].kmer_from to cores[c].kmer_to do
        if kmer.in_degree == 0 then
            contig = kmer;                                   // Initiate a contig
            target_core = find_core(kmer.HQE);
            while !target_core.get(kmer.HQE.visited) do
                contig.expand(kmer.HQE);
                kmer = kmer.HQE;
                target_core = find_core(kmer.HQE);
            end
            contigs.add(contig);
        end
    end
end
```

Algorithm 2: Pseudo code for NDP-based graph traversal (contig assembly).

7.4.1 NDP Parallel Graph Traversal

Figure 7.7 shows the high-level flow for NDP-based graph traversal, and Algorithm 2 shows the pseudo-code.

Data Initialization

The input of graph traversal is the DBG data (hash table) generated in the graph construction phase. The hashing is supported in the general-purpose NDP cores. We use a leveled hashing scheme to resolve conflicts. We distribute the DBG (hash table) over different NDP cores. The DBG is divided into buckets, each of which is stored in a core.

Information Calculation

To efficiently construct contigs, we need to calculate k-mer information used during the traversal. Such information includes the high- quality extensions (HQE) and in-degree of each k-mer. High-quality extension (HQE) is the most likely extension for each k-mer. DBG assemblers use HQE to remove forward k-mers, which are introduced by read errors [224]. We point out that the graph traversal step uses HQEs to generate contigs (long sequences without branches), instead of the whole sequence. If a K-mer has multiple HQEs, the assembler stops extending the current contig because the K-mer may be a branch. The branches caused by repeated DNA patterns will be considered after the traversal phase to assembly the full DNA sequence. Furthermore,

in-degree is used to filter the start k-mer for each contig to remove redundant traversal. Specifically, the DBG assembler only constructs a contig from a k-mer with no in-coming edges. This method avoids assembling the same contig by different cores.

Each NDP core processes the information for its allocated k-mer independently (line 1 - 7). Each core sequentially processes k-mers and checks all 4 possible bases that can extend the k-mer (line 4 - 5). The HQE of each k-mer is determined by the base that leads a k-mer with the highest multiplicity. Then, the core checks the pre-loaded bucket table to locate the core that handles the HQE k-mer (line 6), and increases the in-degree of the HQE k-mer in the target core.

Parallel Contig Construction

The next step is to assemble contigs by graph traversal, where each NDP core constructs contigs independently by selecting a local k-mer as the first segment of a contig (line 8 - 17). As mentioned previously, each NDP core only selects k-mers without in-coming edges and expands the contig in one direction to avoid redundant work (line 11). To extend a contig, the source core, which is the core constructing the contig, checks the availability of the HQE of the current k-mer in the target core. If the k-mer is stored in the local vault, the source core searches its DBG table. Otherwise, the source core uses a remote function call on the target core to check the availability of HQE.

The result of k-mer expansion depends on two facts: 1) whether the k-mer exists, and 2) whether another contig has already included the k-mer. If the k-mer exists, the target core checks the "visited" tag of the k-mer to determine whether the k-mer has been used or not. If the source core receives a response from the target core that the k-mer is available for the extension, the source core uses the HQE to extend the current contig. Otherwise, the source core adds the current contig to the result (*contigs*) and selects another local k-mer as the seed for the next contig construction.

7.4.2 Speculative Contig Expansion

The graph traversal phase also suffers from inefficient inter-core communications, especially during the contig expansion where the source NDP core needs to send the query to a remote core and wait for the remote core responses to search the requested k-mer in the k-mer table. All these operations, including bi-directional communication and the search, are in the critical path of the contig expansion. Based on our experiments, the contig expansion would spend 70% of its time on inter-core communication. Therefore, it is important to reduce this time to achieve the full potential of NDP systems.

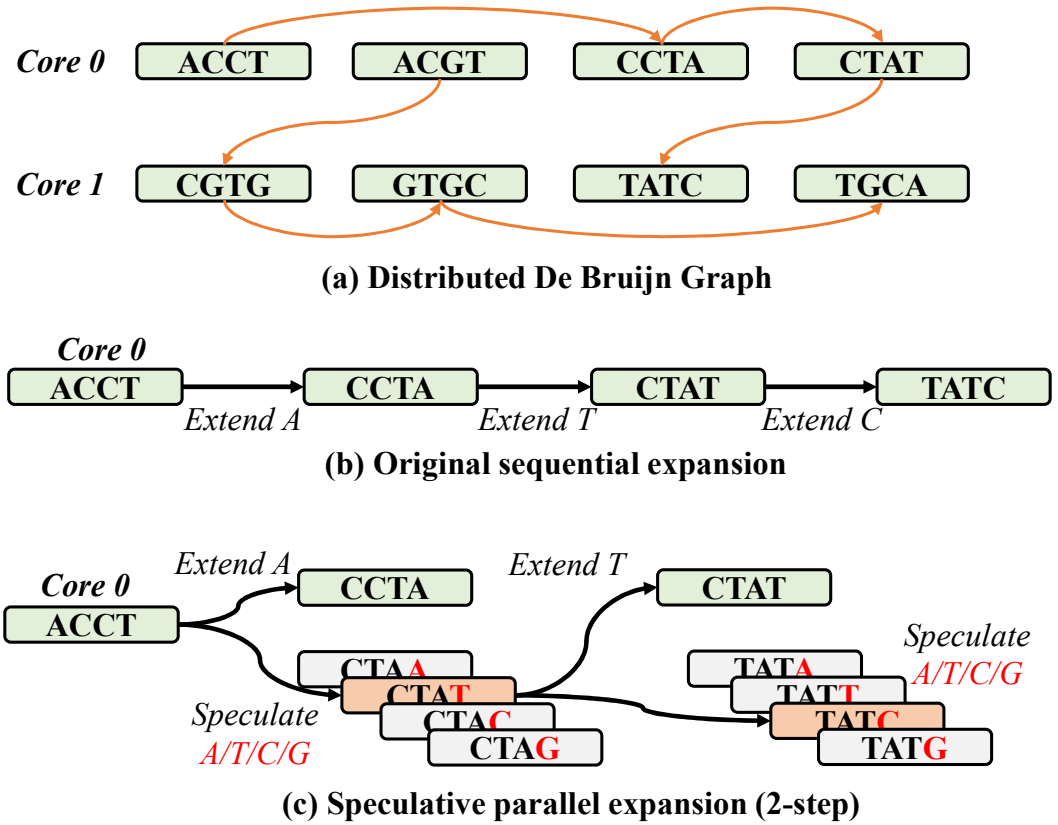


Figure 7.8: **The speculative search optimization.**

Optimization Overview

We propose a speculative contig expansion shown in Figure 7.8. In the speculative contig expansion, each NDP core searches multiple steps ahead, instead of only the HQE. The speculation's insight is to hide the latency of k-mer query by parallelizing subsequent operations.

Unlike the current contig that has the information of HQE, we do not know what will be in future steps for a query if we successfully extend the current contig with the HQE. The NDP core needs to search for all possible k-mers in the speculative steps to guarantee the speculative contig expansion's correctness. The number of possible k-mers is 4^{n-1}, where n is the number of speculative steps. During speculative search, an NDP core calculates hash values and sends search requests to target cores for all possible k-mers in the next n steps.

Operation Combining

An $n-step$ speculative search can achieve up to $O(n)\times$ performance improvement over the default one-step expansion. However, the speculation would introduce significant overhead without any optimization because of more data communications and operations for searching all possible k-mers. The number of messages that

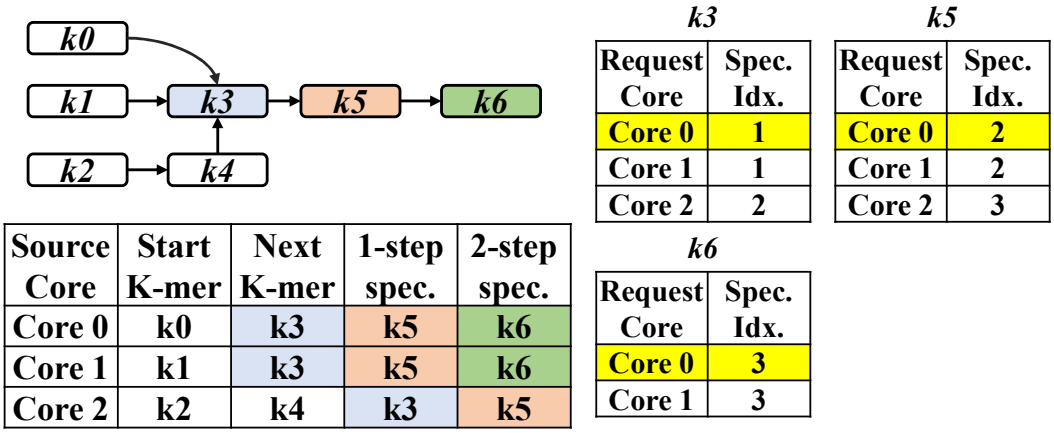


Figure 7.9: **Resolving speculation conflicts.**

are generated could grow exponentially while the performance improvement is always linear as we increase the speculation depth. Thanks to DNA sequences' nature, we can significantly reduce the speculation overhead by combining speculation for similar k-mers into a single move. All possibles k-mers in a speculation step share the most significant bases. Therefore, these k-mers are stored in a contiguous memory location in the sorted k-mers table (bucket). We may only need to send one message for all possible k-mers in a speculative step since the same target core handles these k-mers. The target core can quickly access continuous memory addresses by utilizing the data cache in the core. For example, in Figure 7.8(c), Core 0 may store all four 1-step speculative k-mers ($CTA\{A,T,C,G\}$), and Core 1 may store all sixteen 2-step speculative k-mers ($TA\{A,T,C,G\}\{A,T,C,G\}$) based on the range of hash table. In this case, a two-step speculation only requires 2 messages (1 per core). It is possible that k-mers in a speculation step are stored across cores, requiring multiple messages. In our experiments, we only observe a trivial amount of speculations (up to 5-step) requiring multiple messages in a single step because of the large data size.

Conflict Resolution

Another issue with speculative contig expansion is that we need a more complex mechanism to resolve the conflicts between cores that simultaneously access specific k-mers. Once an NDP core receives results of all possible k-mers in the next n steps, it tries to extend its current contig by checking the HQEs and corresponding "visited" tags of k-mers sequentially. It needs to send messages to cores handling the extended k- mers to avoid redundant work (setting the "visited" tags). However, without an efficient mechanism, the overhead of synchronization can eliminate the benefit of speculation.

To efficiently resolve conflicts, we propose a lightweight mechanism, nearest source assignment, to solve the conflict in the target core. Specifically, each source core extends all speculative k-mers locally as further as

Table 7.1: Programming Interface

Operation	**Remote Function Call**
Copy k-mer	get(id, A func, A addr, A ret, S ret_size)
Set Data	put(id, A func, A addr, S size)
Request k-mer	put(id, A func, A addr, S size)
Get Buffered k-mers	get(id, A func, A ret, S ret_size)
Search k-mer	get(id, A func, V hash, A ret, S ret_size)

possible and then sends the confirmation messages to all target cores to notify the success of k-mer extension. The source code also sends a speculation index, which is the position of the k-mer in the speculation path, along with each message. The target core receives confirmation messages from different source cores on the same k-mer. It picks the source core, which sends the smallest speculation index in the message as the core to use the k-mer for expansion. If multiple cores send the same speculation index, the target core picks the core with the smallest core index to break the equality. This mechanism can effectively resolve the conflicts because different contigs will follow the same path when they conflict on the same k-mer. Therefore, the nearest source assignment can avoid potential deadlocks in a continuous k-mer path.

7.5 Architecture

This section discusses the implementation of software and hardware to support the proposed ideas.

7.5.1 Programming Interface

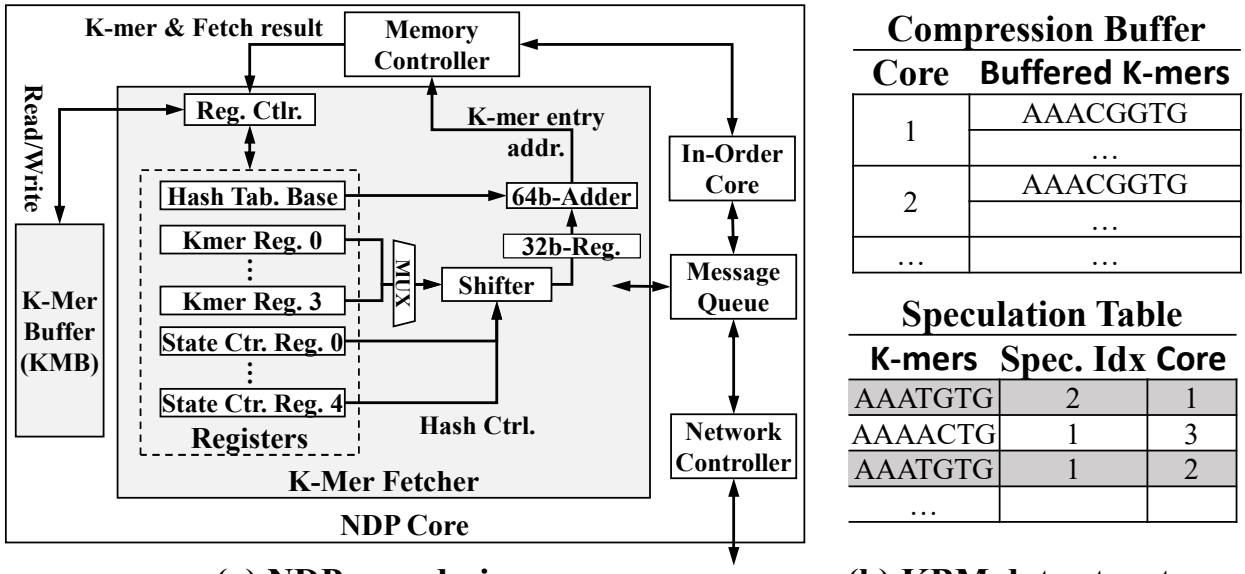
We utilize the message passing and remote function call in Tesseract [17] as the programming interface for its versatile programming interface and lightweight hardware support for message-passing (i.e., message queue). Table 7.1 lists implementations of key operations required in NDP-based DBG assembly. We use the blocking (get) or non-blocking (put) remote function call to implement different operations, where the remote function call is based on a message passing mechanism. Specifically, copying a k-mer from a remote call requires a blocking remote function call (*get*), where the parameters require the target core, the address of target k-mer, the address of return value, and the size of return value. We use A, S, and V to represent the address type, the size type, and the value type respectively. However, the blocking *get* function cannot support our proposed buffering and compression mechanism. Therefore, we propose a request operation that uses the non-blocking *put* to notify a target call to store the target k-mer in the message buffer. During the execution, each core calls the request function for each k-mer while maintaining a counter for the number of messages that have been requested for remote cores. When the counter is equal to the buffer size (introduced in Section 7.5.2), the core calls a *get* function to the target core to get all buffered k-mers. The target core

compresses the buffered k-mers and sends the results back to the source core. After the blocking *get* function call, the source core resets the counter for a specific target core and continue execution.

To enable programmers to implement NDP-based DBG assemblers, we need a framework that combines the parallel computing and the proposed programming interface based on message passing. Since we have no access to the real NDP hardware, we use a simulation-based method to emulate the proposed NDP assembler. In our implementation, we simulate OpenMP-based programs in Sniper simulator [226], which uses Pin-tool [227] as the front-end to generate simulation statistics for multi-core architectures. We insert specialized APIs using Sniper's magic instruction in the OpenMP program so that Sniper can recognize the message-passing based NDP operations. We implement the simulation logic for different message-passing functions using Sniper's synchronous and asynchronous timing models to generate the final simulation results for NDP architectures. Future work can follow a similar scheme to realize the proposed assembler in a real NDP hardware. For example, the framework can extend the syntax of widely used parallel programming APIs (e.g., OpenMP) to include function calls of message-passing, and implement a specialized runtime to schedule operations on NDP cores.

7.5.2 Hardware Support

We propose several lightweight hardware components inside each core in our NDP architecture to support the NDP functionality. Similar to Tesseract [17], each core uses a message queue and a network controller to process remote function calls based on message passing. In addition, we add two lightweight hardware components, a k-mer fetcher (KMF) and a k-mer buffer (KMB) to support the proposed optimizations. Figure 7.10(a) shows the architecture of the proposed NDP core. Specifically, the k-mer fetcher (KMF) is the unit which we can offload operations for the proposed optimizations from the NDP core. KMF can decode the potential memory addresses of k-mers based on the hash value, and generate memory commands directly to the memory controller. The KMF contains several 64-bit hardware registers to store the working information during the k-mer fetching, including the base address of the hash table (1 register), the k-mer data (4 registers), and state control information (4 registers). To generate the address for k-mer fetching, KMF first loads the base address of the hash table from the in-order core. Then, KMF uses a shifter to generate the offset of a k-mer by concatenating different bits of k-mer registers. The offset is stored in a 32-bit register, which is added with the base address in a 64-bit adder. KMF then sends the generated address to the memory controller and receives k-mer data in the k-mer registers for future operations (e.g., writing to the k-mer buffer. The k-mer buffer (KMB) stores k-mer related data which is configured to different formats for graph construction and graph traversal, as shown in Figure 7.10(b). During the graph

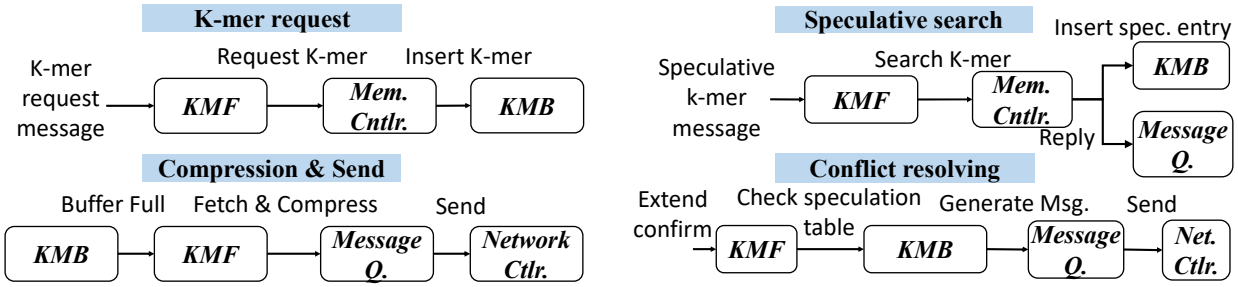


(a) NDP core design (b) KBM data structures

Component	Parameters
Shifter	2 cycles/load&shift
Adder	64-bit 4-pipelined, 4 cycles/add
K-mer buffer	64KB SRAM, 64B/block, 1 cycle/access

(c) Parameters of customized hardware components

Figure 7.10: **Hardware support for NDP-based DBG assembly.**



(a) Operations offloading during *graph construction*. (b) Operations offloading during *graph traversal*.

Figure 7.11: **Operations in hardware components.**

construction, in order to support the k-mer compression, KMB acts as a compression buffer which stores the requested k-mers grouped by the requester core. During the graph traversal, KMB is organized as a speculation table which stores the searched k-mer, requester core, and the speculation index for conflict resolving. Figure 7.10(c) shows key parameters of the proposed hardware components used in our evaluation. We implement the components of KMF using Verilog HDL and synthesize the design on Synopsys Design

Table 7.2: Workstation and NDP Configuration

CPU Model	Intel(R) Xeon(R) E5-2658 v4
Core/ Thread/ Frequency	14/ 28/ 2.30 - 2.80 (GHz)
L1/L2/L3 Cache	32 (KB) / 256 (KB) / 35 (MB)
Main Memory	DDR4-2400 MHz, 54GB/s
Memory Organization	32GB / 2 Channels / 2 Ranks
HMC 2.0 Organization	8 DRAM layers, 8Gb/layer, 8GB/cube, 32 vaults, internal (external) bandwidth: 512GB/s (480GB/s)
NDP cores	1 GHz, single-issue, in-order, 32 KB I$ and D$, LRU, 80 mW, 0.51 mm^2
HMC Memory	tCK = 1.6 ns, tRAS = 22.4 ns, tRCD = 11.2 ns, tCAS = 11.2 ns, tWR = 14.4 ns, tRP = 11.2 ns
NOC Configuration	Crossbar network, 64 KB/message payload
Inter-cube Network	2 cycles/hop, 64 bits/cycle, 2D-Mesh (default) / DragonFly / Fully-connected

Compiler. The synthesized design is placed and routed using Synopsys IC Compiler. The KMB parameters are estimated using the analytical tool CACTI-3DD [228] on 22nm technology node.

Figure 7.11(a) and Figure 7.11(b) show the operations offloaded from the NDP core to the lightweight components during graph construction and graph traversal phases. During graph construction, KMF of each core handles k-mer requests to the local k-mer from other cores. When receiving a k-mer request message (with the address), KMF generates memory commands to the memory controller which will fetch the k-mer. KMF then stores the k-mer to the corresponding entry of compression buffer (stored in KMB). If all entries of the requester core is full in the compression buffer, KMF compresses all k-mers requested by the requester core and generate a compression message. During graph traversal, KMF handles speculative search requests from other cores. Since the speculative search may request a non-existing k-mer, KMF generates memory commands for search operation in the local hash table based on the k-mer's hash value. If the k-mer exists, KMF inserts the k-mer with the requester information (e.g., speculation ID) in the speculation table. No matter whether the k-mer exists or not, KMF sends a message to the requester core about the search result. If a requester core wants to confirm the extension with a speculation k-mer, KMF fetches all entries about the requested k-mer in the speculation table and resolves the conflict. As compared to the pure-software implementation, hardware-assisted optimizations reduce data movements between the memory and the in-order core. Furthermore, the added hardware components can directly communicate with the memory controller and message interface to reduce the latency of optimization in the critical path.

7.6 Methodology

7.6.1 Simulation

We emulate the execution of our NDP-based DBG assembler using multi-threading supported by OpenMP [229]. Specifically, we create a thread for each NDP core and manually assign different tasks and data structures to threads. The proposed NDP system, including all DRAM vaults and NDP cores, is modeled in Sniper [226]

Table 7.3: Genome Datasets

Genome Name	Size
Escherichia coli O157 (E-Coli)	5,528,445 bp
Homo sapiens chromosome 3 (Human)	198,295,559 bp
Ananas comosus cultivar (pineapple)	24,880,688 bp

according to parameters reported in Table 7.2. The parameters are gathered from previously published work [17, 48, 50, 49], data-sheet for commercial products [46], and simulation in Cacti [156] and McPAT [156]. We use a Pin-tool [227] front-end to tag NDP data structures' addresses in the simulation. Therefore, Sniper can recognize and operate on these NDP data structures using NDP-specific models of remote function call based on message passing. We use Ramulator [199] to model the memory behaviors since Sniper lacks a detailed memory model. We use Cacti [156] to simulate the performance and power of customized buffers at 32nm technology. Each NDP core takes $0.51mm^2$ chip area and 32 NDP cores only consume 7.2% of the chip area available in the HMC logic layer ($226mm^2$ [17]).

7.6.2 Baseline System

The baseline performance is measured from MEGAHIT [70] running on a workstation configured in Table 7.2. We should note that CPUs are the predominant platform for DBG assembly, instead of GPUs, and MEGAHIT is one of the fastest implementations [70, 67, 215, 69] that is capable of assembling a large genome in parallel. We do not compare to GPU, since all of the GPU-based DBG assemblers we find are deprecated [230, 231, 70] due to lack of support and performance. In fact, we are informed by the authors of MEGAHIT that the GPU-implemented MEGAHIT is slower and harder to use than its CPU counterpart.

7.6.3 Workloads

We test DNA sequences from three species downloaded from GenBank [232] as shown in Table 7.3. We use a next-generation sequencing read simulator [233] to generate NGS reads using Illumina technology [202, 203, 204]. We set the fold of coverage, length of reads, and mean size of DNA fragments to 20, 150, and 200 to generate sufficient simulation data.

7.7 Results

Figure 7.12 shows the results of comparing the 16-cube NDP system and the CPU baseline. We compare the performance of the optimized NDP implementation (`Opt-HW`) to the `CPU baseline` and the NDP implementation without optimizations (`Original NDP`). `Opt-SW` represents software-implemented optimizations All

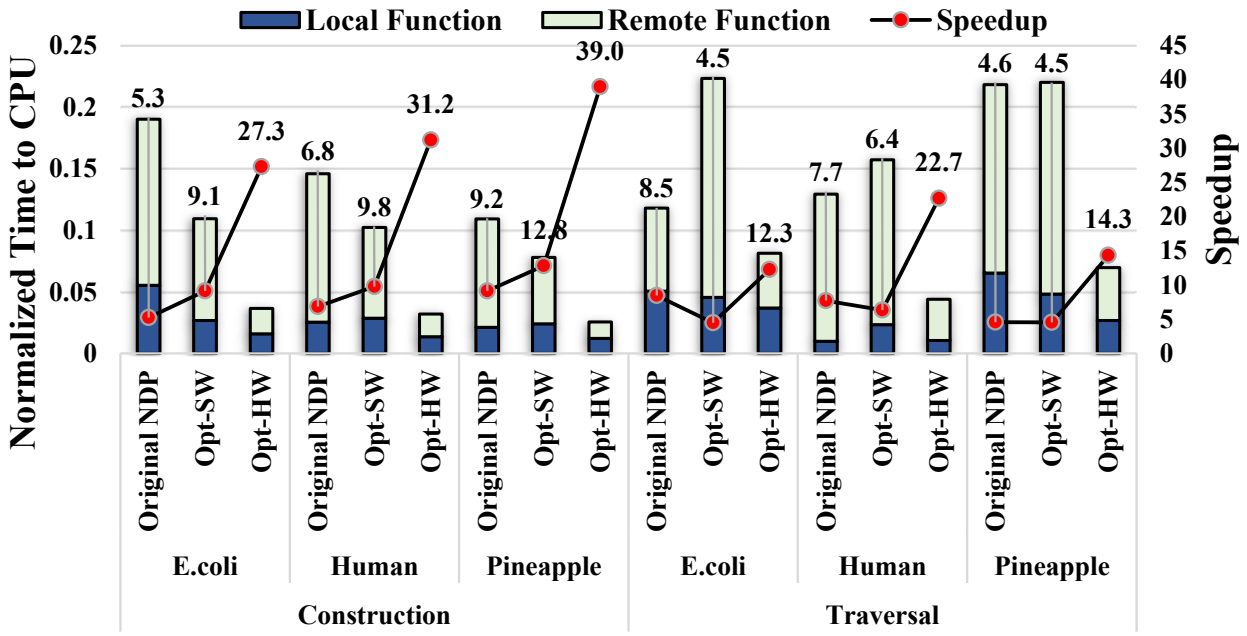


Figure 7.12: **Performance comparison with the baseline on graph construction and graph traversal.**

results are normalized to the `CPU baseline` and we break the total execution time into the time on local functions and remote functions.

Comparison to CPU and original NDP. On the one hand, the original NDP is 7.1× and 6.9× faster than the CPU baseline on graph construction and graph traversal. This result indicates the simply parallel NDP solution does not fully utilize the hardware, considering the number of cores in the NDP system is much larger than that in the CPU baseline. On the other hand, the optimized solution is 32.5× and 16.4× faster than the CPU baseline for graph construction and graph traversal, respectively. The performance improvement provided by `Opt-HW` over `Original NDP` results from the reduced inter-core communication caused by the proposed optimization techniques, including bucket shuffling, message buffer and compression, and speculative contig expansion.

Comparison to software-implemented optimizations. The result shows that the performance improvement for graph construction is more significant than that for graph traversal. It is because the de Bruijn graph has a random structure that may cause cores to have unbalanced workloads. It is also consistent with the observation that the NDP solution performs better on a large genome than a small genome. In general, the proposed techniques perform better on large genomes that exhibit high-degree parallelism and sufficient per-core workload to exploit the parallelism of NDP hardware. `Opt-HW` outperforms `Opt-SW` by 3.1× and 3.2× on average for graph construction and graph traversal respectively. The performance gain

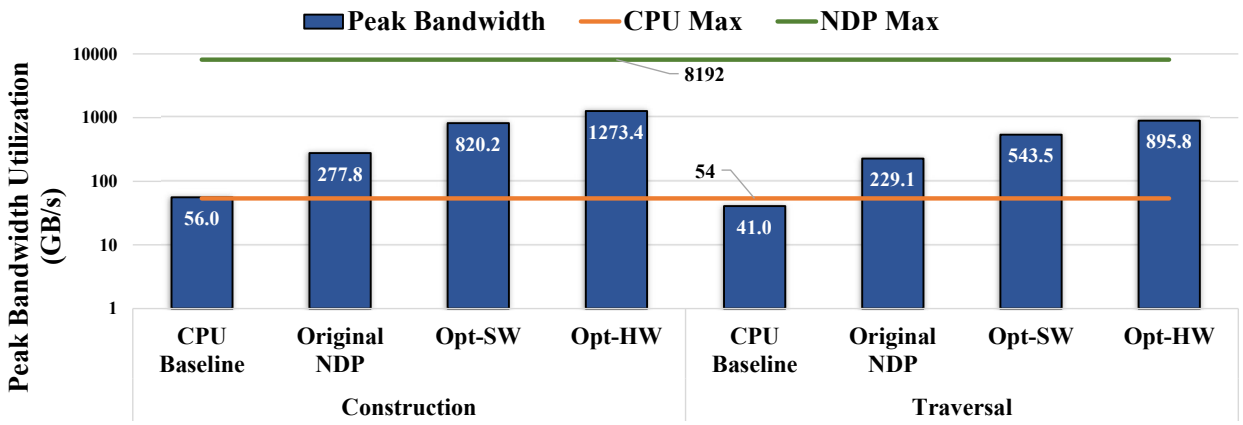


Figure 7.13: **Memory bandwidth utilization for `Human` genome.**

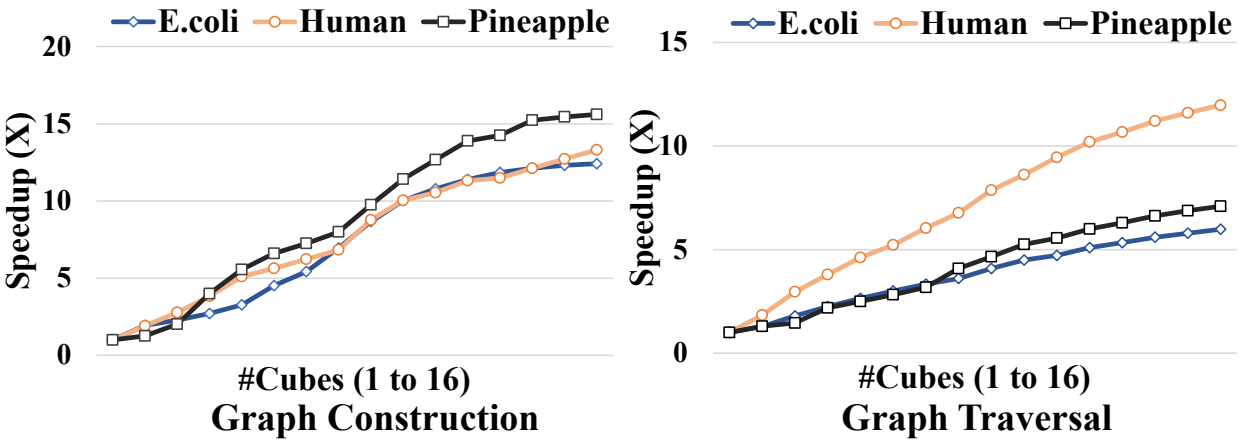


Figure 7.14: **Scalability results from 1-cube to 16-cube.**

provided by the hardware-implemented optimizations results from the reduction of memory accesses and updates for the optimization data structures.

Bandwidth utilization. Figure 7.13 shows the memory utilization for different systems running graph construction and graph traversal on *Human* genome. We get the CPU-baseline result from VTune [234] and NDP configurations from simulation. The results show: (1) NDP solutions, which run 512 parallel threads, require significantly more bandwidth than the CPU baseline maximum bandwidth. (2) The proposed optimization can increase the memory bandwidth utilization because of the better performance than the NDP baseline.

7.7.1 Performance Scalability

Figure 7.14 shows the performance of DBG assembly on the different number of NDP cubes for three genomes. We scale the system from 1 cube to 16 cubes. The 16-cube NDP system is 12.4× to 15.6× faster than the

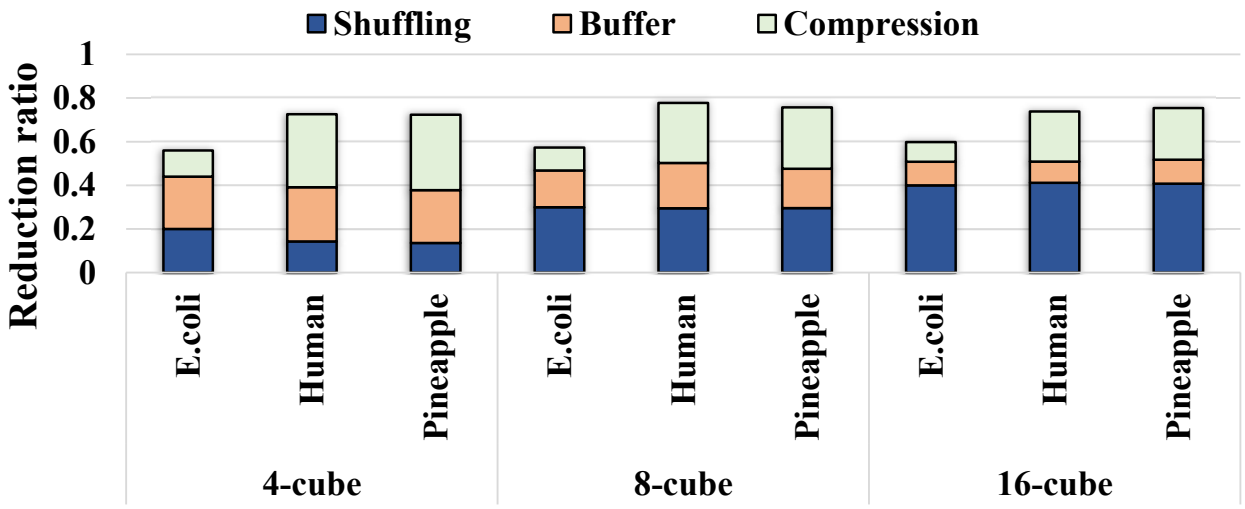


Figure 7.15: **The reduction of inter-core message passing provided by optimizations for graph construction.**

single-cube system over three genomes for graph construction. Such results show good scalability of the NDP implementation. The performance of NDP implementation depends on DNA patterns in the dataset. Suppose the dataset has a lot of repeated patterns (e.g., Pineapple). In that case, the NDP implementation has better scalability, because we can significantly reduce the data access time by mapping the buckets of repeated patterns in the same core with the corresponding sequence.

For graph traversal, the 16-cube NDP system is only 6.0×, 12.0×, and 7.1× faster than the single-cube system for E.coli, Human, and Pineapple, respectively. The overall improvement provided by the large systems is much less than that in graph construction. The reason is that graph traversal has more randomness in the workload, thus is less likely to schedule balanced workloads over the NDP cores in the large system. The results over different genomes also show that the NDP implementation has better scalability in the large genome.

7.7.2 Inter-core Communication Reduction

Figure 7.15 shows the effects of the proposed optimization on the reduction of the inter-core message. The reduction ratio is calculated in the order of shuffling, buffering, and compression. Our experimental results show that bucket shuffling can reduce 14% and 40% of inter-core messages in a 4-cube system and a 16-cube system, respectively, over a random bucket mapping scheme. The gap between small systems and large systems results from that small systems have an even distribution of buckets because each core is allocated more sequences than larger systems.

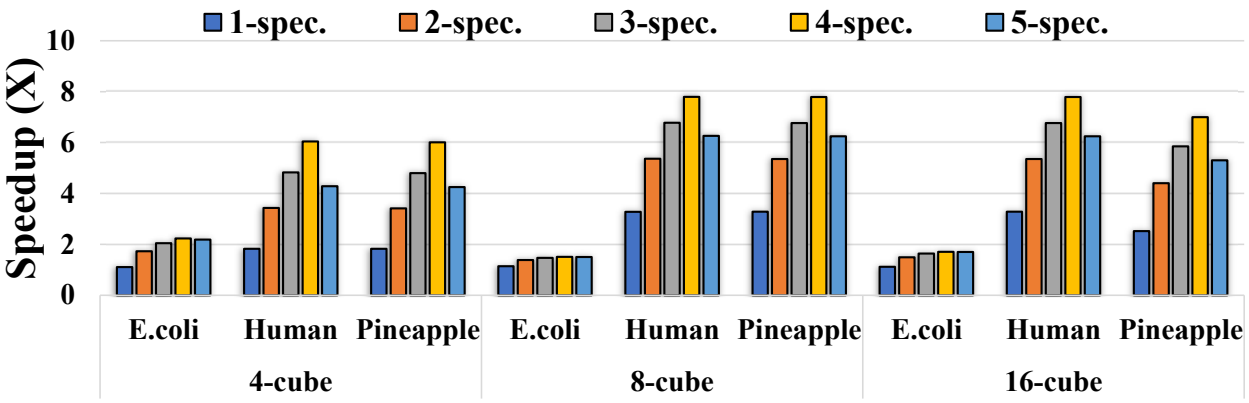


Figure 7.16: **The performance comparison among different steps for speculation.**

Unlike the bucket shuffling, the message reduction provided by k-mer buffering and compression becomes less when increasing the system size. Specifically, k-mer buffering (compression) reduces 24% (26%) of messages in the 4-cube system while reducing only 10% (15%) of messages in the 16-cube system. This is because large systems schedule fewer messages for each core, so that the opportunity for buffering and compression becomes less than smaller systems. In general, our experiment on the data movement reduction shows that the bucket shuffling and k-mer buffering and compression can work well together to reduce the number of inter-core messages in different sizes of systems.

7.7.3 Exploration on Speculation

Figure 7.16 shows the exploration of the speculation steps for graph traversal. We test different speculation steps and show the speedup over the baseline without any speculation. The result shows that the four-step speculation has the best performance for all workloads on systems of different sizes. Smaller speculation steps may not fully exploit the available memory bandwidth and parallelism of the NDP system, while a larger speculation steps have larger overhead of resolving the conflicts between different NDP cores.

7.7.4 Exploration on Network

Because previous works show that the NDP system's interconnect plays a critical role in the performance and energy consumption, we also explore different interconnect structures in the baseline NDP architecture. Figure 7.17 shows graph construction and graph traversal execution time on three structures: mesh, dragon-fly [223], and an ideal fully-connected network. All results are normalized to the mesh structure. The experiment shows that the dragon-fly network can improve the mesh structure's performance by 1.3× on average, while the performance-optimized ideal network is 1.7× faster than the mesh. However, the ideal network incurs 5.4X and 2.7X higher area overhead than the mesh and dragon-fly configurations.

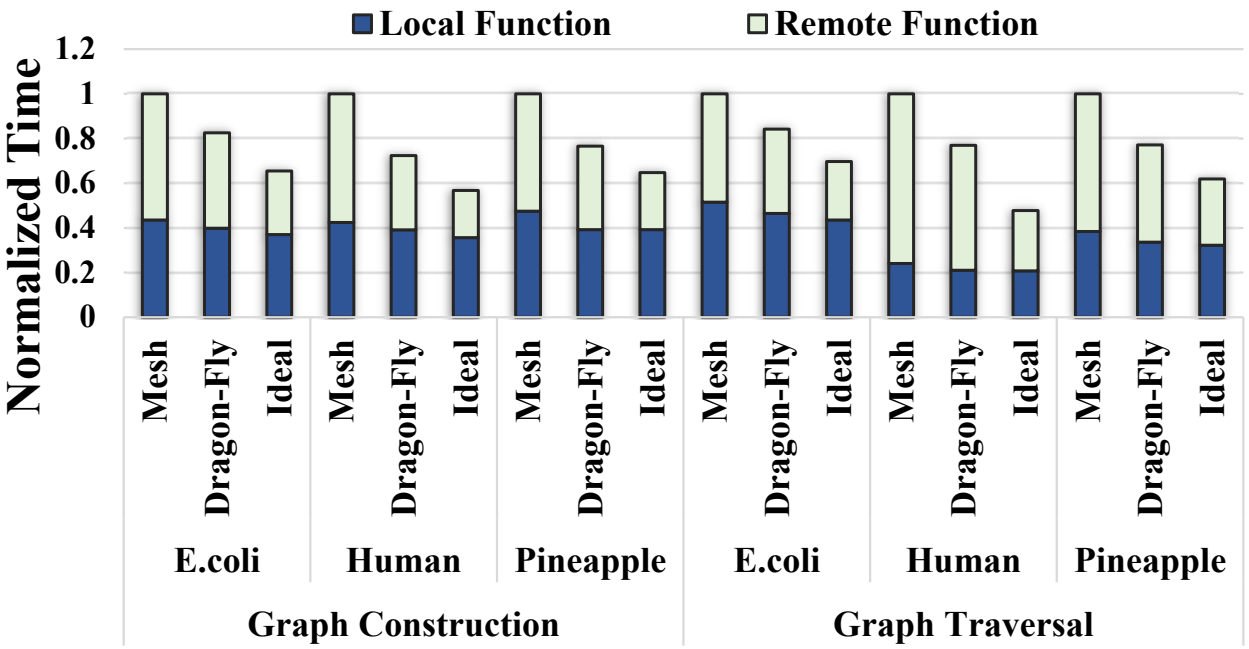


Figure 7.17: **The performance comparison among different network structures.**

7.7.5 Energy Efficiency

We estimate the energy of NDP system based on the average active cycles of cores and memory and the power values reported in the official product description [46] and previous works [17, 49]. Our results show that the proposed NDP system consumes 28.9 × and 15.0 × less energy for graph construction and graph traversal than CPU. Such energy reduction mainly comes from the faster execution. The average power consumed by NDP is higher than the CPU baseline because of the higher power consumed by the memory layers and the NoC power consumption. However, previous work [17] shows that such power consumption density in the memory chip will not exceed the thermal constraints. In this work, we only added a small storage component with a controller in the original NDP hardware, which has trivial power and area overhead ($< 2\%$). Therefore, the proposed NDP-based DBG assembler is practical in terms of power and thermal efficiency.

7.7.6 Comparison with Other Distributed Algorithms

The DBG processing of large genomes is also deployed on distributed-memory parallel computers using frameworks such as MPI due to their scalability (large capacity and high core count). Notable distributed-memory assemblers are Ray [235], PASHA [69], YAGA [236], ABySS [67], HipMer [237], and PakMan [238]. This work shares similarities with distributed-memory DBG assemblers at high-level. For example, addressing the communication imbalance issues during the parallel graph construction phase and avoiding traversing the same contig by multiple processing nodes repeatedly. If handled inefficiently, the overhead of orchestrating

nodes outweighs the performance benefit of parallelization. These challenges are typically not found in shared-memory DBG assemblers, which primarily focus on optimizing algorithm complexity and assembly quality. However, migrating existing distributed-memory DBG schemes into an NDP system is a complex undertaking. Each node in the distributed-memory system handles multi-threading workloads with large memory footprints, but each NDP core is single-threaded with limited memory capacity. For example, PakMan [238] compresses the DBG into a compact graph with macro-nodes to ensure each compute node can fit the whole compact graph during the graph traversal phase, where each process can concurrently traverse multiple independent paths. Therefore, the optimization for cross-node communication in distributed-memory systems is too coarse-grained in the NDP implementation.

We provide an indirect comparison with one of the state-of-the-art distributed assemblers. As reported in the previous work [238], PaKman offers 9.3× speedup over IDBA-UD [239] with 40-cores@2.2GHz in its MPI-based shared-memory mode. Assuming performance scales linearly with core frequency, and since both PaKman and our work demonstrate linear scaling w.r.t core count, PaKman offers 54.08× speedup over IDBA-UD with 512-cores@1GHz. With 512-cores@1GHz, our work outperforms MEGAHIT by 31.6× which is already 3.5× faster than IDBA-UD (110.6×). This result shows our design is about 2× faster than PaKman even if PaKman can be perfectly mapped to an NDP architecture. Finally, this work leverages software/hardware codesign to speedup DBG assembly. Without the appropriate hardware support (e.g., the latency/bandwidth advantages of PIM and our customized hardware components), the software optimizations alone do not achieve the best results.

7.8 Related Works

Non-genome NDP accelerators. There are similar 3D-stacked NDP accelerators for graph processing [17, 48], pointer chasing [118], and large-scale data analytics [50, 240, 49]. Some aspects of these work are similar to ours, such as minimizing communication, optimizing data partitioning, and providing a framework for the proposed architectures. However, these works are not directly applicable to DBG.

PIM bio-accelerators. There are several PIM accelerators for bioinformatics workloads. Wu et al. [53] proposes an in-situ solution which fits minimalist bitwise operation logic inside DRAM chips, and utilizes subarray-level parallelism to support massively parallel K-mer matching. GenCache [84] modifies the SRAM chip to support sequence alignment. Medal [85] leverages off-the-shelf DRAM components to build a DNA seeding accelerator. RADAR [86] is a 3D-ReRAM based accelerator for BLAST. AligneR [241] is a ReRAM-based PIM architecture which accelerates the bottleneck stage of genome sequencing. FindeR [242] enhances the FM-Index EPM search throughput in the gnomic sequencing step using commodity ReRAM

chips. These works target different stages of genome pipelines. To the best of our knowledge, this is the first PIM-based *De Novo* assembly accelerator.

DBG assemblers. We have compared this work to prior distributed-memory DBG assemblers in Section 7.7.6. There has a limited effort of porting de Bruijn graph onto GPU such as [230, 231]. They either focus on only one stage of DBG assembly or only work with small genome. In contrast, we provide comprehensive support for every stage of DBG and work with a much larger genome.

7.9 Conclusion

In this work, we propose a software-hardware co-design for DBG assembly that leverages emerging 3D-stacked memory architectures with high parallelism and bandwidth. We identify graph construction and contig assembly as two bottleneck stages, as they suffer from high communication overhead due to frequent message passing. By exploiting real DNA sequence characteristics, we optimize our design with an effective data partitioning strategy and a message buffering and compression technique to reduce inter-core communication. We also develop a speculation scheme to extend each contig by multiple bases each time tentatively. The optimizations above synergistically offer the combined benefit of speedups and energy savings over the CPU by 24× and 22×. Our NDP-based DBG processing framework can significantly reduce the run time of many critical steps in analyzing human and microbial genomes, which aids in disease diagnosis, precision medicine, vaccine development, and other tasks.

Chapter 8

Abakus: Accelerating k-mer Counting With Storage Technology

8.1 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 [59, 243, 244, 245, 246, 247, 248, 249, 250]. k-mer frequency information is also extensively used in the identification of repeat sequence regions [251, 252, 253, 254, 255, 256], variant calling [257], and alignment of multiple DNA or protein sequences [258, 259, 260].

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 [53]. 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 ∼3600 CPU hours, out of which ∼60% is spent on k-mer counting [59]. 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 [10], and the number of reads that need to be assembled is growing at a rate vastly outstripping Moore's Law [56], 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 [261, 262, 263, 264]. 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 [265]. Furthermore, multiple threads are bound to compete for accessing the same set of k-mer entries, resulting in frequent serialization [264]. 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, 266, 267, 268, 269, 270] 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 [7] reveal that a significant portion of its execution time (over 75%, see Section 8.3.1) is spent on file handling alone, constantly stalling the processor.

Several prior efforts have sought to accelerate k-mer counting using GPUs [7], FPGAs [271, 272], and processing-in-memory architectures [273, 15, 59]. However, these approaches do not consider the I/O

bottleneck – while some only accelerate the compute-intensive counting phase [7, 271] or assume that the data is already loaded into memory [59, 273], others accelerate one-phase in-memory k-mer counting [272, 15] 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 trade-offs 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 [274, 275, 276, 277, 278, 279, 280, 281]. Commercial products include Samsung's SmartSSD [161], 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 [275, 282, 277]. 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 [283, 284, 285, 282, 286], 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 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,

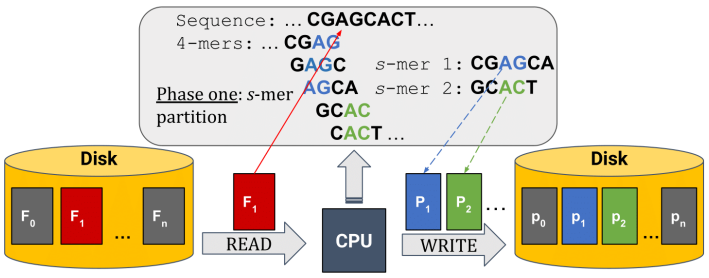
similar to [283] and [285]. The large size of the bioinformatics market suggests that there is a potential market for a product that is purely an accelerator that overcomes the I/O bottleneck. Furthermore, future computing environments are increasingly more likely to be heterogeneous and accelerator-abundant [287, 288]. Therefore, we envision Abakus to be deployed in the cloud with other genomics accelerators to fulfill the need for 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 comprised 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.

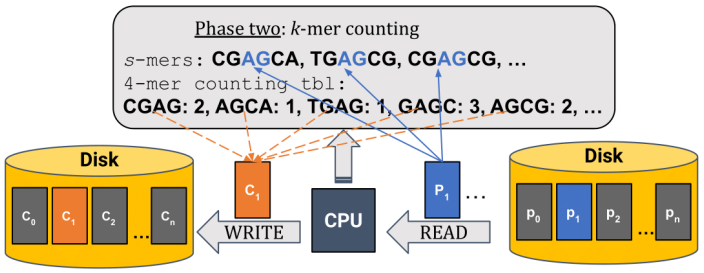
8.2 Background

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 [264]. However, Jellyfish might have difficulty handling large genome files because it keeps the histogram in memory [265]. This is the limitation of in-memory k-mer counting tools



(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

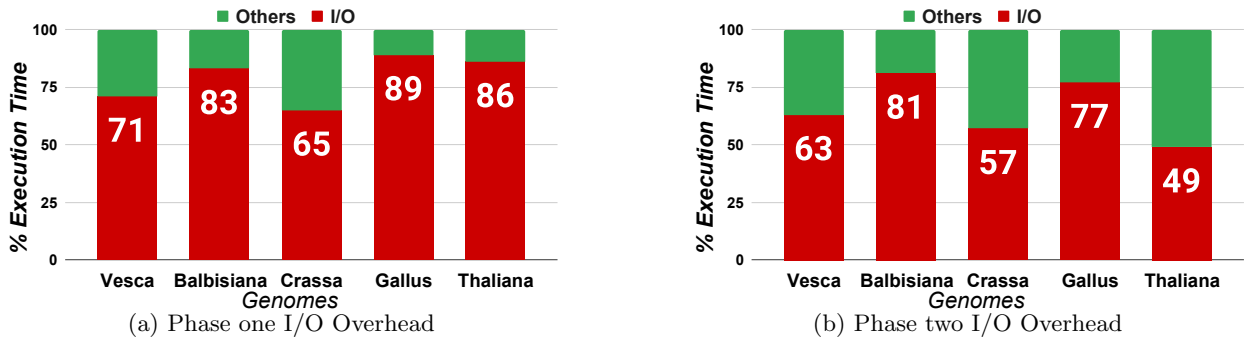
```
1  histogram.initialize()
2  for (partition in partitions):
3      hashTbl.initialize()
4      for (s-mer in partition):
5          kmer_set = extract_kmers(s-mer)
6          for (k-mer in kmer_set):
7              hashTbl[k-mer] += 1
8      histogram.add(hashTbl)
```

Figure 8.1: 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.

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 [265]. 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 [7], KMC3 [266], 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). Fig. 8.1 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 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. Fig. 8.1 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



(a) Phase one I/O Overhead

(b) Phase two I/O Overhead

Figure 8.2: Gerbil [7] I/O overhead.

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 [7, 266].

Counting Phase. In this phase, each partition file is read from the disk to memory for k-mer extraction and counting (Fig. 8.1). Both hashing and sorting-based approaches are viable, but sorting can be slower for longer k values [265, 7]. 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.

8.3 Motivation

8.3.1 I/O Is the Bottleneck

Prior work [15] has shown that I/O greatly affects the performance of Gerbil, one of the best k-mer counting tools available today [7, 265]. 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 [234] – 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. Fig. 8.2 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 $\sim$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.

8.3.2 ISP k-mer Counting Considerations

Benefits of PWST. While k-mer counting can be accelerated through GPU [7], FPGA [271, 272], and even near-data-processing approaches [273, 15], 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 ($\sim$1.2 GB/s per channel × number of channels [285]), while the external bandwidth ($\sim$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 data set 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 data set 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 [289].

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 [274, 275, 276, 277, 278, 279, 280, 281], and (2) a PWST solution with distributed and dedicated custom compute elements deeply integrated along the SSD internal data path to do the processing [283, 284, 285, 282, 286]. While both successfully reduce the data volume coming out of the storage devices, they have different capabilities and trade-offs. We argue that the second approach is more suitable for k-mer counting. The embedded commodity SSD controller is usually an energy-efficient CPU (3-4X lower power than the host CPU) clocked at merely several hundred megahertz [275, 52, 282] 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 [274]. 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 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 8.6.1), 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.

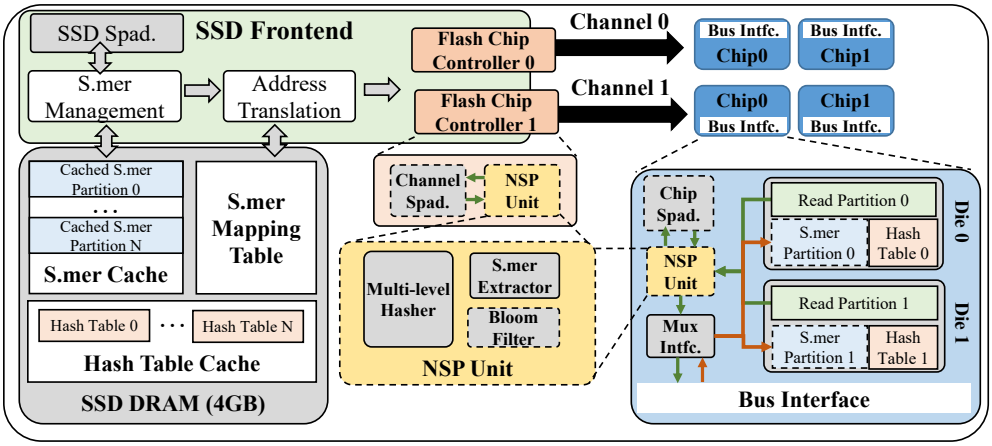


Figure 8.3: The overall architecture of Abakus.

This has been noticed in prior work [286, 283], 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[286].

Finally, prior work [290] finds that SmartSSD [161] 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.

8.4 Architecture

8.4.1 Overview of the PWST Architecture

Fig. 8.3 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, can only exploit chip-level parallelism. In the next section, we describe mechanisms to integrate logic into the channel and SSD levels to extract greater performance.

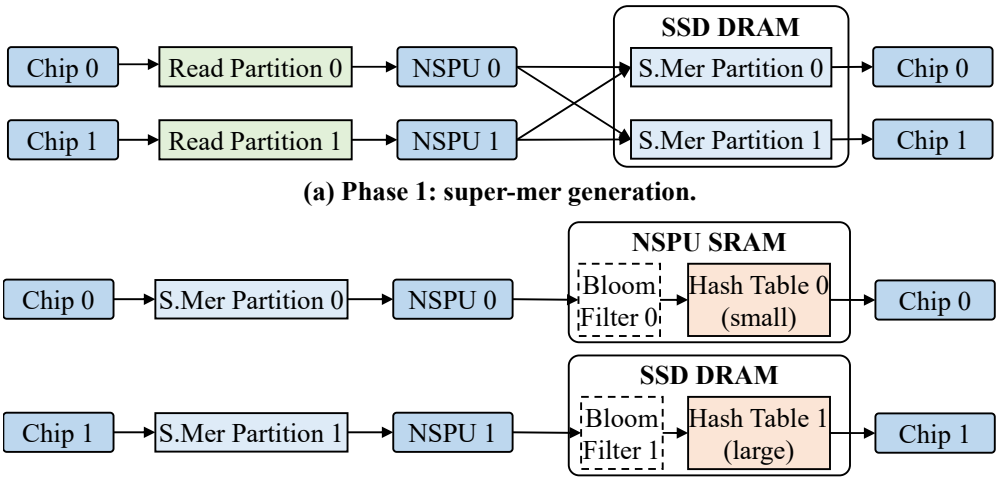


Figure 8.4: The basic two-phase hardware workflow of Abakus. Bloom filter is effective in Abakus-BF

8.4.2 Abakus-Basic Overview

Fig. 8.4 shows the design and workflow of Abakus-Basic that directly maps the two-phase algorithm onto the SSD.

In the first phase, reads are split into s-mers that are then gathered into the same partition if they share the same minimizer (Sec. 6.2). The raw input reads are evenly distributed to each chip a priori so that 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, 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 Sec. 8.5. 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 [7], 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. Each chip-level NSPU has a small bookkeeping data structure ($\sim$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 run time, 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 (Sec 8.9 Abakus-OP) as an optimization.

8.5 Partitioning Strategy

Clearly, the performance of the proposed design is bottlenecked by the number of large partitions whose hash tables won't 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 (4 possible base pairs at each position). If $m = 9$ and we let the chip-level scratchpad size be 1 MB, 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.

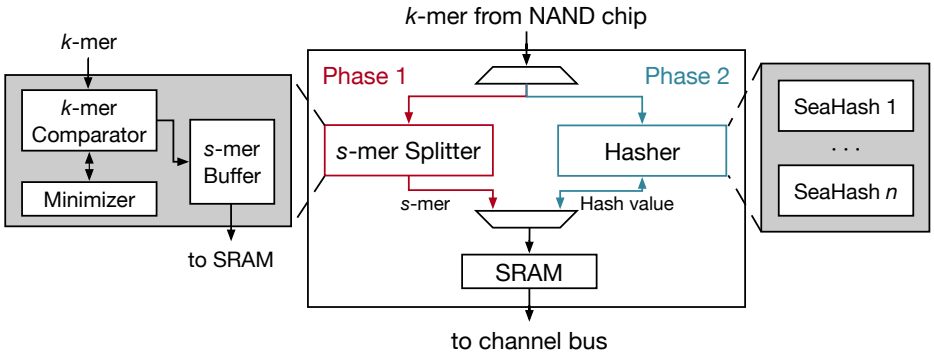


Figure 8.5: Diagram of the near-storage processing unit (NSPU).

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's unlikely to have more NSPUs than partitions. All proposed partition schemes would distribute an equal amount of partitions (2048) 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 (Sec. 8.11.4).

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.

8.6 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.

8.6.1 Chip-level NSPU

Each flash chip implements one NSPU to provide k-mer counting-related computations. As shown in Fig. 8.5, 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 [291], 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 Fig. 8.6). 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.

8.6.2 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.

8.7 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 data

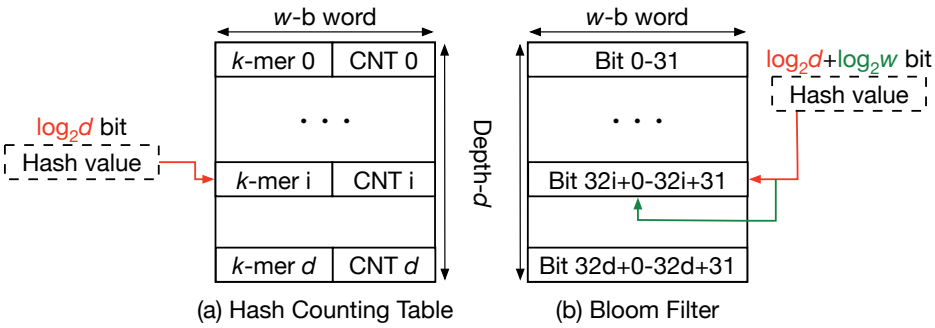


Figure 8.6: Mapping for Hash Table and Bloom Filter modes

set, 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.

8.8 Abakus-BF

8.8.1 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 [59, 243, 244, 245, 246, 247, 248, 249, 250]). 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, 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 8.2.

Notice each k-mer pattern can only appear in one specific partition, thanks to the minimizer-based partitioning strategy. Since each partition is assigned to a specific chip/NSPU, and no partition is split to more than one chip, there won't be any k-mer patterns that appear in more than one private Bloom filter.

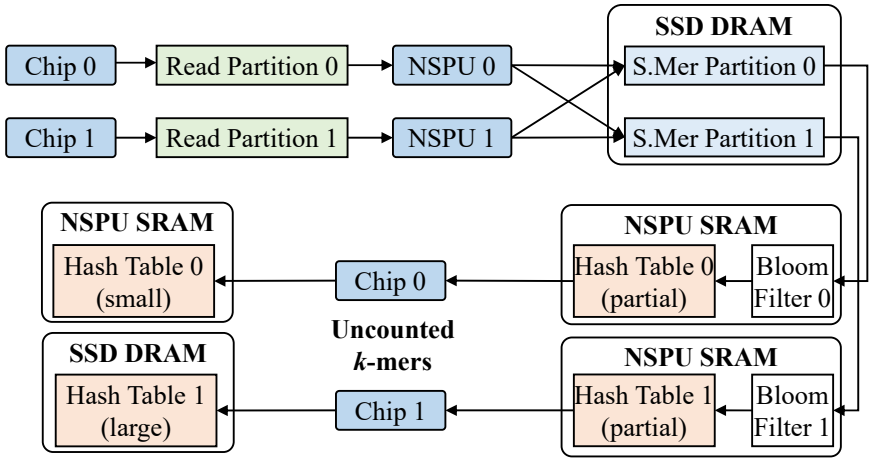


Figure 8.7: Abakus-OP Workflow

Determining when, where, and how to incorporate a Bloom filter into Abakus is a large design space exploration problem. In this section, we introduce one such solution (Abakus-BF) where a set of Bloom filters are instantiated in phase two at the chip level. In the next section (Sec. 8.9), we discuss another variation where the Bloom filters are used earlier.

8.8.2 Abakus-BF Overview

Fig. 8.4(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 run time. 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.

8.8.3 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.

8.8.4 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 [292] that 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.

8.9 Abakus-OP

8.9.1 Motivation

The performance of Abakus-Basic and Abakus-BF is primarily limited by the chip-level SRAM scratchpad memories (512KB 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 (Sec. 8.4.2), 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 [283, 284, 286], trading parallelism for better processing power and area budget [283, 286]. 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 Fig. 8.3) 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.

8.9.2 Abakus-OP Overview

Figure 8.3 illustrates the architecture, and Figure 8.7 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 256KB to 32 MB for a sensitivity study in Sec. 8.11.4) 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.

8.9.3 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 writeback 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.

Table 8.1: 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 512KB	0.48	63.5	0.008
SRAM 2MB	1.71	617.7	0.017
Abakus-Basic (128 × 512KB SRAM/chip)	0.51/chip	**Peak Power (W)** 8.6	
Abakus-BF (128 × 512KB SRAM/chip)	0.51/chip	**Peak Power (W)** 8.6	
Abakus-OP (32 × 2MB SRAM/channel)	1.83/channel	**Peak Power (W)** 20.0	

8.10 Methodology

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 [273]. For CPU and GPU baselines, we use a state-of-the-art disk-based k-mer counting tool, Gerbil [7], that provides the best performance and memory efficiency among other tools [265]. The DIMM-based accelerator, NEST [273], 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 128GB 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 [161]) which has an SSD-level FPGA accelerator with DDR4 SDRAM@2400Mbps, consuming 25W power in total. We assume SmartSSD has infinite (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 64GB 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 2TB size and 3D TLC. It is an integrated PCIe 3 ×4 bus and LPDDR4-4266 DRAM to realize a peak 3.5GB/s sequential read rate. For a fair comparison, we follow a similar methodology described in a prior in-storage acceleration paper [283] 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. The performance of simulated CPU and GPU baselines are 7.6% to 12.8% faster than the performance measured on the real machine.

Table 8.2: 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

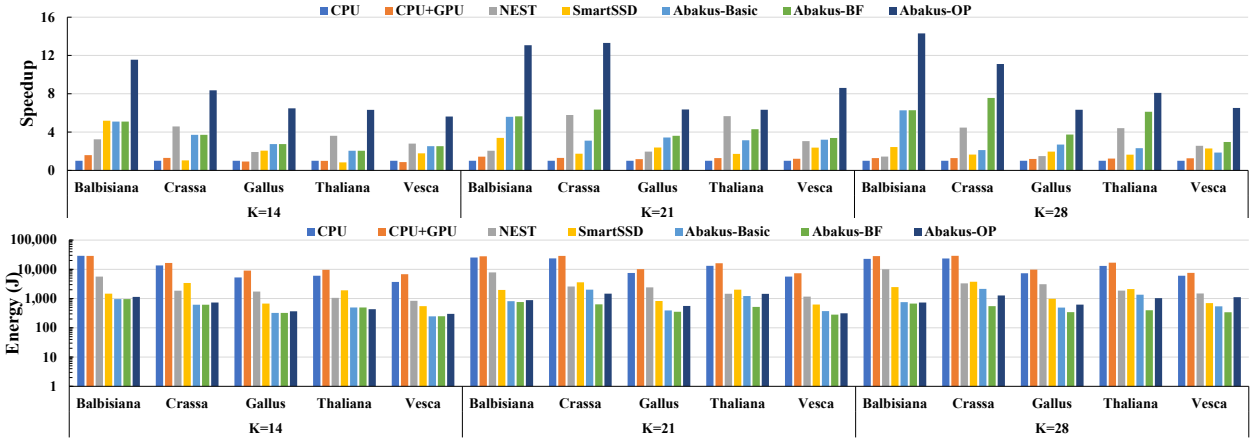


Figure 8.8: The overall performance and energy across different platforms, genomes, and k sizes.

Workloads. We evaluate five genome datasets from different species: Balbisiana, Crassa, Gallus, Thaliana, and Vesca (see Table 8.2), that are large enough to sufficiently exercise all hardware components in Abakus. All datasets are downloaded from NCBI [293] by entering their SRA codes from Gerbil [7].

Simulation Infrastructure. We model the performance of Abakus in a modified, trace-driven, state-of-the-art SSD simulator, MQSim [52]. 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 [52] simulates end-to-end behaviors of SSD requests, including the host, the device, and host-device communication (e.g., PCIe bus). Table 8.1 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 700μs write latency for an 8KB page [294, 295]. 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 40nm technology node. The clock frequency is 200 MHz and the design is scaled to 22nm. Timing and energy values of SRAM are extracted from CACTI-3DD [156] in 22 nm.

8.11 Results

8.11.1 Area Overhead Analysis

Table 8.1 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 0.51mm^2 additional overhead for each chip. Abakus-Basic and Abakus-BF have the same total area of 65.9mm^2 while Abakus-OP (58.6mm^2) is slightly smaller because 2MB SRAM has higher area efficiency than 512KB SRAM. We observe that state-of-the-art flash chips [296, 297] have over 120mm^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 2MB SRAM is implemented in FMC. Although we lack the resources to model the exact area of FMC, we note that LDPC ECC [298], a module implemented in FMC, has a comparable area with 2MB SRAM. Therefore, we believe that all three Abakus variants are practical for manufacturing and have a minor impact on storage density.

8.11.2 Overall Performance and Energy Efficiency

Figure 8.8 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 (64MB) in all architectures for a fair comparison. Specifically, both Abakus-Basic and Abakus-BF have a 512KB scratchpad in each chip, while Abakus-OP features a 2MB 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 [161], Abakus-OP is 3.47× faster and consumes 2.18× less energy. 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 are a large percentage of single-occurrence k-mers, as evidenced by Crassa and Thaliana genomes when $k = 28$ (See Table 8.2). 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

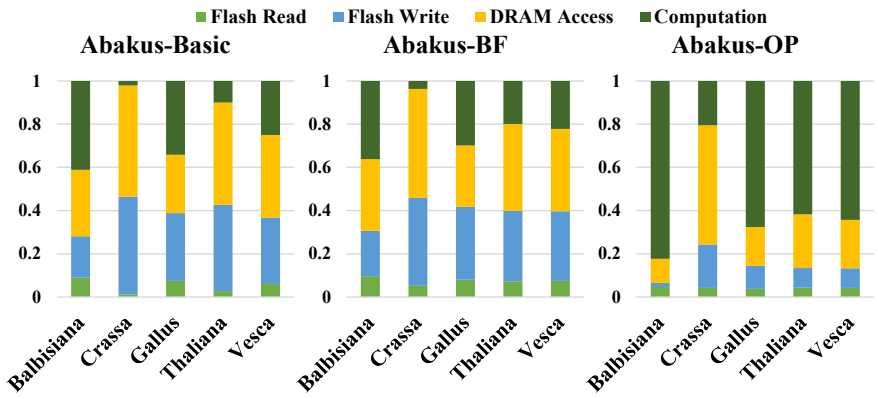


Figure 8.9: Performance breakdown for the three Abakus designs.

(Sec. 8.4.2 and 8.6.2), 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 8.2, are rather small, further reducing the number of Flash writes, providing a substantial speedup. Third, while Abakus-OP outperforms other Abakus setups as well as 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 (Sec 8.9.2), resulting in a significant increase of read/write commands that are handled by one chip.

8.11.3 Performance Breakdown

Fig. 8.9 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 that 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.

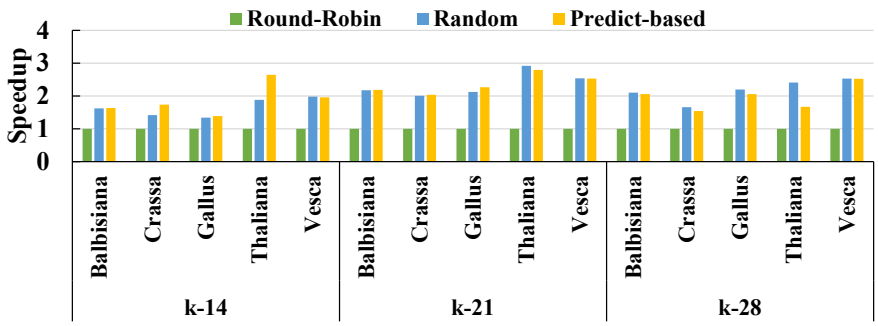


Figure 8.10: The performance of different partitioning strategies.

8.11.4 Sensitivity Analysis

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 Sec. 8.5). 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. Fig. 8.10 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.

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.

Buffer Size

Fig. 8.11 explores the performance sensitivity due to varying the SRAM buffer size. As compared to 4MB, 8MB, 16MB, and 32MB channel-level buffers, we observe that the 2MB buffer is 1.03×, 1.08×, 1.17×, and 1.43× slower. However, if the buffer size is larger than 4MB, the custom hardware requires more than 42.7W power and 104.2mm^2 area in a 32-channel SSD. At the same time, 2MB is 1.19× faster than 1MB while only

requiring 19.8W power and 54.6mm^2 area overhead. Therefore, 2MB 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 (Sec 8.9.2). However, smaller scratchpad designs can exhibit better area/power efficiency than larger scratchpad designs. For example, Abakus-OP with 512KB scratchpad per channel-level NSPU is 1.98× slower but only requires 15.47mm^2/2.03W area/power overhead, which is 3.53×/9.73× better than its 2MB counterpart. Furthermore, 512KB configuration is only 4% slower than its 2MB 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.

8.12 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 [299, 285, 276, 277]. 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 [273, 272]. In general, k-mer counting algorithms do not have to be exact for most use cases [300]. 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} [301]). 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 [301]. 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 [247] 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 8.5 ensures that each chip

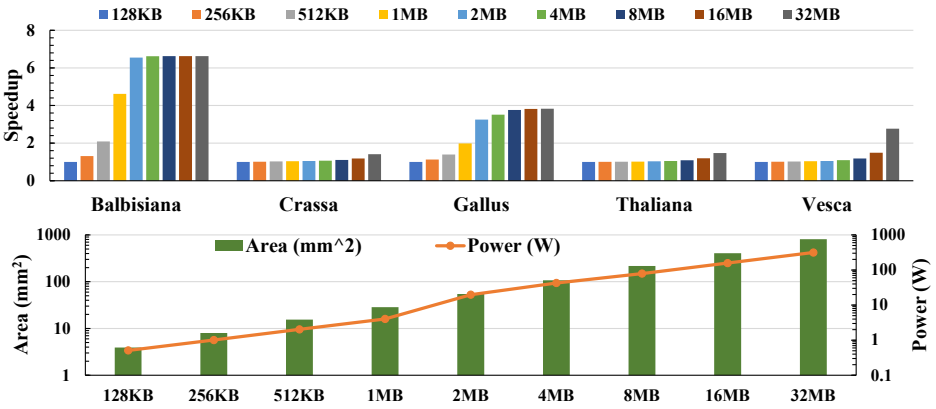


Figure 8.11: Exploration of different buffer sizes for Abakus-OP.

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 interact 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, which happens to also support wear-leveling to a certain degree, avoiding the necessity of designing a custom garbage collector.

8.13 Conclusion

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×.

Chapter 9

New Hardware Trojan Threats in Memristor-based Neuromorphic Computing Systems

9.1 Introduction

Deep neural networks have been extensively employed in several applications. However, their execution on traditional architectures has shown to be inefficient due to their inherently memory-bounded nature, a problem that has been exacerbated by rapidly growing model and input data sizes. For example, data movement in GoogLeNet accounts for roughly 70% of the overall energy consumption [14]. In recent years, emerging non-volatile memory (eNVM)-based neuromorphic computing systems that emulate biological computing with artificial neurons in the analog domain, have gained traction, due to their high energy efficiency and throughput advantages [302]. Their security implications, however, remain largely unexplored.

The process of setting up a trusted end-to-end production line for eNVM-based neuromorphic accelerators can be prohibitively expensive, especially considering frequent algorithmic updates and tuning of models. This has paved the way for a decentralized design-manufacturing approach that involves a coordinated effort among multiple stakeholders. However, it also inevitably invites exploitation from bad actors to stealthily inject malicious hardware Trojans into the product [27]. Semiconductor supply chain attacks are critical threats that can disrupt the operations of high-value mission-critical systems such as military, financial, and medical infrastructure.

This work is the first to demonstrate the feasibility of carrying out a hardware supply chain attack against analog eNVM neural accelerators to leak IP-sensitive synaptic weights. We discuss potential Trojan insertion points within the supply chain and due to the lack of openly available commercial implementations, we dissect a generic eNVM accelerator derived from recent works to identify vulnerable probe points. The crux of this work is the design and stealthy placement of a neuron-suppressing hardware Trojan that can be reliably triggered by a colluding adversary. The findings from this research are expected to foster the design of such eNVM neural accelerators with a security focus.

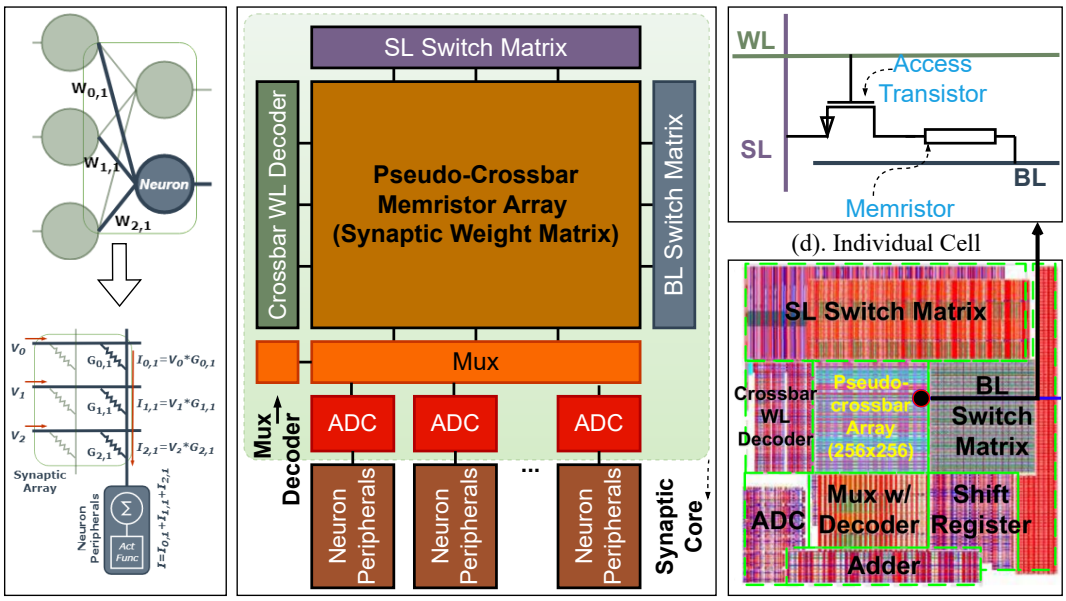
There are two major motivations for such a model extraction attack that aims at cloning a victim model of similar performance without going through the expensive training process. First, the synaptic weights of a neural network are considered core IP as they separate a properly trained network with high accuracy from a poorly trained one. Second, stealing weights is increasingly more economical than training. To obtain a model with competitive performance, typically, a large set of high-quality labeled data and proprietary training algorithms are required [303]. Further, even with access to proprietary training data, the process of training can take weeks and is likely to worsen with model sizes, affecting the time-to-market [304].

The key to our attack is the fact that synaptic weights are encoded as conductances of eNVM devices in the analog eNVM device array, and the total current representing a dot-product result depends on the synaptic weights. This allows for the isolation of the switching activity of a single neuron, enabling the sequential extraction of all the synaptic weights using power side-channel analysis while evading detection.

9.2 Background

Analog eNVM Neuromorphic Devices. Fig. 9.1a illustrates the mapping of an MLP layer onto an eNVM device. All incoming synaptic weights of the highlighted output neuron $(W_{0,1}, W_{1,1}, W_{2,1})$ are stored as distinct conductances $(G_{0,1}, G_{1,1}, G_{2,1})$ of the eNVM cells along the same column. Suppose the inputs to this neuron are encoded as voltage levels applied to the wordlines (V_0, V_1, V_2), then each cell contributes a small current of $V_\mathrm{i} \times G_{\mathrm{i},1}$ Ampere to the bitline. By Kirchoff's Law, the total current passed to the neuron circuit at the end of the bit line is the sum of the three partial currents generated at each cell, representing the dot product of the input vector and the weight vector of a single neuron.

Synaptic cores (SC) are the fundamental building blocks of a neuromorphic architecture (Fig. 9.1b). They consist of a 2D synaptic device array that stores a weight matrix in the form of conductance levels and supporting peripheral circuits. An additional access transistor is inserted per eNVM cell to mitigate the current sneak-path problem (Fig. 9.1d). During the weighted sum operation, wordlines (WLs) are switched on in parallel, thereby selecting multiple rows of eNVM cells. Inputs are provided as serial bit vectors to



(a). NN Layer to Synaptic Array (b). Synaptic Core Block Diagram (c). Sample Synaptic Core Layout

Figure 9.1: **Synaptic Core layout**[8] **and Neuron Architecture.**

the memory cell and the generated currents on the selectline (SLs) represent weighted sums that propagate toward the neuron peripherals. The results of the matrix-vector multiplication are first converted by a series of ADCs (described next) to digital values, then sent to the neuron peripherals for activation, in turn producing the results bit vector to be sent to the next SC to select a set of BLs, subsequently activating neurons of the next layer. To save area, both ADC and neuron peripherals are shared among artificial neurons through a multiplexer [302, 19].

Neuron ADC. Fig. 9.2a shows a generic Integrate-and-Fire ADC, which consists of a thermometer code generator and a Thermometer-to-Binary encoder. Such a design is popular as it provides good energy efficiency [8, 302, 305]. Since the resulting cumulative current is to be *integrated* as a potential that represents the weighted sum, the integration is carried out as a potential buildup on a capacitor (C_{colmn}), and as the potential crosses a threshold value (V_{thr}), the neuron *fires* in the form of a spike, causing an instantaneous discharge of the potential buildup to a predetermined base potential (V_b). The circuit designed for generating spikes is highlighted within the dashed lines and involves an inverter with a reset element that allows for instantaneous discharge and regeneration of a spike potential. Fig. 9.2b depicts the transient waveform showing the potential buildup on the capacitor up to V_{thr} and discharge to V_b. Fig. 9.2c shows the resulting train of spikes.

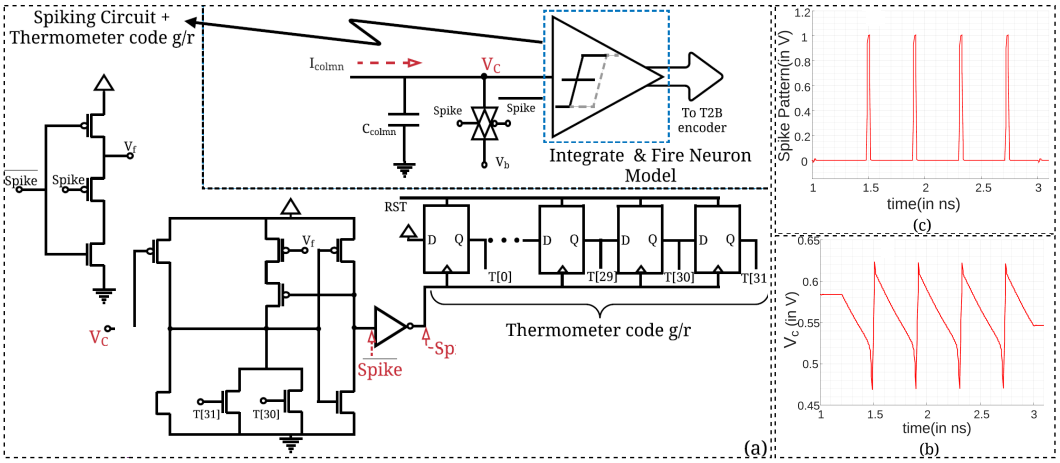


Figure 9.2: **Schematics and waveforms that depict (a). Generic Integrate and Fire Neuron model; transient signals that exhibit (b). current integration, and (c). spiking pattern.**

9.3 Related

Related Work. An adversary with physical access to an eNVM device can probe it to extract the weights directly [306, 307, 308]. However, such an attack is destructive as probing one cell could damage adjacent ones [306]. Moreover, encrypting data before the system powers down is an effective countermeasure [306, 308]. On the other hand, stealing weights online is superior as it is non-destructive and cannot be mitigated by encryption because, at any time, there is at least one layer of synaptic weights remaining in plaintext [308]. Rajamanikkam et al. [307] outline two attacks to compromise the *availability* of neuromorphic devices. The first makes use of current sneak paths to mount a fault injection attack by sending malicious inputs and leveraging leakage currents to alter synaptic weights, resulting in incorrect inference outputs. This can be mitigated by inserting gating transistors. The other attack embeds hardware Trojans to degrade classification accuracy, as opposed to our attack which steals IP-sensitive model weights, which is of greater interest because properly trained weights are usually hard to obtain due to lack of high-quality and proprietary training data and algorithms [303].

9.4 Threat Model

Attacker Intent. The attacker intends to extract the synaptic weights of a neural network from an analog neuromorphic system in two phases. First, a Trojan is inserted at the hardware design or fabrication stage. Second, the synaptic weights are extracted at the NN inference stage by activating the Trojan such that the resulting power trace can be attributed to the requested synaptic weight. The attacker at each of these phases might not necessarily be a single entity, but could involve two separate colluding malicious parties. A detailed

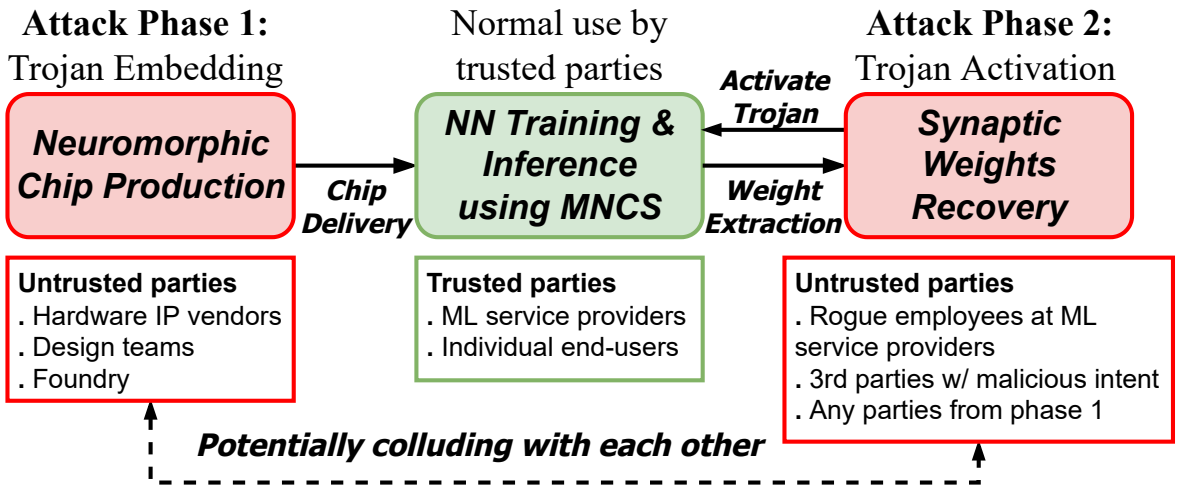


Figure 9.3: **Trusted and untrusted parties in the supply chain.**

overview that depicts a product development life-cycle and the potential parties involved in the two phases of the threat model is shown in Fig. 9.3. The malicious entities in the supply chain do not possess the intricate details of the NN models (including weights) but have the ability to embed a Trojan, given the distributed nature of modern IC supply chains. A colluding entity could then trigger the Trojan post-deployment using a known activation code and then steal sensitive IP information. Alternatively, a rogue engineer in the supply chain can cause damage by simply publishing the Trojan activation code without explicitly colluding with another player. Either way, even if a trusted entity is tasked with securely programming the synaptic weights into the device, it would still remain vulnerable to a Trojan placed in the supply chain.

Trojan Insertion Points. We consider three possible insertion points. First, the Trojan could be injected at a very early stage (e.g., in the HDL code). However, this might stand out under scrutiny during post-design verification. Second, the Trojan could be placed in open spaces in the GDSII layout file after the circuits have been placed and routed following the model described in [27]. Third, an attacker from an untrusted fab house could inject the Trojan, which entails reverse engineering the victim wires to tap into, leveraging the knowledge of algorithms used in floor-planning, placement, and layout tools, which has been shown to be feasible [309].

Grey-box Model. As is common with conventional grey-box models [310], we assume that the attacker is aware of the neural network structure, such as the number of layers, but not the IP-sensitive model parameters, i.e., synaptic weights. Many production ML services leverage well-known neural networks whose structures are publicly available (e.g., ResNet, VGG-16/19, etc.) [303], due to which, all entities along the supply chain would have access to the model structures. However, the weights learned during the training process is often proprietary. We also assume that a malicious entity with access to the Trojan trigger code has the ability to buy such a device from the market and activate the Trojan by sending arbitrary inputs to

covertly extract the synaptic weights through a power side-channel attack [310, 27]. We note that leaking synaptic weights in the absence of a Trojan is likely more challenging as it entails attributing signal leakage to particular weights.

9.5 Attack Overview

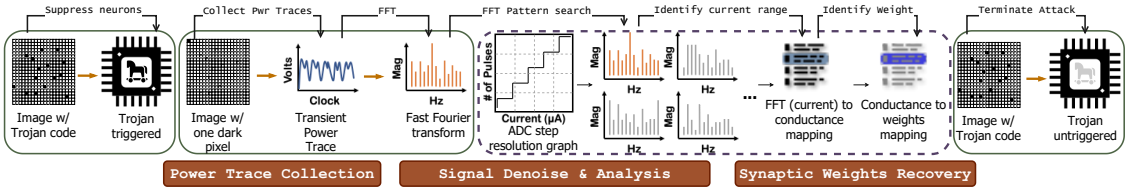


Figure 9.4: **Synaptic weights recovery through a Trojan-created power side-channel.**

9.5.1 Feasibility of Exploitation

The key insight to this attack is that the dot-product results are represented as analog currents, and the strengths of those currents are directly correlated to the synaptic weights, i.e., larger weights (conductances) produce larger currents. As this current is converted into a train of spikes by the Neuron ADC, it results in dynamic switching transients within the power trace, allowing the attacker to approximate weights by intercepting the power trace. Even if an alternative ADC is chosen, the generation of spikes would lead to a certain amount of switching activity, exposing it as a possible target for exploitation. Furthermore, ADCs consume over 80% of the total system power, allowing the attacker to estimate the power of ADCs using the global power trace [310]. Finally, since the ADCs are time-shared due to their large area, the attacker can target each ADC individually using our novel neuron suppression scheme that allows a malicious Trojan to isolate a particular neuron ADC, ultimately correlating the switching activity of the architecture to a single eNVM cell.

9.5.2 Attack Procedure

Online Weight Recovery. Fig. 9.4 illustrates the proposed Trojan-assisted power side-channel attack. The key idea is to attribute the observed power activity to a single ADC by sending specially-crafted input images containing Trojan codes (certain pixel patterns) to the device, which triggers the Trojan to iteratively select only one ADC to be functional and forces it to process a current generated by a weighted sum operation of one eNVM cell. An attacker can then collect power traces from a Trojan-infected chip using off-the-shelf instruments such as oscilloscopes [27], to deduce the weights by comparing it to a library of reference power traces obtained offline (described below). It is preferred that the attacker use a high sampling rate ($\geq$ 10GS/s) to capture sufficient sampling points within a read cycle. Once the conductance values associated with the

currently functioning ADC are recovered, the attacker sends a reset signal using the same malicious image that activates the Trojan (toggle trigger), putting all ADCs back in working order. A different activation code is needed to target a different ADC. The attacker applies the above procedure repeatedly to cycle through all ADCs to recover all synaptic weights.

Offline Characterization. The offline characterization step (highlighted in Fig. 9.4 dashed line) allows the attacker to build a library of reference traces used for comparison during online weight recovery. This is possible since: (1) the operational states of an ADC are finite, and (2) each conductance value generates a unique ADC output spiking pattern, allowing the attacker to thoroughly sweep through an ADC's current resolution steps and collect a library of distinct power traces. The development of such a characterization portfolio involves the generation of a step response chart (ADC step resolution graph in Fig. 9.4) that allows the attacker to map a unique spike pattern to a deterministic range of input current values during the Trojan embedding phase. Next, for each distinct ADC input current, its frequency domain signature (FFT) is extracted, which is used to approximate the synaptic weights.

9.5.3 Establishing Power-to-Weight Correlation

Signal Processing (FFT). The attacker can infer the ADC input current based on the decomposed frequency components of the power trace, as the frequency domain allows for a significantly higher fidelity comparison than the time domain. As a result, a Fast Fourier transform (FFT) is performed on both the victim and reference traces to compare and identify key frequency components, increasing the visibility of individual sub-components within the trace, thereby isolating unique signatures. The strongest signals within the spectrum can be attributed to the static (DC) energy costs, clock tree consumption, and spiking signature. The FFT analysis reveals that, (1) larger the input current, higher the frequency signature, and (2) each current step resolution emits a unique frequency signature. This allows the attacker to examine the frequencies extracted from the victim traces and match them against the signature frequencies within the library.

Synaptic Weights Recovery. There are two factors that determine the precision of recovered weights. First, the ADC can only respond to a set of discrete current ranges (i.e., ADC resolution steps) rather than continuous current values. This means two conductances (i.e., weights) with small differences could produce similar currents that lead to a similar switching activity (ADC power traces). The larger the step count, the more "sensitive" the ADC is to differentiate between input currents, and thus increase the resolution of the stolen weights. Seconds, ADCs are typically calibrated to work with the cumulative current produced by multiple eNVM cells. The current generated from one eNVM cell might be too small to excite the ADCs

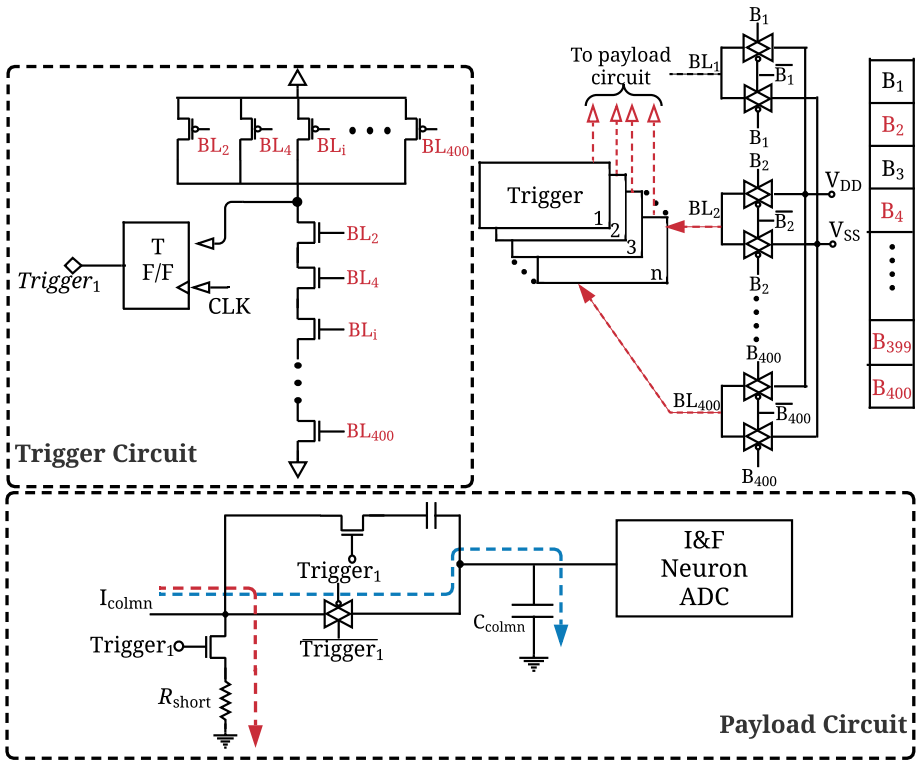


Figure 9.5: **Trojan trigger module and payload circuit**

thermometer circuits, as a result of which the power trace might not yield meaningful leakage information for the attacker to extract weights. We assign minimal conductance values to those that are not recoverable through the power side channel. While this results in a small loss in the overall model inference accuracy, in some cases, it results in the cloned network outperforming the original (Sec. 9.8.2).

9.6 Trojan Design

The suppression of the neuron is achieved through the design of an analog Trojan that consists of a trigger and a payload module. The trigger circuit determines the operating condition of the payload, i.e., if the trigger state is high, the payload is active. If the payload is activated, the neuron circuit is suppressed through bypassing the current generated by the synaptic array away from its signal path, thereby depriving it of a valid input. Fig. 9.5, shows the circuit for a switched leakage short-circuit path (highlighted in red), that deviates the current flow from its normal path (highlighted in blue). As long as the trigger state is high, the cumulative current leaks through this path and prevents any switching activity. The possibility of current leakage creeping into the neuron ADC is prevented using a DC blocking capacitor C_{DC} and the average sizing of the transistors ensures minimal charge leakage.

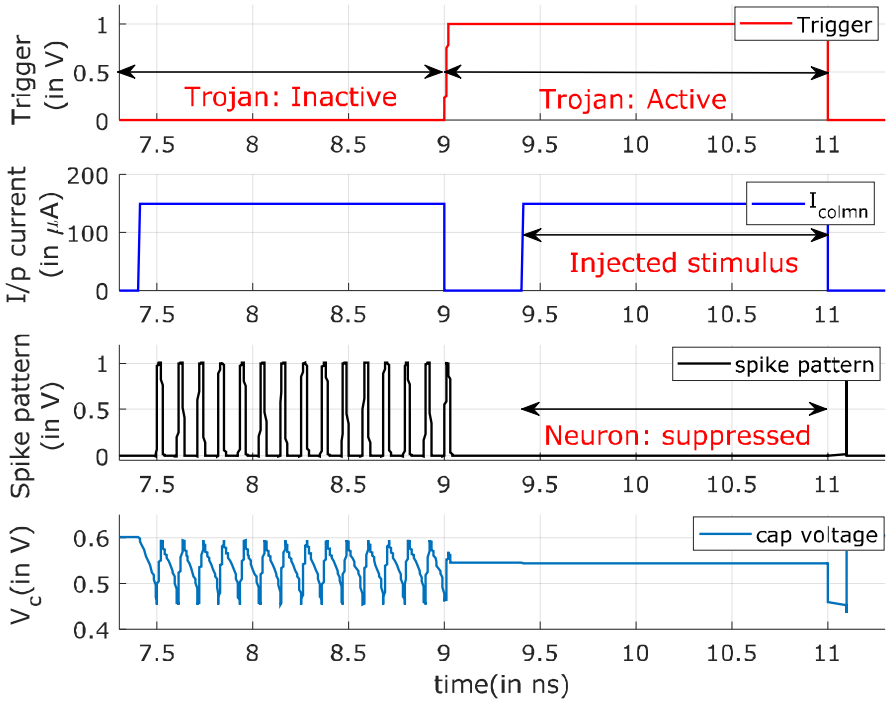


Figure 9.6: **Transient waveforms of payload circuit.**

The functioning of the payload can be observed by analyzing the transient signal characteristics of the neuron ADC between two trigger states (see Fig. 9.6). When the Trojan is inactive (shown as *Trojan:Inactive*), the cumulative input current triggers a train of spikes. However, when the Trojan is active (shown as *Trojan:Active*) and an input current stimulus is provided, it can be seen that the neuron is *"suppressed"*. The deviation of current by the payload element can be confirmed by analyzing the build-up of potential on the capacitor, V_c. Hardware Trojans typically use combinational or sequential elements to monitor internal states within a system and trigger a payload based on a predefined condition [75]. Since the attacker is capable of sending specific input image vectors, combinational logic can be used, where the input vector contains a unique combination of pixels to activate a trigger circuit. We prefer complementary pull-up and pull-down network (PUN/PDN)-based combinational circuit over a standard cell-based logic tree circuit, to enable microscopic design corruption. Fig. 9.5 depicts the schematic diagram of the trigger circuit that implements a NAND function that directly taps the inputs from the BL switch matrix and the inclusion of a Toggle flip-flop allows for the state of the trigger circuit to switch between the two operational conditions.

9.7 Methodology

Due to the lack of openly available commercial eNVM neuromorphic implementations, we customize a high-fidelity simulation environment representative of several recently published designs (Fig. 9.1), using

Neurosim [8]. A 3-layer MLP (400-neuron input layer, followed by a 100-neuron hidden layer, and a 10-neuron output layer corresponding to 10 digits) is mapped to such architecture for training using 60,000 black-and-white images in 125 epochs until its inference accuracy stabilizes (∼93%). We then collect a set of output currents making up the dot-product operations by opening different rows. Neurosim faithfully models an analog synaptic device with many non-ideal device properties such as variations within Long-Term Potentiation/Depression (LTP/LTD), cycle-to-cycle conductance variation, and, spatial variations across a memory array. This ensures that a realistic weight-to-conductance mapping is implemented, and the generated current traces encompasses both temporal and spatial variations. Note that, since this is an initial foray into this field, we limit the scope of this work to target eNVM accelerators with limited hyperparameter reconfigurability (e.g., number of layers and dimension of each layer). We leave the secret extraction of more complex models for future work. However, the key insight of this work, namely that the spiking activity of the neuron ADC can leak sensitive model parameters, is expected to hold for other architectures.

The overall neuron microarchitecture is designed and evaluated in Cadence Virtuoso and Calibre tool using the TSMC 65nm Low Power(LP) flavor PDK. Transistor-level simulations are carried out to generate power traces and other relevant transient signals that allow for generating the offline characterization power traces, as well as mimicking conditions for triggering the payload. Process, Voltage, and Temperature (PVT) variations within the neuron ADC design are considered during the *offline characterization* phase of the attack. These variations result in a deviation in the step response mapping, which is visualized as a mismatch in step width and height in comparison to ideal characteristics (Fig.9.7b). We inject stochastic noise sources that replicate the average switching activities that potentially occur in a co-processor[311].

The switching transients are monitored over the power rail trace, and by denoising the baseline power trace from the monitored signals, a higher SNR trace can be deduced upon which the FFT transform is applied. The trace is sampled at a 100ps sampling interval and a 4096 FFT bin trace is generated, which translates to approximately a 250 kHz resolution. To build the characterization portfolio, the dynamic input current range is generated based on the parameters of selected eNVM characteristics. The most prominent large signal frequency bins from each step of the sweep are collected and assigned to an input current value. The relevance of their magnitude can be deduced from their unique frequency signatures, as they share a similar spectral magnitude characteristic.

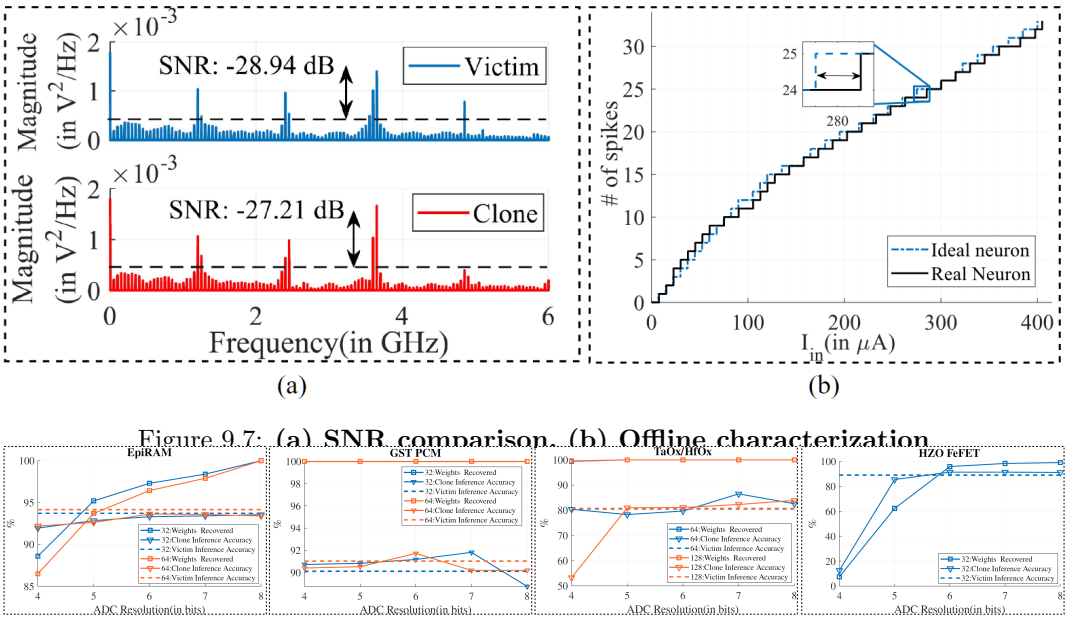


Figure 9.7: **(a) SNR comparison. (b) Offline characterization.**

Figure 9.8: **Sensitivity to Conductance Levels and ADC Resolutions for add references for devices.**

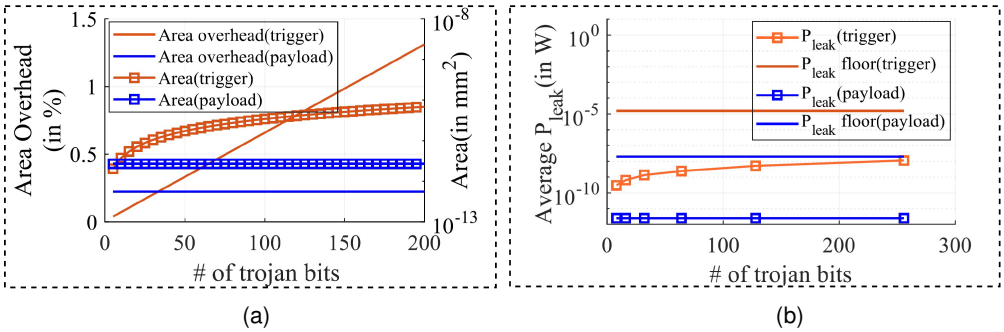


Figure 9.9: **(a). Area overhead and (b).** P_{leak} **in comparison to the noise floor, as a function of the input Trojan vector.**

9.8 Results

9.8.1 Trojan Stealth

Design and Verification Time Detection. To remain stealthy, it is imperative that the Power Spectral Density (PSD) of the Trojan-infected malicious unit under normal operating conditions must not significantly deviate from the average PSD of unaffected hardware. The PSD of unaffected hardware (*victim*) is visualized by extracting the switching current trace from the power rail across 400 read cycles under normal operating conditions. The resulting spectrogram is then compared with that of a Trojan-infected (*clone*) malicious unit, where the payload circuits are deactivated to mimic normal operating conditions. As seen in Fig. 9.7 the Signal-to-Noise Ratio (SNR) deviation is under 1.75 dB, with both spectrograms exhibiting a similar fingerprint.

To ensure the undetectability of the trigger and payload circuits through test pattern generation techniques, we stipulate that the area overhead is under a margin of 0.5% [312]. The area overhead of the trigger element is a function of the synaptic core area. The number of input bits that map to the payload element controls the length of the Trojan code and hence the size of the PUN/PDN network. Every neuron ADC must be embedded with a payload element, resulting in a fixed overhead. A similar trend can be observed with the average leakage power dissipated by the Trojan (Fig. 9.9a and Fig. 9.9b).

While our ability to evaluate against verification-time detection frameworks is limited, as they are not open source, we offer a qualitative discussion. Frameworks such as FANCI [313] that operate at the RTL level would not be able to detect our Trojan, owing to its form factor and its ability to embed the Trojan at the GDS-II levels and polygon pattern etching foundry stages. Methods such as UCI [314] mainly apply for digital Trojans, which rely on analyzing switching activity.

Run-time Detection. By exploiting the input vector to encode the necessary bits, it is possible to generate an extremely large set of combinations for the trigger code (there are 2^{400} possible codes that can be uniquely assigned to each ADC). Our analysis shows that a 50-75 bit long trigger code results in a false activation of a single Trojan, only once in 1000 random input test patterns. Furthermore, a trigger code that is at least 35 bits long can ensure the prevention of two simultaneous false activations of Trojans across 1000 input test patterns, significantly enhancing our ability to evade run-time detection.

9.8.2 Sensitivity Study

We sweep the ADC resolution from 8 to 256 steps and vary the device conductance levels, thereby simulating a wide array of neuromorphic design choices. Fig. 9.8 shows the inference accuracy of the original (victim) model, the cloned model using the stolen weights, and the percentage of weights recovered by the attacker for accelerators implemented using multiple eNVM technologies (EpiRAM(Ag:SiGe), HZO FeFET, TaOx/HfOx , and GST PCM). We draw four major conclusions. **First**, regardless of the underlying eNVM technology, we are able to recover more than 90% of the weights. The remainder of the weights do not build a sufficient input impulse to generate a spike train. **Second**, as the ADC resolution improves, more weights can be recovered, because the ADC resolution becomes more sensitive to the small current generated by a single eNVM cell. The overall weight extraction of our attack improves from 94.4% $\rightarrow$ 97.8% and 64.1% $\rightarrow$ 97.1% for a 2-bit improvement in ADC resolution, when the hardware is simulated in EpiRAM and HZO FeFET, respectively. The greater improvement in the case of HZO FeFET is attributed to the larger conductance density characteristics offered by the device. **Third**, in many cases, even an ADC with a lower resolution, poses a serious threat, as the attacker can reliably clone a model with comparable performance. For instance, when evaluating the attack strategy for a 5-bit resolution ADC, the worst case performance delta between the

original and recovered inference accuracy across the four memory flavors is under 2.65%. **Fourth**, a higher percentage of weights recovered by the attacker does not always translate to higher inference accuracy of the cloned model. This is because ADCs are calibrated to segment continuous current ranges to resolution steps and the current generated from a single cell may be mapped to a current value that is slightly off compared to the true current. In some cases, we're able to obtain a cloned network with higher accuracy than the victim model. We suspect that this is because the weights of the victim models are sometimes stuck at local minimums.

9.9 Mitigation

Trojan detection. Several techniques have been proposed to prevent the insertion of a hardware Trojan into ICs. Waksman et al. [313] propose FANCI, a framework for profiling activities of wires inside a chip, and flagging nearly unused ones as possible Trojan paths. Their insight stems from the fact that Trojan functionalities are mostly dormant until triggered by external malicious inputs. This can potentially catch the Trojan logic embedded in an eNVM device. However, FANCI, as described in the paper, examines the hardware implementation at the RTL level, such as a netlist file, while the Trojan we describe can also be placed inside a layout GDSII file, thereby circumventing it. Extending detection frameworks to enable more comprehensive detection is interesting future work.

Trojan insertion prevention. To prevent insertion at the layout level, a potential countermeasure that can be used is layout masking [315]. However, this is expected to prohibitively increase the power and area overhead (by $\sim 10\%$). If an eNVM device is deployed as an IoT or a wearable device that is power/area-constrained, layout masking may not be ideal. The weight recovery attack may also be defeated by integrating an ultra-low resolution ADC with four or eight resolution steps, preventing the successful extraction of most of the weights. However, this would severely limit the capabilities of such devices to scale to larger workloads.

Side-channel prevention. Masking the signal (EM, power, thermal, etc.) signature, therefore, preventing the side-channel attacks (SCA), usually requires dedicated SCA countermeasure hardware which can be impractical (power and area overhead) to integrate for neuromorphic devices.

9.10 Conclusion

We explore the feasibility of a novel supply-chain threat against eNVM neuromorphic devices. We identify ADC as the key element that exposes a vulnerability within the neuron core and further deduce the weights

by analyzing and isolating the switching activities of the ADC. We also design a stealthy hardware Trojan that allows the attacker to correlate the transient system power consumption to the synaptic weight and subsequently reconstruct a cloned model with high fidelity.

Chapter 10

Conclusions and Future Work

10.1 Conclusions

For data-intensive tasks, data movement dominates computation. We hypothesize that near-data-processing techniques that enable computation as close as possible to where data resides (e.g., DRAM or SSD) can be optimal solutions to address the performance bottleneck caused by data movement in various applications. Additionally, the NDP approach offers much higher data bandwidth and computational throughput than traditional Von Neumann architectures. This dissertation explores design spaces of various NDP techniques and architectures that accelerate application execution and save energy. We have made the following contributions to this dissertation.

First, we examine a series of digital DRAM-based bit-serial SIMD-style processing architectures targeting bioinformatics (Chapter 3 Sieve), exact-pattern matching (Chapter 4 DRAM-CAM), OLAP (Chapter 5 Membrane), and general-purpose computing (Chapter 6 DRAM-BitSIMD).

In Chapter 3, we present Sieve, a set of DRAM-based in-memory architectures to accelerate k-mer matching by storing reference k-mer patterns along the bitlines and enhancing row buffers with a minimal set of Boolean logic for k-mer matching. To explore optimal Sieve designs, we compare the placement of custom k-mer matching logic at three different levels in the DRAM hierarchy: from the I/O interface of the DRAM chips (Sieve Type-1) to the local row buffer of each subarray (Sieve Type-3), and Type-2 as the middle ground where several subarrays share one matcher. In this work, we devise a novel data layout, an indexing scheme, and an early termination mechanism that synergistically provides 1.01X/55.49X/404.48X speedup and 3.80X/28.11X/55.89X energy saving over the state-of-the-art CPU baseline. Compared to GPU, Type-1 is 3X to 5X slower than the GPU but more energy efficient, and Type-2 is only modestly faster (2.59x

to 9.43x), but Type-3 offers speedups of 33.13X–55.0X and energy savings of 83.77X–141.15X, showing the effectiveness of in-situ PIM processing, which is an aggressive form of NDP. **This is the first work** to introduce and showcase the effectiveness of a column-wise data placement for k-mer matching with early termination, substantially advancing the state-of-the-art in both throughput and efficiency.

In Chapter 4, we extend Sieve Type-3, which leverages subarray-level parallelism, with logic integrated into each local row buffer. DRAM-CAM retains the core architectural designs of Sieve and serves as a PCI-attached accelerator with an offload model. We introduce several new hardware components (population count logic and a hardware data transposition unit) and runtime optimizations (chip-level parallelism, pattern distribution, and pattern replication) to increase functionality and boost performance. DRAM-CAM can accelerate a wide range of pattern-matching tasks and offers impressive performance benefits (6217X speedup and 5888X energy savings over CPU).

In Chapter 5, we further investigated an alternative use of Sieve Type-3, which is for high-throughput database scan operation, which often dominates the execution time in OLAP query processing. To this end, we design a minimalist custom logic embedded at the DRAM subarray levels to enable high-throughput in-situ table scans, dubbed Membrane-V. The integrated logic incurs a small area and power overhead compared to a commodity DRAM chip, and supports ranged query comparison, which is more complicated than exact pattern matching enabled in Sieve. We designed a non-intrusive co-processing scheme to integrate Membrane into a DBMS to leverage its efficient predicate filtering potential fully. Specifically, Membrane-V first returns a bitmask indicating which database records satisfy the conditions in the filter predicate, and the host then "pulls" data from the DRAM based on the bitmask. Membrane-V offers 1.26×/25.97×/5.94× min/max/geomean speedup compared to the CPU. Notably absent from the existing efforts in this domain is a comprehensive consideration of hardware options (e.g., digital vs. analog) and software implications (e.g., vertical VS. horizontal data layout). Moreover, prior research has rarely thoroughly examined end-to-end query performance across a full benchmark. **This work makes significant strides** in PIM-based data analytics accelerator, including comprehensive PIM architecture design space exploration, workload distribution between CPU host and PIM accelerator, a WideTable pre-processing technique that pairs well with PIM framework, and a Rank-level hardware unit that removes the data retrieval/selection bottleneck.

In Chapter 6, inspired by the performance of bit-serial DRAM processing for pattern matching, we explored the design space of a general-purpose bit-serial DRAM-based PIM architecture. The key idea to enable in-DRAM bit-serial computing is to treat each bitline as a vector lane and align the source and destination data elements vertically on top of each other. A series of subarray row activations perform the computation sequentially at each bit position. The vertical layout allows each activation to access a bit slice across a row of vector elements (i.e., bitlines or lanes). Two additional advantages of the vertical layout

are that it enables arbitrary bit access within the operands (e.g., left or right shifting within each word is cheap) and it supports flexible operand size without having a word spread across multiple chips. Many prior architectures leverage DRAM's analog property by connecting three DRAM rows to the sense amplifiers, AKA triple-row-activation (TRA), to force charge-sharing at the row buffer, equivalent to performing a row-wide bitwise logical operation. However, analog-based bit-serial DRAM computing has the disadvantages of high latency and energy overhead. We show that our performance-optimized design bit-serial architecture outperforms the CPU by 20X, GPU by 5X, and SIMDRAM (prior art) by 1.7X and is substantially more energy- and area-efficient. **This work** explores the complex design space of digital bit-serial PIM, which is not comprehensively analyzed in prior work. We also propose and discuss for the first time a new demarcation of designing and evaluating PIM accelerator, namely accelerator-first vs. memory-first. Our evaluation also shows that the analog-based designs popularized in prior work are less energy- and area-efficient than their digital counterparts. This work also discussed various system integration challenges and solutions, which opens up future research opportunities.

Second, in Chapter 7, we present a different NDP choice based on a 3D-stacked memory cube for de Bruijn graph acceleration. The proposed NDP architecture consists of multiple Hybrid Memory Cubes (HMC), and each HMC connects to the others using an inter-cube network [22, 23]. Each cube's memory is divided into several vertical memory vaults, and each vault is coupled with an integrated processing core connected to a memory controller for local vault access. We can schedule parallel applications on NDP systems by exploiting massive NDP cores (small CPU processors at the logic layer of HMC). NDP system supports remote function calls based on message passing to handle inter-core communication without expensive coherence management. Our evaluation shows that the proposed.NDP implementation can improve the performance of graph construction by 33× and traversal by 16× compared to the state-of-the-art. **This is the first work** that tackles in-memory accelerator for DBG-based *de novo* genome assembly.

Third, in Chapter 8, we step away from the memory technology and investigate the potential of processing-with-storage-technology (PWST). PWST can fundamentally solve the bottleneck caused by data movement issues. In this work, We design an architecture Abakus leveraging 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. Through a set of domain-specific hardware extensions to accelerate the key operations for k-mer counting at various levels of the SSD hierarchy, Abakus can achieve 9.9×, 8.2×, and 3.3× speedup over the CPU-, GPU-, and PIM NDP solutions. Unlike prior work, which does not consider the I/O bottleneck, **this work** for the first time leverages PWST to propose novel and scalable accelerator designs to eliminate the I/O overheads, improving the performance end-to-end.

Finally, in Chapter 9, we contribute to NDP security analysis by demonstrating **for the first time** the feasibility of carrying out a hardware supply chain attack against a neuromorphic DNN accelerator that performs neuron computation inside resistive memory cell arrays. The crux of this attack is the design and stealthy placement of a neuron-suppressing hardware Trojan that a colluding adversary can reliably trigger. We also design the Trojan such that the attacker can correlate the transient system power consumption to an eNVM cell conductance (i.e., synaptic weight). There are two major motivations for a weight-stealing attack. First, the synaptic weights of a neural network are considered core IP. Second, stealing weights is increasingly more economical than training. To obtain a model with competitive performance, typically, a large set of high-quality labeled data and proprietary training algorithms are required [303]. Further, even with access to proprietary training data, the training process can take weeks and will likely worsen with model sizes, affecting the time-to-market [304]. Our evaluation suggests an adversary can stealthily recover 90% model parameters while evading detection, highlighting the dire need for future NDP design with security in mind.

10.2 Future Research Opportunities

This dissertation opens a few future research directions.

- **System-level support:** To ease the adoption friction, NDP architectures need more system-level support. For example, to integrate a PIM-enabled memory into an existing system, a new data allocation routine and a new address interleaving are needed. To name a few challenges, the PIM-aware data allocation routine needs to track available physical memory rows and columns for PIM data structures allocation, gather data from the host memory region to the PIM-eligible region, and transpose the data between vertical format and horizontal format if necessary. Developing a working allocation scheme means tapping into OS layers and making non-trivial changes. In traditional address interleaving, consecutive words are spread out in different physical chips to maximize data I/O, but PIM processes prefer data to be physically adjacent. Figuring out an optimal address interleaving is a complex undertaking. It would require researchers to generate many performance profiles of PIM processes running in a realistic system with a mixture of different co-running workloads.
- **Bit-serial architecture with a horizontal data layout:** While the vertically laid out data format supports massive parallelism and shows excellent kernel-level performance improvement, it has the disadvantages of data transposition, which incurs extra data movement cost (relaying out data) and a myriad of system-integration difficulties. A horizontal layout data might avoid several system-level difficulties with slightly worse performance. We leave this as future work.

- **Compiler and programming model:** Adopting a new architecture is highly influenced by its programmability and ease of execution. The performance benefit of NDP is well understood, but the corresponding APIs and library routines are lacking. There is also limited study on the compilation support for NDP. A working compilation toolchain and expressive API library routines must be developed. Currently, we manually select high-level operations (e.g., vector arithmetic) from application code, rewrite them using intrinsics/macros that an NDP system can execute, and convert them to a series of micro-ops for performance and functional evaluation. In the future, a non-expert developer can express the application logic using traditional C/C++ style code with optional sets of pragmas to surround core loops, and the NDP compiler automatically converts the high-level algorithm into vector-style intermediate representations (IRs). The sequence of NDP IRs is then offloaded to the targeted NDP hardware to be broken down into micro codes and executed.
- **Mitigation for supply-chain Trojan attack:** As we explained in Chapter 2.4, malicious circuits can be implanted at many places in the supply chain during the IC life cycle, posing a serious threat to the deployment of NDP devices. We have discussed a few conventional mitigation solutions in Chapter 9.9 to detect and prevent the insertion of a hardware Trojan. However, NDP devices introduce a unique set of challenges that can render prior hardware Trojan countermeasures ineffective. Due to the sensitive deployment scenarios for NDP devices, defeating the supply-chain hardware Trojan attack is becoming increasingly important. It has garnered much interest from the industry and various government agencies. As we have demonstrated the possibility of stealing sensitive information from a recently proposed NDP architecture, developing effective mitigation solutions is of great importance.

10.3 Appendix

10.3.1 Accepted publications

- Sieve: Scalable In-situ DRAM-based Accelerator Designs for Massively Parallel k-mer Matching
  Lingxi Wu, Rasool Sharifi, Marzieh Lenjani, Kevin Skadron, Ashish Venkat
  ISCA 2021
- Ultra Efficient Acceleration for *De Novo* Genome Assembly via Near-Memory Computing
  Minxuan Zhou, **Lingxi Wu (Joint 1st author)**, Muzhou Li, Niema Moshiri, Kevin Skadron, Tajana Rosing
  PACT 2021
- DRAM-CAM: General-Purpose Bit-Serial Exact Pattern Matching
  Lingxi Wu, Rasool Sharifi, Ashish Venkat, Kevin Skadron
  Computer Architecture Letters 2022
- Hardware Trojans in eNVM Neuromorphic Devices
  Lingxi Wu, Rahul Sreekumar (Joint 1st author), Rasool Sharifi, Mircea Stan, Kevin Skadron, Ashish Venkat
  DATE 2023
  (Best Paper Nominee)

10.3.2 Under Review

- Membrane: A PIM-based Architecture to Accelerate Database OLAP Queries
  Akhil Shekar, **Lingxi Wu (Joint 1st author)**, Kevin P. Gaffney, Helena Caminal, Martin Prammer, Yimin Gao, Ashish Venkat, Mircea Stan, José Martínez, Jignesh M. Patel, Kevin Skadron
  HPCA 2023
- Abakus: Accelerating k-mer Counting With Storage Technology
  Lingxi Wu, Minxuan Zhou (Joint 1st author), Weihong Xu, Ashish Venkat, Tajana Rosing, Kevin Skadron
  TACO 2023
- DRAM-BitSIMD: DRAM-based Bit-Serial Vector Computing Architecture.
  Deyuan Guo, **Lingxi Wu (Joint 1st author)**, Farzana Siddique, Ashish Venkat, Kevin Skadron
  ASPLOS 2023

Bibliography

[1] Derrick E Wood and Steven L Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15, 2014.

[2] Timothy J. Close Rachid Ounit, Steve Wanamaker and Stefano Lonardi. Scalable metagenomic taxonomy classification using a reference genome database. *BMC Genomics*, 16, 2015.

[3] I King Jordan Anuj Gupta and Lavanya Rishishwar. stringMLST: a fast k-mer based tool for multilocus sequence typing. *Bioinformatics*, 2017.

[4] Daniel Navarro-Gomez, Jeremy Leipzig, Lishuang Shen, Marie Lott, Alphons PM Stassen, Douglas C Wallace, Janey L Wiggs, Marni J Falk, Mannis van Oven, and Xiaowu Gai. Phy-Mer: a novel alignment-free and reference-independent mitochondrial haplogroup classifier. *Bioinformatics*, 31(8), 2015.

[5] Sasha K. Ames, David A. Hysom, Shea N. Gardner, G. Scott Lloyd, Maya B. Gokhale, and Jonathan E. Allen. Scalable metagenomic taxonomy classification using a reference genome database. *Bioinformatics*, 29(18), 2013.

[6] Stephen Altschul, Warren Gish, Webb Miller, Eugene Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[7] Marius Erbert, Steffen Rechner, and Matthias Müller-Hannemann. Gerbil: A fast and memory-efficient k-mer counter with gpu-support. *Algorithms for Molecular Biology*, 12(1), 2017.

[8] Pai-Yu Chen, Xiaochen Peng, and Shimeng Yu. Neurosim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(12):3067–3080, 2018.

[9] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *ISCA*, 2016.

[10] GRAND VIEW RESEARCH. 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, 2017.

[11] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. Profiling hyperscale big data processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[12] Lingxi Wu, Rasool Sharifi, Marzieh Lenjani, Kevin Skadron, and Ashish Venkat. Sieve: Scalable in-situ DRAM-based accelerator designs for massively parallel k-mer matching. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 251–264. IEEE, 2021.

[13] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 1995.

[14] Tien-Ju Yang, Yu-Hsin Chen, Joel Emer, and Vivienne Sze. A method to estimate the energy consumption of deep neural networks. *2017 51st Asilomar Conference on Signals, Systems, and Computers*, 2017.

[15] Biresh Kumar Joardar, Priyanka Ghosh, Partha Pratim Pande, Ananth Kalyanaraman, and Sriram Krishnamoorthy. 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, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Debendra Das Sharma. Compute Express Link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 5–12. IEEE, 2022.

[17] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K.Choi. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *ISCA*, 2015.

[18] Tim Finkbeiner, Glen Hush, Troy Larsen, Perry Lea, John Leidel, and Troy Manning. In-memory intelligence. *IEEE Micro*, 37(4):30–38, 2017.

[19] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 14–26, 2016.

[20] Karl Freund. Ibm research says analog ai will be 100x more efficient. yes, 100x., Sep 2021.

[21] Lixue Xia, Boxun Li, Tianqi Tang, Peng Gu, Xiling Yin, Wenqin Huangfu, Pai-Yu Chen, Shimeng Yu, Yu Cao, Yu Wang, Yuan Xie, and Huazhong Yang. Mnsim: Simulation platform for memristor-based neuromorphic computing system. In *2016 Design, Automation and Test in Europe Conference Exhibition (DATE)*, pages 469–474, 2016.

[22] Miao Hu, Hai Li, Yiran Chen, Qing Wu, Garrett S. Rose, and Richard W. Linderman. Memristor crossbar-based neuromorphic computing system: A case study. *IEEE Transactions on Neural Networks and Learning Systems*, 25(10):1864–1878, 2014.

[23] Xiaoxiao Liu, Mengjie Mao, Beiye Liu, Boxun Li, Yu Wang, Hao Jiang, Mark Barnell, Qing Wu, Jianhua Yang, Hai Li, and Yiran Chen. Harmonica: A framework of heterogeneous computing systems with memristor-based neuromorphic computing accelerators. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 63(5):617–628, 2016.

[24] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–39, 2016.

[25] M. Prezioso, F. Merrikh-Bayat, B. D. Hoskins, G. C. Adam, K. K. Likharev, and D. B. Strukov. Training and operation of an integrated neuromorphic network based on metal-oxide memristors. *Nature*, 521(7550):61–64, 2015.

[26] Accelerating edge ai.

[27] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. A2: Analog malicious hardware. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 18–37, 2016.

[28] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design & Test of Computers*, 27, 2010.

[29] Shivam Bhasin and Francesco Regazzoni. A survey on hardware trojan detection techniques. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2021–2024, 2015.

[30] Yier Jin, Nathan Kupp, and Yiorgos Makris. Experiences in hardware trojan design and implementation. In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 50–57, 2009.

[31] Lang Lin, Wayne P. Burleson, and Christof Paar. Moles: Malicious off-chip leakage enabled by side-channels. *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*, pages 117–122, 2009.

[32] Lang Lin, Markus Kasper, Tim Güneysu, Christof Paar, and Wayne Burleson. Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 382–395, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[33] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 273–287, 2017.

[34] Nastaran Hajinazar, Geraldo F Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. SIMDRAM: A framework for bit-serial SIMD processing using DRAM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–345, 2021.

[35] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. ComputeDRAM: In-memory compute using off-the-shelf DRAMs. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pages 100–113, 2019.

[36] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. DRISA: A DRAM-based reconfigurable in-situ accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 288–301, 2017.

[37] Mustafa F Ali, Akhilesh Jaiswal, and Kaushik Roy. In-memory low-cost bit-serial addition using commodity DRAM technology. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(1):155–165, 2019.

[38] Xin Xin, Youtao Zhang, and Jun Yang. ELP2IM: Efficient and low power bitwise operation processing in DRAM. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 303–314. IEEE, 2020.

[39] Lingxi Wu, Rasool Sharifi, Ashish Venkat, and Kevin Skadron. Dram-cam: General-purpose bit-serial exact pattern matching. *IEEE Computer Architecture Letters*, 21(2):89–92, 2022.

[40] Minxuan Zhou, Lingxi Wu, Muzhou Li, Niema Moshiri, Kevin Skadron, and Tajana Rosing. Ultra efficient acceleration for de novo genome assembly via near-memory computing. In *PACT*, 2021.

[41] Lingxi Wu, Rahul Sreekumar, Rasool Sharifi, Kevin Skadron, Mircea R. Stant, and Ashish Venkat. Hardware trojans in envm neuromorphic devices. In *2023 Design, Automation and Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2023.

[42] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A case for exploiting subarray-level parallelism (SALP) in DRAM. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 368–379, 2012.

[43] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W. Keckler. Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 204–216. IEEE Press, 2016.

[44] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally. Fine-grained dram: Energy-efficient dram for extreme bandwidth systems. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 41–54, 2017.

[45] Ramyad Hadidi, Bahar Asgari, Burhan Ahmad Mudassar, Saibal Mukhopadhyay, Sudhakar Yalamanchili, and Hyesoon Kim. Demystifying the characteristics of 3d-stacked memories: A case study for hybrid memory cube. In *Proceedings of the IEEE International Symposium on Workload Characterization* , pages 66–75, 2017.

[46] Micron. Hybrid memory cube specification 2.1. http://hybridmemorycube.org/specification-v2-download/.

[47] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R. Stan, and Kevin Skadron. Fulcrum: A Simplified Control and Access Mechanism Toward Flexible and Practical In-Situ Accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 556–569, February 2020.

[48] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian. Graphp: Reducing communication for pim-based graph processing with efficient data partition. In *HPCA*, 2018.

[49] Seth Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads.

[50] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. The Mondrian Data Engine. In *ISCA*, 2017.

[51] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 57–70, USA, 2008. USENIX Association.

[52] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQSim: A framework for enabling realistic studies of modern Multi-Queue SSD devices. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 49–66, Oakland, CA, February 2018. USENIX Association.

[53] Lingxi Wu et al. Sieve: Scalable in-situ dram-based accelerator designs for massively parallel k-mer matching. In *ISCA*, 2021.

[54] Jonas Almeida Andrzej Zielezinski, Susana Vinga and Wojciech M. Karlowski. Alignment-free sequence comparison: benefits, applications, and tools. *Genome Biology*, 18, 2017.

[55] Karen L. Adair Stinus Lindgreen and Paul P. Gardner. An evaluation of the accuracy and speed of metagenome analysis tools. *Scientific Reports*, 6, 2016.

[56] Dna sequencing costs: Data. https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data.

[57] Steven Flygare, Keith Simmon, Chase Miller, Yi Qiao, Brett Kennedy, Tonya Di Sera, Erin H Graf, Keith D Tardif, Aurélie Kapusta, Shawn Rynearson, et al. Taxonomer: an interactive metagenomics analysis portal for universal pathogen detection and host mRNA expression profiling. *Genome Biology*, 17, 2016.

[58] Smithsonian Magazine. Ambitious project to sequence genomes of 1.5 million species kicks off, Nov 2018.

[59] Minxuan Zhou, Lingxi Wu, Muzhou Li, Niema Moshiri, Kevin Skadron, and Tajana Rosing. Ultra efficient acceleration for de novo genome assembly via near-memory computing. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 199–212, 2021.

[60] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the national academy of sciences*, 98(17):9748–9753, 2001.

[61] 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. Rapid snp discovery and genetic mapping using sequenced rad markers. *PloS one*, 3(10):e3376, 2008.

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

[63] Elaine R Mardis. The impact of next-generation sequencing technology on genetics. *Trends in genetics*, 24(3):133–141, 2008.

[64] Jay Shendure and Hanlee Ji. Next-generation dna sequencing. *Nature biotechnology*, 26(10):1135–1145, 2008.

[65] Can Alkan, Jeffrey M Kidd, Tomas Marques-Bonet, Gozde Aksay, Francesca Antonacci, Fereydoun Hormozdiari, Jacob O Kitzman, Carl Baker, Maika Malig, Onur Mutlu, S Cenk Sahinalp, Richard A Gibbs, and Evan E Eichler. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature genetics*, 41(10):1061, 2009.

[66] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the national academy of sciences*, 98(17):9748–9753, 2001.

[67] JT Simpson, K Wong, SD Jackman, JE Schein, SJ Jones, and I. Birol. Abyss: a parallel assembler for short read sequence data. *Genome Res*, 2009.

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

[69] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. Parallelized short read assembly of large genomes using de bruijn graphs. *BMC Bioinformatics*, 12(1), 2011.

[70] Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. Megahit: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph. *Bioinformatics*, 31(10):1674–1676, 2015.

[71] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.

[72] Mark Chaisson, Pavel Pevzner, and Haixu Tang. Fragment assembly with short reads. *Bioinformatics*, 20(13):2067–2074, 2004.

[73] Anja Bog, Kai Sachs, and Alexander Zeier. Benchmarking database design for mixed OLTP and OLAP workloads. In *Proceeding of the Second Joint WOSP/SIPEW International Conference on Performance Engineering (ICPE)*, page 417, 2011.

[74] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, March 1997.

[75] M. Tehranipoor, J. Plusquellic, R. M. Rad, and X. Wang. Power supply signal calibration techniques for improving detection resolution to hardware trojans. In *Computer-Aided Design, International Conference on*, pages 632–639, Los Alamitos, CA, USA, nov 2008. IEEE Computer Society.

[76] Xiaoxiao Wang, Hassan Salmani, Mark Mohammad Tehranipoor, and James F. Plusquellic. Hardware trojan detection and isolation using current integration and localized current analysis. *2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, pages 87–95, 2008.

[77] You Li, Tayla B Heavican, Neetha N Vellichirammal, Javeed Iqbal, and Chittibabu Guda. ChimeRScope: a novel alignment-free algorithm for fusion transcript prediction using paired-end RNA-Seq data. *Nucleic Acids Res.*, 2017.

[78] FP Breitwieser, DN Baker, and Steven L Salzberg. KrakenUniq: confident and fast metagenomics classification using unique k-mer counts. *Genome Biology*, 19, 2018.

[79] Robert A Edwards, Robert Olson, Terry Disz, Gordon D Pusch, Veronika Vonstein, Rick Stevens, and Ross Overbeek. Real time metagenomics: using k-mers to annotate metagenomes. *Bioinformatics*, 28(24), 2012.

[80] Samuel S Minot, Niklas Krumm, and Nicholas B Greenfield. One Codex: A Sensitive and Accurate Data Platform for Genomic Microbial Identification. *BioRxiv*, 2015.

[81] Ricardo Assunção Vialle, Fábio de Oliveira Pedrosa, Vinicius Almir Weiss, Dieval Guizelini, Juliana Helena Tibaes, Jeroniza Nunes Marchaukoski, Emanuel Maltempi de Souza, and Roberto Tadeu Raittz. RAFTS3: Rapid Alignment-Free Tool for Sequence Similarity Search. *bioRxiv*, 2016.

[82] Tajana Simunic Rosing, Niema Moshiri, and Rob Knight. Acceleration of bioinformatics workloads. *DARPA ERI Summit*, Jul 2020.

[83] C Lee. Ventola. The antibiotic resistance crisis: part 1: causes and threats. *P & T : a peer-reviewed journal for formulary management*, 40(4):277–83, 2015.

[84] Anirban Nag, C. Ramachandra, Rajeev Balasubramonian, Ryan Stutsman, Edouard Giacomin, Hari Kambalasubramanyam, and Pierre-Emmanuel Gaillardon. GenCache: Leveraging In-Cache Operators for Efficient Sequence Alignment. In *MICRO*, 2019.

[85] Wenqin Huangfu, Xueqi Li, Shuangchen Li, Xing Hu, Peng Gu, , and Yuan Xie. MEDAL: Scalable DIMM based Near Data Processing Accelerator for DNA Seeding Algorithm. In *MICRO*, 2019.

[86] Wenqin Huangfu, Shuangchen Li, Xing Hu, and Yuan Xie. RADAR: A 3D-reRAM Based DNA Alignment Accelerator Architecture. In *DAC*, 2018.

[87] Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *MICRO*, 2017.

[88] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: In-memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *MICRO*, 2017.

[89] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. Near-Data Processing: Insights from a MICRO-46 Workshop. *MICRO*, 2014.

[90] R. Sharifi and Z. Navabi. Online Profiling for Cluster-Specific Variable Rate Refreshing in High-Density DRAM Systems. *ETS*, 2017.

[91] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Fast Bulk Bitwise AND and OR in DRAM. *CAL*, 2015.

[92] Arpith Jacob, Joseph Lancaster, Jeremy Buhler, and Roger Chamberlain. FPGA-accelerated seed generation in mercury BLASTP. In *FCCM*, 2007.

[93] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. Computedram: In-memory compute using off-the-shelf drams. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 100–113, 2019.

[94] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally. Architecting an Energy-Efficient DRAM System for GPUs. In *HPCA*, 2017.

[95] E. Fernandez, W. Najjar, and S. Lonardi. String Matching in Hardware Using the FM-Index. In *FCMM*, 2011.

[96] Kevin K Chang, Prashant J Nair, Donghyuk Lee, Saugata Ghose, Moinuddin K Qureshi, and Onur Mutlu. Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram. In *HPCA*, 2016.

[97] Micron. Tn-40-07: Calculating memory power for ddr4 sdram introduction. https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf, 2020.

[98] Ben Langmead Derrick E Wood, Jennifer Lu. Improved metagenomic analysis with kraken 2. *Genome Biology*, 20, 2019.

[99] Müller André Kobus Robin, Hundt Christian and Bertil Schmidt. Accelerating metagenomic read classification on CUDA-enabled GPUs. *BMC Bioinformatics*, 2009.

[100] Alif Ahmed and Kevin Skadron. Hopscotch: A Micro-benchmark Suite for Memory Performance Evaluation. In *MEMSYS*, 2019.

[101] Intel Performance Counter Monitor. https://github.com/opcm/pcm.

[102] NVIDIA Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler.

[103] A. Jahanshahi, H. Sabzi, C. Lau, and D. Wong. GPU-NEST: Characterizing Energy Efficiency of Multi-GPU Inference Servers. *CAL*, 2020.

[104] Predictive Technology Model (PTM). http://ptm.asu.edu/.

[105] FreePDK45: 45nm variant of the FreePDK Process Design Kit. https://www.eda.ncsu.edu/wiki/FreePDK45:Contents.

[106] M. R. Guthaus, J. E. Stine, S. Ataei, Brian Chen, Bin Wu, and M. Sarwar. OpenRAM: An open-source memory compiler. In *ICCAD*, 2016.

[107] Aaron Stillmaker, Zhibin Xiao, and Bevan M. Baas. Toward more accurate scaling estimates of cmos circuits from 180 nm to 22 nm. 2012.

[108] J. B. Park, W. R. Davis, and P. D. Franzon. 3D-DATE: A Circuit-Level Three-Dimensional DRAM Area, Timing, and Energy Model. *IEEE Transactions on Circuits and Systems*, 2019.

[109] PCI-E Specification. https://pcisig.com/specifications.

[110] K. Song, J. Kim, J. Yoon, S. Kim, H. Kim, H. Chung, H. Kim, K. Kim, H. Park, H. C. Kang, N. Tak, D. Park, W. Kim, Y. Lee, Y. C. Oh, G. Jin, J. Yoo, D. Park, K. Oh, C. Kim, and Y. Jun. A 31 ns Random Cycle VCAT-Based 4F ^{2} DRAM With Manufacturability and Enhanced Cell Efficiency. *IEEE Journal of Solid-State Circuits*, 2010.

[111] Micron. 4 f2 folded bit line dram cell structure having buried bit and word lines. https://patents.google.com/patent/US6689660B1/en, 2020.

[112] H. S. Stone. A Logic-in-Memory Computer. *IEEE Transactions on Computers*, C-19(1):73–78, 1970.

[113] Yi Kang, Wei Huang, Seung-Moon Yoo, D. Keen, Zhenzhou Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: toward an advanced intelligent memory system. In *ICCD*, 1999.

[114] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM. *MICRO*, 1997.

[115] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, Jaewook Shin, and Joonseok Park. Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture. In *SC*, 1999.

[116] D. G. Elliott, W. M. Snelgrove, and M. Stumm. Computational RAM: A Memory-SIMD Hybrid And Its Application To DSP. In *CICC*, 1992.

[117] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the terasys massively parallel pim array. *Computer*, 1995.

[118] Kevin Hsieh, Samira Manabi Khan, Nandita Vijaykumar, Kevin K. Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. *ICCD*, 2016.

[119] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference (DAC)*, 2016.

[120] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute caches. In *HPCA*, 2017.

[121] Qing Guo et al. A resistive TCAM accelerator for data-intensive computing. In *Micro*, 2011.

[122] Qing Guo et al. AC-DIMM: Associative computing with STT-MRAM. In *ISCA*, 2013.

[123] Nastaran Hajinazar et al. SIMDRAM: A framework for bit-serial simd processing using dram. In *ASPLOS*, 2021.

[124] Rajaraman Ramanarayanan et al. Combined set bit count and detector logic, U.S. Patent US8214414B2 Sep. 2008.

[125] Jeremie S. Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, Onur Mutlu, and et al. Grim-filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies. *BMC Genomics*, 19(S2), 2018.

[126] Yinan Li and Jignesh M. Patel. BitWeaving: Fast scans for main memory data processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2013.

[127] In-Memory Database Market. https://www.alliedmarketresearch.com/in-memory-database-market-A31497, October 2022.

[128] Utku Sirin and Anastasia Ailamaki. Micro-architectural analysis of OLAP: limitations and opportunities. *Proceedings of the VLDB Endowment*, 13(6):840–853, 2020.

[129] Shimeng Yu, Hongwu Jiang, Shanshi Huang, Xiaochen Peng, and Anni Lu. Compute-in-memory chips for deep learning: Recent trends and prospects. *IEEE Circuits and Systems Magazine*, 21(3):31–56, 2021.

[130] Business Intelligence and Analytics Market. https://www.emergenresearch.com/request-sample/467, January 2021.

[131] Daniel Abadi, Anastasia Ailamaki, David Andersen, Peter Bailis, Magdalena Balazinska, Philip A. Bernstein, Peter Boncz, Surajit Chaudhuri, Alvin Cheung, Anhai Doan, Luna Dong, Michael J. Franklin, Juliana Freire, Alon Halevy, Joseph M. Hellerstein, Stratos Idreos, Donald Kossmann, Tim Kraska, Sailesh Krishnamurthy, Volker Markl, Sergey Melnik, Tova Milo, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Jignesh Patel, Andrew Pavlo, Raluca Popa, Raghu Ramakrishnan, Christopher Re, Michael Stonebraker, and Dan Suciu. The seattle report on database research. *Communications of the ACM*, 65(8):72–79, August 2022.

[132] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R Stan, and Kevin Skadron. Fulcrum: A simplified control and access mechanism toward flexible and practical in-situ accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 556–569. IEEE, 2020.

[133] Helena Caminal, Yannis Chronis, Tianshu Wu, Jignesh M. Patel, and José F. Martínez. Accelerating database analytic query workloads using an associative processor. ISCA '22, 2022.

[134] Yinan Li and Jignesh M. Patel. WideTable: An accelerator for analytical data processing. *Proceedings of the VLDB Endowment*, 7(10):907–918, June 2014.

[135] Ben Hannel and Kevin Leong. Rockset performance evaluation on the star schema benchmark.

[136] Clickhouse. https://clickhouse.com/docs/en/getting-started/example-datasets/star-schema, 2023.

[137] StarRocks. https://docs.starrocks.io/en-us/2.5/benchmarking/ SSB_benchmarking, 2023.

[138] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. Quickstep: A data platform based on the scaling-up approach. *Proc. VLDB Endow.*, 11(6):663–676, oct 2018.

[139] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. Byteslice: Pushing the envelope of main memory data processing with a new storage layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 31–46, New York, NY, USA, 2015. Association for Computing Machinery.

[140] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA database - an architecture overview. *IEEE Data Eng. Bull.*, 35:28–33, 03 2012.

[141] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast updates on read-optimized databases using multi-core CPUs. *Proc. VLDB Endow.*, 5(1):61–72, sep 2011.

[142] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. Constant-time query processing. In *2008 IEEE 24th International Conference on Data Engineering*, pages 60–69, 2008.

[143] Craig Chasseur and Jignesh M. Patel. Design and evaluation of storage organizations for read-optimized main memory databases. *Proc. VLDB Endow.*, 6(13):1474–1485, aug 2013.

[144] P. O'Neil, E. O'Neil, and X. Chen. The star schema benchmark. http://www.cs.umb.edu/~poneil/StarSchemaB.pdf, Jan 2007.

[145] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, KyungSoo Kim, Jin Jung, Ilkwon Yun, Sung Joo Park, Hyunsun Park, Joonho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin S. Lee. Near-memory processing in action: Accelerating personalized recommendation with AxDIMM. *IEEE Micro*, 42(1):116–127, 2022.

[146] Mohammad Alian, Seung Won Min, Hadi Asgharimoghaddam, Ashutosh Dhar, Dong Kai Wang, Thomas Roewer, Adam McPadden, Oliver O'Halloran, Deming Chen, Jinjun Xiong, Daehoon Kim, Wen-mei Hwu, and Nam Sung Kim. Application-transparent near-memory processing architecture with memory channel network. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 802–814, 2018.

[147] Tpc-h benchmark specification.

[148] Intel® Data Direct I/O Technology (Intel® DDIO): A primer.

[149] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostic. Reexamining direct cache access to optimize i/o intensive applications for multi-hundred-gigabit networks. In *USENIX ATC'20*, pages 673–689, 2020.

[150] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Packet Chasing: Spying on Network Packets over a Cache Side-Channel. In *ISCA*, 2020.

[151] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network i/o. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 50–59, 2005.

[152] Dan Tang, Yungang Bao, Weiwu Hu, and Mingyu Chen. DMA cache: Using on-chip storage to architecturally separate i/o data from cpu data for improving i/o performance. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[153] Minhu Wang, Mingwei Xu, and Jianping Wu. Understanding I/O direct cache access performance for end host networking. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(1), feb 2022.

[154] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. Don't forget the I/O when allocating your LLC. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 112–125, 2021.

[155] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The RISC-V instruction set manual. Volume I User-level ISA, 2014.

[156] Steven JE Wilton and Norman P Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, 1996.

[157] Paulo C. Santos, Geraldo F. Oliveira, Diego G. Tomé, Marco A. Z. Alves, Eduardo C. de Almeida, and Luigi Carro. Operand size reconfiguration for big data processing in memory. In David Atienza and Giorgio Di Natale, editors, *Design, Automation and Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 710–715. IEEE, 2017.

[158] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, oct 2008.

[159] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, Aug 2009.

[160] Haran Boral and David J DeWitt. Database machines: An idea whose time has passed? A critique of the future of database machines, 1983.

[161] Samsung smartssd. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html, 2023.

[162] Sam Likun Xi, Aurelia Augusta, Manos Athanassoulis, and Stratos Idreos. Beyond the wall: Near-data processing for databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, 2015.

[163] Amirali Boroumand, Saugata Ghose, Geraldo F. Oliveira, and Onur Mutlu. Polynesia: Enabling high-performance and energy-efficient hybrid transactional/analytical databases with hardware/-software co-design. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2997–3011, 2022.

[164] Peter Bakkum and Kevin Skadron. Accelerating SQL database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, page 94–103, 2010.

[165] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, page 1617–1632, 2020.

[166] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment*, 7(11):963–974, Jul 2014.

[167] Daniel Ziener, Florian Bauer, Andreas Becher, Christopher Dennl, Klaus Meyer-Wegener, Ute Schürfeld, Jürgen Teich, Jörg-Stephan Vogt, and Helmut Weber. FPGA-based dynamically reconfigurable sql query processing. *ACM Transactions on Reconfigurable Technology Systems*, Aug 2016.

[168] Philippos Papaphilippou and Wayne Luk. Accelerating database systems using fpgas: A survey. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 125–1255, 2018.

[169] Nvidia Thrust: Parallel algorithms library.

[170] Mark Raasveldt and Hannes Mühleisen. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, Amsterdam Netherlands, June 2019. ACM.

[171] Oreste Villa, Daniel R Johnson, Mike Oconnor, Evgeny Bolotin, David Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero, et al. Scaling the power wall: A path to exascale. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 830–841. IEEE, 2014.

[172] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman, Apoorv Srivastava, et al. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, pages 57–es, 1999.

[173] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pages 5–14. IEEE, 2012.

[174] Christoforos E Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanovic, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, et al. Scalable processors in the billion-transistor era: IRAM. *Computer*, 30(9):75–78, 1997.

[175] Harold S Stone. A logic-in-memory computer. *IEEE Transactions on Computers*, 100(1):73–78, 1970.

[176] Kevin K Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding latency variation in modern DRAM chips: Experimental characterization, analysis, and optimization. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, pages 323–336, 2016.

[177] Nezam Rohbani, Sina Darabi, and Hamid Sarbazi-Azad. PF-DRAM: A precharge-free DRAM structure. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 126–138. IEEE, 2021.

[178] Nezam Rohbani, Mohammad Arman Soleimani, and Hamid Sarbazi-Azad. PIPF-DRAM: Processing in precharge-free DRAM. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 1075–1080, 2022.

[179] Helena Caminal, Kailin Yang, Srivatsa Srinivasa, Akshay Krishna Ramanathan, Khalid Al-Hawaj, Tianshu Wu, Vijaykrishnan Narayanan, Christopher Batten, and José F Martínez. CAPE: A content-addressable processing engine. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 557–569. IEEE, 2021.

[180] João Dinis Ferreira, Gabriel Falcao, Juan Gómez-Luna, Mohammed Alser, Lois Orosa, Mohammad Sadrosadati, Jeremie S Kim, Geraldo F Oliveira, Taha Shahroodi, Anant Nori, et al. pLUTo: Enabling massively parallel computation in DRAM via lookup tables. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 900–919. IEEE, 2022.

[181] Quan Deng, Youtao Zhang, Minxuan Zhang, and Jun Yang. LAcc: Exploiting lookup table-based fast and accurate vector multiplication in DRAM-based CNN accelerator. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[182] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. To PIM or not for emerging general purpose processing in DDR memory systems. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 231–244, 2022.

[183] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and TN Vijaykumar. Newton: A DRAM-maker's accelerator-in-memory (AiM) architecture for machine learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 372–385. IEEE, 2020.

[184] Jin Hyun Kim, Shin-haeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, Bengseng Phuah, et al. Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for ML accelerators and beyond. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–26. IEEE, 2021.

[185] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[186] Kanishkan Vadivel, Lorenzo Chelini, Ali BanaGozar, Gagandeep Singh, Stefano Corda, Roel Jordans, and Henk Corporaal. TDO-CIM: Transparent detection and offloading for computation in-memory. In *2020 Design, Automation and Test in Europe Conference & Exhibition (DATE)*, pages 1602–1605. IEEE, 2020.

[187] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[188] Zhengrong Wang, Christopher Liu, and Tony Nowatzki. Infinity stream: Enabling transparent and automated in-memory computing. *IEEE Computer Architecture Letters*, 21(2):85–88, 2022.

[189] Ramyad Hadidi, Lifeng Nai, Hyojong Kim, and Hyesoon Kim. Cairo: A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(4):1–25, 2017.

[190] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.

[191] Orian Leitersdorf, Dean Leitersdorf, Jonathan Gal, Mor Dahan, Ronny Ronen, and Shahar Kvatinsky. AritPIM: High-throughput in-memory arithmetic. *IEEE Transactions on Emerging Topics in Computing*, 2023.

[192] Jingcheng Wang, Xiaowei Wang, Charles Eckert, Arun Subramaniyan, Reetuparna Das, David Blaauw, and Dennis Sylvester. A 28-nm compute SRAM with bit-serial logic/arithmetic operations for programmable in-memory vector computing. *IEEE Journal of Solid-State Circuits*, 55(1):76–86, 2019.

[193] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. Ieee, 2007.

[194] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001.

[195] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking memory-centric computing systems: Analysis of real processing-in-memory hardware. In *2021 12th International Green and Sustainable Computing Conference (IGSC)*, 2021.

[196] Ki-Whan Song, Jin-Young Kim, Jae-Man Yoon, Sua Kim, Huijung Kim, Hyun-Woo Chung, Hyungi Kim, Kanguk Kim, Hwan-Wook Park, Hyun Chul Kang, et al. A 31 ns random cycle vcat-based $4F^2$ DRAM with manufacturability and enhanced cell efficiency. *IEEE Journal of Solid-State Circuits*, 45(4):880–888, 2010.

[197] Christopher Celio, Pi-Feng Chiu, Krste Asanović, Borivoje Nikolić, and David Patterson. BROOM: An open-source out-of-order processor with resilient low-voltage operation in 28-nm CMOS. *IEEE Micro*, 39(2):52–60, 2019.

[198] Intel. Intel performance counter monitor. http://www.intel.com/software/pcm, 2019.

[199] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[200] 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. GenASM: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[201] Stephan C Schuster. Next-generation sequencing transforms today's biology. *Nature methods*, 5(1):16–18, 2008.

[202] Illumina. Miseq system. https://www.illumina.com/systems/sequencing-platforms/miseq.html/.

[203] Illumina. Nextseq 2000 system. https://www.illumina.com/systems/sequencing-platforms/nextseq-1000-2000.html.

[204] Illumina. Novaseq 6000 system. https://www.illumina.com/systems/sequencing-platforms/novaseq.html.

[205] M. Alser, Z. Bingöl, D. S. Cali, J. Kim, S. Ghose, C. Alkan, and O. Mutlu. Accelerating genome analysis: A primer on an ongoing journey. *IEEE Micro*, 40(5):65–75, 2020.

[206] Jay Shendure, Shankar Balasubramanian, George M Church, Walter Gilbert, Jane Rogers, Jeffery A Schloss, and Robert H Waterston. Dna sequencing at 40: past, present and future. *Nature*, 550(7676):345–353, 2017.

[207] Hongyi Xin, Donghyuk Lee, Farhad Hormozdiari, Samihan Yedkar, Onur Mutlu, and Can Alkan. Accelerating read mapping with fasthash. In *BMC genomics*, volume 14, page S13. Springer, 2013.

[208] Matthias Hess, Alexander Sczyrba, Rob Egan, Tae-Wan Kim, Harshal Chokhawala, Gary Schroth, Shujun Luo, Douglas S Clark, Feng Chen, Tao Zhang, Roderick I. Mackie, Len A. Pennacchio, Susannah G. Tringe, Axel Visel, Tanja Woyke, Zhong Wang, and Edward M. Rubin. Metagenomic discovery of biomass-degrading genes and genomes from cow rumen. *Science*, 331(6016):463–467, 2011.

[209] Adina Chuang Howe, Janet K Jansson, Stephanie A Malfatti, Susannah G Tringe, James M Tiedje, and C Titus Brown. Tackling soil diversity with the assembly of large, complex metagenomes. *Proceedings of the National Academy of Sciences*, 111(13):4904–4909, 2014.

[210] Junjie Qin, Ruiqiang Li, Jeroen Raes, Manimozhiyan Arumugam, Kristoffer Solvsten Burgdorf, Chaysavanh Manichanh, Trine Nielsen, Nicolas Pons, Florence Levenez, Takuji Yamada, et al. A human gut microbial gene catalogue established by metagenomic sequencing. *nature*, 464(7285):59–65, 2010.

[211] Erwin L van Dijk, Yan Jaszczyszyn, Delphine Naquin, and Claude Thermes. The third revolution in sequencing technology. *Trends in Genetics*, 34(9):666–681, 2018.

[212] Can Alkan, Bradley P Coe, and Evan E Eichler. Genome structural variation discovery and genotyping. *Nature Reviews Genetics*, 12(5):363–376, 2011.

[213] Lars Feuk, Andrew R Carson, and Stephen W Scherer. Structural variation in the human genome. *Nature Reviews Genetics*, 7(2):85–97, 2006.

[214] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, Alexey V. Pyshkin, Alexander V. Sirotkin, Nikolay Vyahhi, Glenn Tesler, Max A. Alekseyev, and Pavel A. Pevzner. Spades: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of computational biology*, 19(5):455–477, 2012.

[215] SD Jackman, BP Vandervalk, H Mohamadi, J Chu, S Yeo, SA Hammond, G Jahesh, H Khan, L Coombe, RL Warren, and I. Birol. Abyss 2.0: resource-efficient assembly of large genomes using a bloom filter. *Genome Res*, 2017.

[216] Sergey Nurk, Dmitry Meleshko, Anton Korobeynikov, and Pavel A. Pevzner. metaspades: a new versatile metagenomic assembler. *Genome Research*, 27(5):824–834, 2017.

[217] Mikhail Kolmogorov, Derek M Bickhart, Bahar Behsaz, Alexey Gurevich, Mikhail Rayko, Sung Bong Shin, Kristen Kuhn, Jeffrey Yuan, Evgeny Polevikov, Timothy PL Smith, et al. metaflye: scalable long-read metagenome assembly using repeat graphs. *Nature Methods*, 17(11):1103–1110, 2020.

[218] Yu Lin, Jeffrey Yuan, Mikhail Kolmogorov, Max W Shen, Mark Chaisson, and Pavel A Pevzner. Assembly of long error-prone reads using de bruijn graphs. *Proceedings of the National Academy of Sciences*, 113(52):E8396–E8405, 2016.

[219] Mikhail Kolmogorov, Jeffrey Yuan, Yu Lin, and Pavel A Pevzner. Assembly of long, error-prone reads using repeat graphs. *Nature biotechnology*, 37(5):540–546, 2019.

[220] R. Gebelhoff. Sequencing the genome creates so much data we don't know what to do with it. " *The Washington Post*, 2015.

[221] Minxuan Zhou, Andreas Prodromou, Rui Wang, Hailong Yang, Depei Qian, and Dean Tullsen. Temperature-aware dram cache management—relaxing thermal constraints in 3-d systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):1973–1986, 2019.

[222] Minxuan Zhou, Muzhou Li, Mohsen Imani, and Tajana Rosing. Hygraph: Accelerating graph processing with hybrid memory-centric computing. In *2021 Design, Automation and Test in Europe Conference & Exhibition (DATE)*, pages 330–335. IEEE, 2021.

[223] G. Kim, J. Kim, J. H. Ahn, and J. Kim. Memory-centric system interconnect design with hybrid memory cubes. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 145–155, 2013.

[224] Evangelos Georganas, Aydin Buluç, Jarrod Chapman, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. Parallel de bruijn graph construction and traversal for de novo genome assembly. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 437–448. IEEE, 2014.

[225] Jorge Duitama, Alena Zablotskaya, Rita Gemayel, An Jansen, Stefanie Belet, Joris R. Vermeesch, Kevin J. Verstrepen, and Guy Froyen. Large-scale analysis of tandem repeat variability in the human genome. *Nucleic Acids Research*, 42(9):5728–5741, 2014.

[226] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–52:12, November 2011.

[227] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[228] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory. In *2012 Design, Automation and Test in Europe Conference & Exhibition (DATE)*, pages 33–38. IEEE, 2012.

[229] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[230] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524, 2008.

[231] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, 2006.

[232] Dennis A Benson, Ilene Karsch-Mizrachi, David J Lipman, James Ostell, Barbara A Rapp, and David L Wheeler. Genbank. *Nucleic acids research*, 28(1):15–18, 2000.

[233] Weichun Huang, Leping Li, Jason R Myers, and Gabor T Marth. Art: a next-generation sequencing read simulator. *Bioinformatics*, 28(4):593–594, 2012.

[234] Intel. Intel VTune Amplifier. https://software.intel.com/en-us/vtune, 2019.

[235] Sébastien Boisvert, François Laviolette, and Jacques Corbeil. Ray: Simultaneous assembly of reads from a mix of high-throughput sequencing technologies. *Journal of Computational Biology*, 17(11):1519–1533, 2010.

[236] B. G. Jackson, M. Regennitter, X. Yang, P. S. Schnable, and S. Aluru. Parallel de novo assembly of large genomes from high-throughput short reads. *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010.

[237] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. Hipmer: an extreme-scale de novo genome assembler. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[238] Priyanka Ghosh, Sriram Krishnamoorthy, and Ananth Kalyanaraman. Pakman: Scalable assembly of large genomes on distributed memory machines. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 578–589, 2019.

[239] Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. Idba-ud: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth. *Bioinformatics*, 28(11):1420–1428, 2012.

[240] S. H. Pugsley, J. Jestes, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. Comparing implementations of near-data computing with in-memory mapreduce workloads. *IEEE Micro*, 2014.

[241] Farzaneh Zokaee, Hamid R. Zarandi, and Lei Jiang. Aligner: A process-in-memory architecture for short read alignment in rerams. *IEEE Computer Architecture Letters*, 17(2):237–240, 2018.

[242] Farzaneh Zokaee, Mingzhe Zhang, and Lei Jiang. Finder: Accelerating fm-index-based exact pattern matching in genomic sequences through reram technology. In *PACT*, 2019.

[243] De novo genome assembly from next-generation sequencing (ngs) reads. *Next-Generation Sequencing Data Analysis*, page 144–155, 2016.

[244] Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[245] Daniel R. Zerbino and Ewan Birney. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18(5):821–829, 2008.

[246] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven J M Jones, and Inanç Birol. Abyss: A parallel assembler for short read sequence data, Jun 2009.

[247] Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics*, 31(10):1674–1676, 01 2015.

[248] Jason R. Miller, Arthur L. Delcher, Sergey Koren, Eli Venter, Brian P. Walenz, Anushka Brownley, Justin Johnson, Kelvin Li, Clark Mobarry, Granger Sutton, and et al. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24(24):2818–2824, 2008.

[249] Myers EW;Sutton GG;Delcher AL;Dew IM;Fasulo DP;Flanigan MJ;Kravitz SA;Mobarry CM;Reinert KH;Remington KA;Anson EL;Bolanos RA;Chou HH;Jordan CM;Halpern AL;Lonardi S;Beasley EM;Brandon RC;Chen L;Dunn PJ;Lai Z;Liang Y;Nusskern DR;Zhan M;Zhang Q;Zheng X;Rubin. A whole-genome assembly of drosophila.

[250] David B. Jaffe, Jonathan Butler, Sante Gnerre, Evan Mauceli, Kerstin Lindblad-Toh, Jill P. Mesirov, Michael C. Zody, and Eric S. Lander. Whole-genome sequence assembly for mammalian genomes: Arachne 2. *Genome Research*, 13(1):91–96, 2003.

[251] Binghang Liu, Yujian Shi, Jianying Yuan, Xuesong Hu, Hao Zhang, Nan Li, Zhenyu Li, Yanxiang Chen, Desheng Mu, Wei Fan, and et al. Estimation of genomic characteristics by analyzing k-mer frequency in de novo genome projects, Aug 2013.

[252] Price AL;Jones NC;Pevzner PA;. De novo identification of repeat families in large genomes.

[253] Ruiqiang Li, Jia Ye, Songgang Li, Jing Wang, Yujun Han, Chen Ye, Jian Wang, Huanming Yang, Jun Yu, Gane Wong, and et al. Reas: Recovery of ancestral sequences for transposable elements from the unassembled reads of a whole genome shotgun. *PLoS Computational Biology*, preprint(2005), 2005.

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

[255] A. Lefebvre, T. Lecroq, H. Dauchel, and J. Alexandre. Forrepeats: Detects repeats on entire chromosomes and between genomes. *Bioinformatics*, 19(3):319–326, 2003.

[256] Stefan Kurtz, Apurva Narechania, Joshua C Stein, and Doreen Ware. A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genomics*, 9(1), 2008.

[257] Fanny-Dhelia Pajuste, Lauris Kaplinski, Märt Möls, Tarmo Puurand, Maarja Lepamets, and Maido Remm. Fastgt: An alignment-free method for calling common snvs directly from raw sequencing reads. 2016.

[258] Leena Salmela and Jan Schröder. Correcting errors in short reads by multiple alignments. *Bioinformatics*, 27(11):1455–1461, 04 2011.

[259] Robert C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–1797, 03 2004.

[260] L. Salmela and J. Schroder. Correcting errors in short reads by multiple alignments. *Bioinformatics*, 27(11):1455–1461, 2011.

[261] Pandey P;Bender MA;Johnson R;Patro R;Berger B;. Squeakr: An exact and approximate k-mer counting system, 2023.

[262] Páll Melsted and Jonathan K Pritchard. Efficient counting of k-mers in dna sequences using a bloom filter. *BMC Bioinformatics*, 12(1), 2011.

[263] Roy RS;Bhattacharya D;Schliep A;. Turtle: Identifying frequent k-mers with cache-efficient algorithms.

[264] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 01 2011.

[265] Swati C Manekar and Shailesh R Sathe. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience*, 2018.

[266] Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, 05 2017.

[267] Sebastian Deorowicz, Agnieszka Debudaj-Grabysz, and Szymon Grabowski. Disk-based k-mer counting on a pc. *BMC Bioinformatics*, 14(1), 2013.

[268] Peter Audano and Fredrik Vannberg. Kanalyze: A fast versatile pipelined k-mer toolkit. *Bioinformatics*, 30(14):2070–2072, 2014.

[269] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. Kmc 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

[270] Lauris Kaplinski, Maarja Lepamets, and Maido Remm. Genometester4: A toolkit for performing basic set operations - union, intersection and complement on k-mer lists. *GigaScience*, 4(1), 2015.

[271] Nicola Cadenelli, Zoran Jaksić, Jordà Polo, and David Carrera. Considerations in using opencl on gpus and fpgas for throughput-oriented genomics workloads. *Future Generation Computer Systems*, 94:148–159, 2019.

[272] Nathaniel Mcvicar, Chih-Ching Lin, and Scott Hauck. K-mer counting using bloom filters with an fpga-attached hmc. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 203–210, 2017.

[273] Wenqin Huangfu, Krishna T. Malladi, Shuangchen Li, Peng Gu, and Yuan Xie. Nest: Dimm based near-data-processing accelerator for sea counting. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2020.

[274] Yu-Ching Hu, Murtuza Taher Lokhandwala, Te I., and Hung-Wei Tseng. Dynamic multi-resolution data storage. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 196–210, New York, NY, USA, 2019. Association for Computing Machinery.

[275] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. Ssd in-storage computing for list intersection. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16, New York, NY, USA, 2016. Association for Computing Machinery.

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

[277] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, and Sang-Won Lee. Fast, energy efficient scan inside flash memory. In *ADMS@VLDB*, 2011.

[278] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating application objects efficiently for heterogeneous computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 53–65, 2016.

[279] Young-Sik Lee, Luis Cavazos Quero, Youngjae Lee, Jin-Soo Kim, and Seungryoul Maeng. Accelerating external sorting via On-the-fly data merge in active SSDs. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, June 2014. USENIX Association.

[280] Simona Boboila, Youngjae Kim, Sudharshan S. Vazhkudai, Peter Desnoyers, and Galen M. Shipman. Active flash: Out-of-core data analytics on flash storage. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2012.

[281] Yunjae Lee, Jinha Chung, and Minsoo Rhu. 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, page 932–945, New York, NY, USA, 2022. Association for Computing Machinery.

[282] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. In-storage processing of database scans and joins. *Inf. Sci.*, 327(C):183–200, jan 2016.

[283] 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. Deepstore: In-storage acceleration for intelligent queries. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 224–238, New York, NY, USA, 2019. Association for Computing Machinery.

[284] Jilan Lin, Ling Liang, Zheng Qu, Ishtiyaque Ahmad, Liu Liu, Fengbin Tu, Trinabh Gupta, Yufei Ding, and Yuan Xie. Inspire: In-storage private information retrieval via protocol and architecture co-design. ISCA '22, page 102–115, New York, NY, USA, 2022. Association for Computing Machinery.

[285] 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. 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 2022, page 635–654, New York, NY, USA, 2022. Association for Computing Machinery.

[286] Duck-Ho Bae, Jin-Hyung Kim, Yong-Yeon Jo, Sang-Wook Kim, Hyun-Kyo Oh, and Chanik Park. Intelligent ssd: a turbo for big data mining. *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, 2013.

[287] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor. ASIC Clouds: Specializing the Datacenter. In *ISCA*, 2016.

[288] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor. Moonwalk: NRE optimization in asic clouds. In *ASPLOS*, 2017.

[289] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. TriCache: A User-Transparent block cache enabling High-Performance Out-of-Core processing with In-Memory programs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 395–411, Carlsbad, CA, 2022. USENIX Association.

[290] Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, and Jason Cong. Fans: Fpga-accelerated near-storage sorting. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 106–114, 2021.

[291] Crate seahash, 2023.

[292] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science*, DMTCS Proceedings vol. AH, 03 2012.

[293] national center for biotechnology information, 2023.

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

[295] Hiroshi Maejima, Kazushige Kanda, Susumu Fujimura, Teruo Takagiwa, Susumu Ozawa, Jumpei Sato, Yoshihiko Shindo, Manabu Sato, Naoaki Kanagawa, Junji Musha, et al. A 512gb 3b/cell 3d flash memory on a 96-word-line-layer technology. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 336–338. IEEE, 2018.

[296] 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, et al. A 512-gb 3-b/cell 64-stacked wl 3-d-nand flash memory. *IEEE Journal of Solid-State Circuits*, 53(1):124–133, 2017.

[297] 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. 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 *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 218–220. IEEE, 2020.

[298] Yuta Toriyama and Dejan Marković. 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 journal of solid-state circuits*, 53(8):2378–2388, 2018.

[299] Yu Cai, Saugata Ghose, E. Haratsch, Yixin Luo, and O. Mutlu. Errors in flash-memory-based solid-state drives: Analysis, mitigation, and recovery: Semantic scholar, Jan 1970.

[300] Páll Melsted and Jonathan K Pritchard. Efficient counting of k-mers in dna sequences using a bloom filter. *BMC Bioinformatics*, 12(1), 2011.

[301] Shigui Qi, Dan Feng, Nan Su, Linjun Mei, and Jingning Liu. Cdf-ldpc: A new error correction method for ssd to improve the read performance. *ACM Trans. Storage*, 13(1), feb 2017.

[302] Pai-Yu Chen, Xiaochen Peng, and Shimeng Yu. Neurosim+: An integrated device-to-algorithm framework for benchmarking synaptic devices and array architectures. In *2017 IEEE International Electron Devices Meeting (IEDM)*, pages 6.1.1–6.1.4, 2017.

[303] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 601–618, Austin, TX, August 2016. USENIX Association.

[304] IBM. Ibm advances research in analog ai computing. https://cambrian-ai.com/downloads/ibm-advances-research-in-analog-ai-computing/, 2021.

[305] Deepak Kadetotad, Zihan Xu, Abinash Mohanty, Pai-Yu Chen, Binbin Lin, Jieping Ye, Sarma Vrudhula, Shimeng Yu, Yu Cao, and Jae-sun Seo. Parallel architecture with resistive crosspoint array for dictionary learning acceleration. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 5(2):194–204, 2015.

[306] Chaofei Yang, Beiye Liu, Hai Li, Yiran Chen, Mark Barnell, Qing Wu, Wujie Wen, and Jeyavijayan Rajendran. Security of neuromorphic computing: Thwarting learning attacks using memristor's obsolescence effect. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, 2016.

[307] Chidhambaranathan Rajamanikkam, S. RajeshJ., Sanghamitra Roy, and Koushik Chakraborty. Understanding security threats in emerging neuromorphic computing architecture. *Journal of Hardware and Systems Security*, 5:45–57, 2021.

[308] Yi Cai, Xiaoming Chen, Lu Tian, Yu Wang, and Huazhong Yang. Enabling secure in-memory neural network computing by sparse fast gradient encryption. In *ICCAD*, pages 1–8, 2019.

[309] Jeyavijayan Rajendran, Ozgur Sinanoglu, and Ramesh Karri. Is split manufacturing secure? In *2013 Design, Automation and Test in Europe Conference Exhibition (DATE)*, pages 1259–1264, 2013.

[310] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 393–406, New York, NY, USA, 2018. Association for Computing Machinery.

[311] Vasileios Tenentes et al. Run-time protection of multi-core processors from power-noise denial-of-service attacks. TDMR '20.

[312] Rajat Subhra Chakraborty, Francis G. Wolff, Somnath Paul, Christos A. Papachristou, and Swarup Bhunia. Mero: A statistical approach for hardware trojan detection. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 396–410. Springer, 2009.

[313] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. Fanci: Identification of stealthy malicious logic using boolean functional analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer amp; Communications Security*, CCS '13, page 697–708, New York, NY, USA, 2013. Association for Computing Machinery.

[314] Matthew Hicks et al. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *2010 IEEE Symposium on Security and Privacy*, pages 159–172, 2010.

[315] Maxime Montoya, Thomas Hiscock, Simone Bacles-Min, Anca Molnos, and Jacques Fournier. Adaptive masking: a dynamic trade-off between energy consumption and hardware security. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 559–566, 2019.

[316] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature Methods*, 9(4):357–359, 2012.

[317] Eric A. Franzosa, Lauren J. Mciver, Gholamali Rahnavard, Luke R. Thompson, Melanie Schirmer, George Weingart, Karen Schwarzberg Lipson, Rob Knight, J. Gregory Caporaso, Nicola Segata, and et al. Species-level functional profiling of metagenomes and metatranscriptomes. *Nature Methods*, 15(11):962–968, 2018.

[318] Sergey Nurk, Dmitry Meleshko, Anton Korobeynikov, and Pavel A. Pevzner. metaspades: a new versatile metagenomic assembler. *Genome Research*, 27(5):824–834, 2017.

[319] D. Takashima, S. Watanabe, H. Nakano, Y. Oowaki, and K. Ohuchi. Open/folded bit-line arrangement for ultra-high-density dram's. *IEEE Journal of Solid-State Circuits*, 1994.

[320] Derrick Wood and Jennifer Lu. Kraken website. https://ccb.jhu.edu/software/kraken/, 2020.

[321] Samsung. Samsung hbm. https://semiconductor.samsung.com/dram/hbm/, 2023.

[322] Albert Einstein. Zur Elektrodynamik bewegter Körper. (German) [On the electrodynamics of moving bodies]. *Annalen der Physik*, 322(10):891–921, 1905.

[323] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[324] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[325] M. F. Chang, Y. Chen, J. Cong, P. Huang, C. Kuo, and C. H. Yu. The smem seeding acceleration for dna sequence alignment. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 32–39, 2016.

[326] Panagiotis D. Vouzis and Nikolaos V. Sahinidis. Gpu-blast: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188, 2011.

[327] Intel. Intel Performance Counter Monitor. http://www.intel.com/software/pcm, 2019.

[328] Heshan Lin Jing Zhang, Hao Wang and Wu chun Feng. cublastp: Fine-grained parallelization of protein sequence search on a gpu. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[329] JiaShun Xiao. BLAST-bioinfor-tool. https://github.com/JiaShun-Xiao/BLAST-bioinfor-tool, 2019.

[330] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem, 2013.

[331] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[332] Onur Mutlu. Processing data where it makes sense in modern computing systems: Enabling in-memory computation. In *2018 7th Mediterranean Conference on Embedded Computing (MECO)*, pages 8–9, 2018.

[333] Klus Petr, Lam Simon, Lyberg Dag, Cheung Ming Sin, Pullan Graham, McFarlane Ian, Yeo Giles SH, and Lam Brian YH. Barracuda - a fast short read sequence aligner using graphics processing units. *BMC Research Notes*, 5, 2012.

[334] Yongchao Liu and Bertil Schmidt. Long read alignment based on maximal exact match seeds. *Bioinformatics*, 28(18):318—-324, 2012.

[335] Mohammed Alser, Hasan Hassan, Hongyi Xin, Oğuz Ergin, Onur Mutlu, and Can Alkan. Gatekeeper: a new hardware architecture for accelerating pre-alignment in dna short read mapping. *Bioinformatics*, 33(21):3355–3363, 2017.

[336] W. James Kent. BLAT–the BLAST-like alignment tool. *Genome research*, 12(4):656—664, 2002.

[337] Matei David, Misko Dzamba, Dan Lister, Lucian Ilie, and Michael Brudno. SHRiMP2: Sensitive yet Practical Short Read Mapping. *Bioinformatics*, 27(7):1011–1012, 2011.

[338] Barry Merriman Nils Homer and Stanley F. Nelson. BFAST: An Alignment Tool for Large Scale Genome Resequencing. *PLoS ONE*, 4(11).

[339] U Kang, Hak soo Yu, Churoo Park, Hongzhong Zheng, John B. Halbert, Kuljit S. Jalandhar Bains, Seong-Jin Jang, and Joo Sun Choi. Co-architecting controllers and dram to enhance dram process scaling. 2014.

[340] Donghyuk Lee, Yoongu Kim, Vivek Seshadri, Jamie Liu, Lavanya Subramanian, and Onur Mutlu. Tiered-latency DRAM: enabling low-latency main memory at low cost. *CoRR*, abs/1805.03048, 2018.

[341] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu. Softmc: A flexible and practical open-source infrastructure for enabling experimental dram studies. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 241–252, 2017.

[342] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[343] Samuel S. Minot, Niklas Krumm, and Nicholas B. Greenfield. One Codex: A Sensitive and Accurate Data Platform for Genomic Microbial Identification. *bioRxiv*, 2015.

[344] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[345] J. T. Pawlowski. Hybrid Memory Cube (HMC). In *HCS*, 2011.

[346] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C. . Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3):17:1–17:14, 2015.

[347] Wei Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan. Hotspot: a compact thermal modeling methodology for early-stage vlsi design. *TVLSI*, 14(5):501–513, 2006.

[348] High Bandwidth Memory (HBM) DRAM. https://www.amd.com/en/technologies/hbm.

[349] 4Gb: x4, x8, x16 DDR3 SDRAM Features. https://www.micron.com/-/media/documents/products/data%20sheet/dram/ddr3/4gb_ddr3_sdram.pdf.

[350] Bruce Jacob, David Wang, and Spencer Ng. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[351] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 415–426, 2015.

[352] Helena Caminal, Kailin Yang, Srivatsa Srinivasa, Akshay Krishna Ramanathan, Khalid Al-Hawaj, Tianshu Wu, Vijaykrishnan Narayanan, Christopher Batten, and José F. Martínez. Cape: A content-addressable processing engine. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 557–569, 2021.

[353] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. Materialization strategies in a column-oriented dbms. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 466–475, 2007.

[354] Lakshmikant Shrinivas, Sreenath Bodagala, Ramakrishna Varadarajan, Ariel Cary, Vivek Bharathan, and Chuck Bear. Materialization strategies in the vertica analytic database: Lessons learned. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1196–1207, 2013.

[355] George Chernishev, Viacheslav Galaktionov, Valentin Grigorev, Evgeniy Klyuchikov, and Kirill Smirnov. A comprehensive study of late materialization strategies for a disk-based column-store. In *Proceedings of the 24th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP) co-located with the 25th International Conference on Extending Database Technology and the 25th International Conference on Database Theory(EDBT/ICDT 2022)*, DOLAP' 22, 2022.

[356] Intel. Intel In-Memory Analytics Accelerator Architecture Specification. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, Apr 2023.

[357] Amazon Web Services. AQUA (Advanced Query Accelerator). https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/, Apr 2021.

[358] Oracle. Oracle Data Analytics Accelerator (DAX) for SPARC. https://blogs.oracle.com/linux/post/oracle-data-analytics-accelerator-dax-for-sparc, Jul 2018.

[359] Donghyuk Lee, Yoongu Kim, Vivek Seshadri, Jamie Liu, Lavanya Subramanian, and Onur Mutlu. Tiered-latency DRAM: A low latency and low cost DRAM architecture. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 615–626, February 2013.

[360] Clark D French. "one size fits all" database architectures do not work for dss. In *Proceeding of the Second Joint WOSP/SIPEW International Conference on Performance Engineering - ICPE '11*, SIGMOD '95, 1995.

[361] Micron System Power Calculator for SDRAM devices. https://www.micron.com/support/tools-and-utilities/power-calc.

[362] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. To PIM or not for emerging general purpose processing in DDR memory systems. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, page 231–244, 2022.

[363] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. Query processing on tensor computation runtimes. *Proc. VLDB Endow.*, 15(11):2811–2825, sep 2022.

[364] Kevin Gaffney. ssb-baselines.

[365] Compute express link.

[366] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network i/o. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 50–59, 2005.

[367] Arm cache stashing.

[368] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. Gpu join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 55–62, 2012.

[369] Emily Furst, Mark Oskin, and Bill Howe. Profiling a gpu database implementation: a holistic view of gpu resource utilization on tpc-h queries. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, pages 1–6, 2017.

[370] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Pump up the volume: Processing large data on gpus with fast interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1633–1649, New York, NY, USA, 2020. Association for Computing Machinery.

[371] Chaemin Lim, Suhyun Lee, Jinwoo Choi, Jounghoo Lee, Seongyeon Park, Hanjun Kim, Jinho Lee, and Youngsok Kim. Design and analysis of a processing-in-dimm join algorithm: A case study with upmem dimms. *Proc. ACM Manag. Data*, 1(2), jun 2023.

[372] Fabrice Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24, 2019.

[373] Michael Jungmair, André Kohn, and Jana Giceva. Designing an open framework for query optimization and compilation. *Proc. VLDB Endow.*, 15(11):2389–2401, jul 2022.

[374] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. SQLite: Past, present, and future. *Proc. VLDB Endow.*, 15(12):3535–3547, aug 2022.

[375] Aarati Kakaraparthy, Jignesh M. Patel, Brian P. Kroth, and Kwanghyun Park. VIP hashing: Adapting to skew in popularity of data on the fly. *Proc. VLDB Endow.*, 15(10):1978–1990, jun 2022.

[376] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. Quantifying tpc-h choke points and their optimizations. *Proc. VLDB Endow.*, 13(8):1206–1220, apr 2020.

[377] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, et al. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–197, 2013.

[378] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. Schedule trees. In *International Workshop on Polyhedral Compilation Techniques, Date: 2014/01/20-2014/01/20, Location: Vienna, Austria*, 2014.

[379] BLAS: Basic linear algebra subprograms.

[380] Brent Keeth and R Jacob Baker. *DRAM circuit design: a tutorial*. IEEE, 2001.

[381] National Human Genome Research Institute. The cost of sequencing a human genome. https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost.

[382] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009.

[383] Rasool Sharifi and Zainalabedin Navabi. Online profiling for cluster-specific variable rate refreshing in high-density dram systems. In *2017 22nd IEEE European Test Symposium (ETS)*, pages 1–6, 2017.

[384] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks, 2018.

[385] Matthias Hess, Alexander Sczyrba, Rob Egan, Tae-Wan Kim, Harshal Chokhawala, Gary Schroth, Shujun Luo, Douglas S Clark, Feng Chen, Tao Zhang, et al. Metagenomic discovery of biomass-degrading genes and genomes from cow rumen. *Science*, 331(6016):463–467, 2011.

[386] Adina Chuang Howe, Janet K Jansson, Stephanie A Malfatti, Susannah G Tringe, James M Tiedje, and C Titus Brown. Tackling soil diversity with the assembly of large, complex metagenomes. *Proceedings of the National Academy of Sciences*, 111(13):4904–4909, 2014.

[387] Elizabeth T Cirulli and David B Goldstein. Uncovering the roles of rare variants in common disease through whole-genome sequencing. *Nature Reviews Genetics*, 11(6):415–425, 2010.

[388] Claudia Gonzaga-Jauregui, James R Lupski, and Richard A Gibbs. Human genome sequencing in health and disease. *Annual review of medicine*, 63:35–61, 2012.

[389] Cynthia C Steiner, Andrea S Putnam, Paquita EA Hoeck, and Oliver A Ryder. Conservation genomics of threatened animal species. *Annu. Rev. Anim. Biosci.*, 1(1):261–281, 2013.

[390] Richard Frankham. Challenges and opportunities of genetic approaches to biological conservation. *Biological conservation*, 143(9):1919–1927, 2010.

[391] Mark T Ross, Darren V Grafham, Alison J Coffey, Steven Scherer, Kirsten McLay, Donna Muzny, Matthias Platzer, Gareth R Howell, Christine Burrows, Christine P Bird, et al. The dna sequence of the human x chromosome. *Nature*, 434(7031):325, 2005.

[392] Marcel Margulies, Michael Egholm, William E Altman, Said Attiya, Joel S Bader, Lisa A Bemben, Jan Berka, Michael S Braverman, Yi-Ju Chen, Zhoutao Chen, et al. Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–380, 2005.

[393] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W. Keckler. Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. *SIGARCH Comput. Archit. News*, 44(3):204–216, June 2016.

[394] Jintao Meng, Bingqiang Wang, Yanjie Wei, Shengzhong Feng, and Pavan Balaji. Swap-assembler: Scalable and efficient genome assembly towards thousands of cores. *BMC Bioinformatics*, 15(Suppl 9), 2014.

[395] Jintao Meng, Sangmin Seo, Pavan Balaji, Yanjie Wei, Bingqiang Wang, and Shenzhong Feng. Swap-assembler 2: Optimization of de novo genome assembler at extreme scale. *2016 45th International Conference on Parallel Processing (ICPP)*, 2016.

[396] G. Rizk, D. Lavenier, and R. Chikhi. Dsk: K-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, 2013.

[397] Yatish Turakhia, Gill Bejerano, and William J. Dally. *Darwin: A Genomics Co-Processor Provides up to 15,000X Acceleration on Long Read Assembly*, page 199–213. Association for Computing Machinery, New York, NY, USA, 2018.

[398] Daichi Fujiki, Shunhao Wu, Nathan Ozog, Kush Goliya, David Blaauw, Satish Narayanasamy, and Reetuparna Das. Seedex: A genome sequencing accelerator for optimal alignments in subminimal space. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 937–950, 2020.

[399] L. Jiang and F. Zokaee. Exma: A genomics accelerator for exact-matching. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 399–411, Los Alamitos, CA, USA, mar 2021. IEEE Computer Society.

[400] Daichi Fujiki, Arun Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. Genax: A genome sequencing accelerator. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 69–82, 2018.

[401] 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. Fpga accelerated indel realignment in the cloud. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 277–290, 2019.

[402] 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. Genesis: A hardware acceleration framework for genomic data analysis. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 254–267, 2020.

[403] Yoohyuk Lim, Jaemin Lee, Cassiano Campes, and Euiseong Seo. {Parity-Stream} separation and {SLC/MLC} convertible programming for life span and performance improvement of {SSD}{RAIDs}. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[404] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, Jaehong Kim, Jaechun Park, Ki-Whan Song, Ki-Tae Park, Sangyeun Cho, Hwaseok Oh, Daniel D.G. Lee, Jin-Hyeok Choi, and Jaeheon Jeong. A flash memory controller for 15s ultra-low-latency ssd using high-speed 3d nand flash with 3s read time. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pages 338–340, 2018.

[405] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1994.

[406] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. *2018 IEEE Symposium on Security and Privacy (SP)*, pages 36–52, 2018.

[407] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18, 2017.

[408] Shayan Moini, Shanquan Tian, Daniel Holcomb, Jakub Szefer, and Russell Tessier. Power side-channel attacks on bnn accelerators in remote fpgas. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(2):357–370, 2021.

[409] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 515–532, Santa Clara, CA, August 2019. USENIX Association.

[410] Johanna Sepulveda, Cezar Reinbrecht, and Jean-Philippe Diguet. Security aspects of neuromorphic mpsocs. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, New York, NY, USA, 2018. Association for Computing Machinery.

[411] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, 2014.

[412] Sachhidh Kannan, Naghmeh Karimi, Ozgur Sinanoglu, and Ramesh Karri. Security vulnerabilities of emerging nonvolatile main memories and countermeasures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(1):2–15, 2015.

[413] Fatih Gül. Addressing the sneak-path problem in crossbar rram devices using memristor-based one schottky diode-one resistor array. *Results in Physics*, 12:1091–1096, 2019.

[414] Mohammed Affan Zidan, Hossam Aly Hassan Fahmy, Muhammad Mustafa Hussain, and Khaled Nabil Salama. Memristor-based memory: The sneak paths problem and solutions. *Microelectronics Journal*, 44(2):176–183, 2013.

[415] Yuval Cassuto, Shahar Kvatinsky, and Eitan Yaakobi. Sneak-path constraints in memristor crossbar arrays. In *2013 IEEE International Symposium on Information Theory*, pages 156–160, 2013.

[416] Jiale Liang and H.-S. Philip Wong. Cross-point memory array without cell selectors—device characteristics and data storage pattern dependencies. *IEEE Transactions on Electron Devices*, 57(10):2531–2538, 2010.

[417] Shinhyun Choi, Scott H. Tan, Zefan Li, Yunjo Kim, Chanyeol Choi, Pai-Yu Chen, Hanwool Yeon, Shimeng Yu, and Jeehwan Kim. Sige epitaxial memory for neuromorphic computing with reproducible high performance based on engineered dislocations. *Nature Materials*, 17(4):335–340, 2018.

[418] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1322–1333, New York, NY, USA, 2015. Association for Computing Machinery.

[419] Shuangchen Li, Alvin Oliver Glova, Xing Hu, Peng Gu, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. Scope: A stochastic computing engine for dram-based in-situ accelerator. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 696–709, 2018.

[420] Jennifer Hasler and Bo Marr. Finding a roadmap to achieve large neuromorphic hardware systems. *Frontiers in Neuroscience*, 7, 2013.

[421] Mark Randolph and William Diehl. Power side-channel attack analysis: A review of 20 years of study for the layman. *Cryptography*, 4(2):15, 2020.

[422] Deepak Kadetotad, Zihan Xu, Abinash Mohanty, Pai-Yu Chen, Binbin Lin, Jieping Ye, Sarma Vrudhula, Shimeng Yu, Yu Cao, and Jae-sun Seo. Parallel architecture with resistive crosspoint array for dictionary learning acceleration. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 5(2):194–204, 2015.

[423] Yandong Luo, Xiaochen Peng, and Shimeng Yu. Mlp+neurosimv3.0: Improving on-chip learning performance with device to algorithm optimizations. In *Proceedings of the International Conference on Neuromorphic Systems*, ICONS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[424] Manu Rastogi, Vaibhav Garg, and John G. Harris. Low power integrate and fire circuit for data conversion. In *2009 IEEE International Symposium on Circuits and Systems*, pages 2669–2672, 2009.

[425] Jonathan Tapson and André van Schaik. An asynchronous parallel neuromorphic adc architecture. In *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2409–2412, 2012.

[426] Manu Rastogi, Alexander Singh Alvarado, John G. Harris, and José C. Príncipe. Integrate and fire circuit as an adc replacement. In *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*, pages 2421–2424, 2011.

[427] Rozhin Yasaei, Sina Faezi, and Mohammad Abdullah Al Faruque. Hardware trojan power em side-channel dataset, 2021.

[428] Juan Ai, Zhu Wang, Xinping Zhou, and Changhai Ou. Improved wavelet transform for noise reduction in power analysis attacks. In *2016 IEEE International Conference on Signal and Image Processing (ICSIP)*, pages 602–606, 2016.

[429] Deepak Kadetotad, Zihan Xu, Abinash Mohanty, Pai-Yu Chen, Binbin Lin, Jieping Ye, Sarma Vrudhula, Shimeng Yu, Yu Cao, and Jae-sun Seo. Parallel architecture with resistive crosspoint array for dictionary learning acceleration. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 5(2):194–204, 2015.