THE UNIVERSITY OF CHICAGO

HIGH-PERFORMANCE ARCHITECTURES FOR DATA CENTER COMPUTATIONAL STORAGE

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
CHEN ZOU

CHICAGO, ILLINOIS
DEC 7TH 2022

Dedicated to my parents Linquan and Youhong, for their endless love and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

ABSTRACT

The dramatic growth in the importance of large-scale data analytics has driven the transformation of data center storage from hard disk drives to solid-state drives (SSD). Central to increased capability is the rapid growth in SSD/flash bandwidth and associated compute requirements, resurrecting the question of how to distribute compute across CPUs and storage resources.

In this dissertation, we address three fundamental questions on architecting general-purpose computational storage. First, what are key opportunities for computational storage acceleration? Second, what SSD architecture provides the most efficient support for computational storage? Third, what computational storage processor architecture enables high performance that can match the continued rapid improvement in flash bandwidth?

Based on a broad survey of computational storage proposals and research, we show that function properties of 'data size change', 'offload direction' and 'vectorizable' determine the system efficiency and performance benefits of computational SSD offloads and thus determines the priority of offloading. Common properties including streaming access and variable-width values exposed by first-priority functions call for architectural support.

Existing computational SSD architectures suffer from poor cost scaling. This is exacerbated by the continued improvement in flash array bandwidth, creating a SSD DRAM bottleneck. We propose the ASSASIN SSD architecture, which provides a unified set of compute engines between SSD DRAM and the flash array to eliminate the bottleneck by enabling direct computing on flash data streams with streambuffers. ASSASIN thus delivers 1.5x - 2.4x speedup for various computational SSD offloads along with 2.0x power efficiency and 3.2x area efficiency.

Existing processor architectures suffer from low datapath efficiency when computing on variable-width values for requiring padding each value to 32/64 bits. Variable-width values are central to coding and storage efficiency, and thus critical for computational storage. We propose VarVE, a vector instruction set architecture extension that provides native variable-width value vector support to compute directly without padding. VarVE delivers 1.3x - 5.4x speedup over ARM's current best, scalable vector extension (SVE), on popular file system and database computational SSD kernels by achieving higher datapath efficiency. VarVE builds on the vector-length agnostic (VLA) approach, which is gaining widespread adoption. As a result, VarVE has broad potential impact as a general SIMD extension for all processors, increasing datapath efficiency and mitigating the SIMD instruction count explosion.

CHAPTER 1

DISSERTATION STATEMENT

Computational storage has demanding performance and low-power requirements that are not met by conventional CPU core and SSD architecture designs. Desirable computational storage architectures can not just meet these requirements, but must also do so with flexible and general-purpose programmability. We show this is possible in three parts:

1. Study of proposed and extant applications shows that the most promising offloads are those reduce data size on read-path or increases data size on write-path and are vectorizable. Further, the properties of most promising computational storage applications create opportunities for better core and system architecture design.

2. With computational storage applications dominated by streaming data access, SSD architecture and organization are critical for cost-efficient performance. ASSASIN, a novel SSD architecture providing direct computation on data streams between flash channels and SSD DRAM with a streaming-oriented memory hierarchy, is essential to provide cost-efficient and power-efficient performance.

3. With variable-width values abundant in storage data, efficient computation on them is critical for high performance. VarVE, a variable-width value vector instruction set architecture is flexible, high-efficiency, and high-performance for a wide variety of computational storage workloads. This VarVE extension to the fixed-width vector architecture provides aforementioned benefits not only for computational storage, but also for general-purpose CPUs.

In the computational storage community, many advocate customized hardware approaches, including both special function ASICs and FPGAs, these approaches are limited by their inflexibility and inability to catalyze new applications. In this dissertation, we present workload, SSD architecture, and computational storage processor architecture results that demonstrate a focused but generally programmable approach can meet the challenging performance requirements. We believe this is the most promising path forward to delivering the widespread benefits of computational storage.

CHAPTER 2

INTRODUCTION

2.1 Storage is the Heart of Data Centers

The past two decades have seen explosive growth in large-scale data, ranging from internet services (web search, e-commerce, social networking, music and video streaming), scientific research (sequence search, high-energy physics, weather forecasting, etc.) to nearly every aspect of the economy and society through the explosion of data science and analytics. More recently, the success of machine learning in deriving functions from a large amount of data has accelerated this trend. These data sets range from a few hundred terabytes to hundreds of petabytes, largely transcending the storage capacity of a single server. Further, to provide timely insights and analysis on top of these massive data, massive compute resources grow rapidly and accordingly, and co-locate with the data storage for efficiency. As a result, gigantic global networks of data centers [1, 9, 14, 32] each with thousands to millions servers are rapidly growing.

The efficient storage and processing for these massive amount of data is the raison d'etre of 'big data' and 'cloud', i.e. the modern distributed systems hosted in these data centers. As shown in Figure 2.1, distributed block storage services, e.g. GFS [62]/S3 [23], sit at

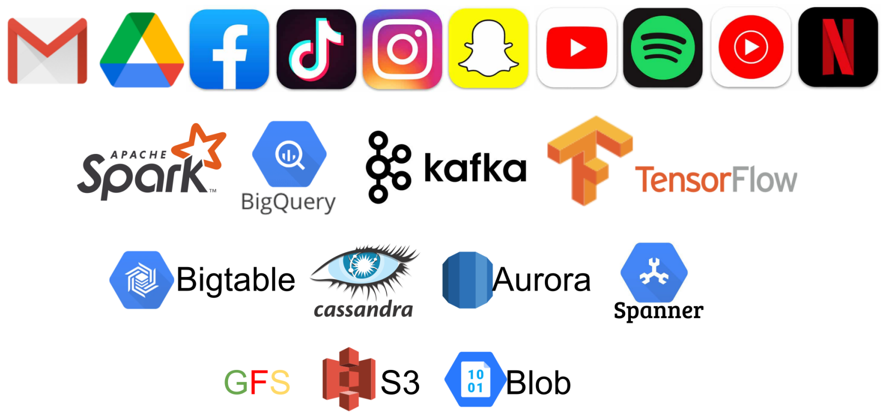


Figure 2.1: Storage sits at the heart of modern distributed systems

the core of the distributed systems as the backbone to provide durability, reliability and availability for the massive amount of data. On top of the backbone storage, efficient and application-facing management of the data are provided by by different midd-level storage services, including key-value storage (Bigtable [52], Cassandra [83], etc.), relational storage (Spanner [54], Aurora [109], etc.) and so on to support high-performance high-bandwidth and low-latency retrieval for massively parallel distributed computing middle-ware, e.g. Spark [114], Kafka [11] etc. Applications are built on top of these computing middleware to provide fast and reliable services for millions to billions of users across the globe.

2.2 Challenges from the Growing Flash Bandwidth

While rotating disks (HDDs) remain cheaper capacity storage, the continued scaling of NAND flash has reduced the cost of SSD capacity sufficiently such that its superior bandwidth and IOPS performance make it a mainstay of modern data centers. Leading enterprise SSDs can reach 6.95 GB/s and 900K IOPS [20], and consumer SSDs exceed 7 GB/s and 690K IOPS [24] at a lower price tag. However, the computing and interconnect resources coupled with the storage system are not ready for this bandwidth jump. Quantitatively, it is common for a storage server to hold 20 HDDs, delivering 3GB/s read throughput. Switching to 20 SSDs would deliver 120 GB/s instead, a 40x bandwidth capacity increase. Compute resources in data centers strive to match this increase. But it is costly and architecturally difficult to scale computing resources at least forty times to match the storage bandwidth improvement.

Moreover, cloud data centers have evolved into separate managements of compute clusters and storage clusters for separate scaling and management as shown in Figure 2.2, because storage devices and computing resources have different failure models, lifetime as well as scalability. Simply scaling the computing capabilities at the compute cluster would result in network bandwidth bottlenecks when fetching data from storage nodes with twenty times higher data rates. A natural idea comes alone: Part of the computation should be moved

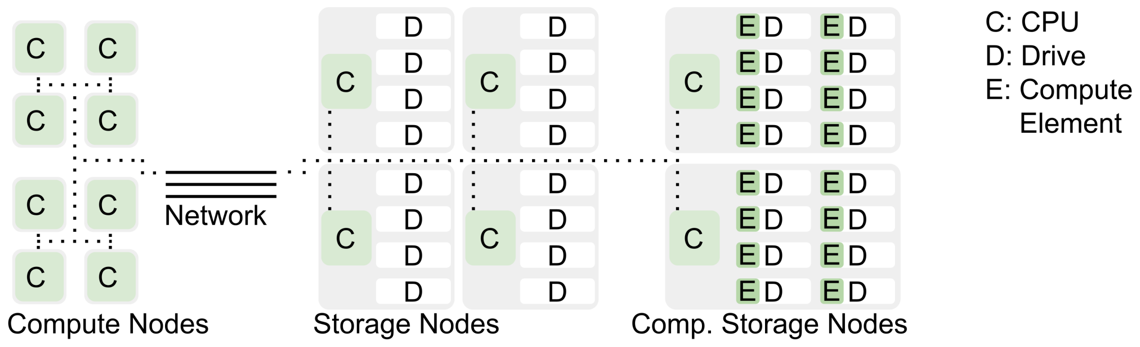


Figure 2.2: Modern data center architecture: Separate compute and storage

into the storage cluster to reap the storage bandwidth and avoid network bottlenecks.

How to best divide computation across compute and storage has long been important to system performance, cost-effectiveness, and the subject of research going back to the 1990s [39, 98], when HDDs scaled to higher bandwidth and microprocessors were becoming largely available at low costs. However, with later aggressive scaling of microprocessors to multi-core architectures, and per HDD bandwidth capacity stagnating at hundreds of megabytes, ‘active disks did not blossom into the mainstream storage systems.

2.3 Computational SSDs to the Rescue

But the recent rapid increases in storage device bandwidth in terms of flashes and non-volatile memories has reopened a set of critical research questions about what kinds of computation is beneficial to be added to storage, how to redistribute work and rebalance systems, etc. As a result, an explosion of innovation [80, 109] and research [53, 59, 73, 75, 82, 88, 108, 113] is exploring computational storage [27] as shown in Figure 2.2.

Flash chip architecture is highly parallel with a wide interface (see Section 3.2), further combined with chip-level parallelism, the flash array in SSDs possess huge excessive bandwidth [59], not exposed by the SSD interface and thus guarantees further SSD bandwidth improvement for the upcoming decade. On the other hand, with the ending of Dennard scaling and the slowdown if not the end of the moore’s law, microprocessor performance improvement has been mostly stagnating. These two trends warrant the development of computational storage, where compute elements are brought into SSDs, as shown in Figure 2.2, to exploit the surplus internal bandwidth capacity. Additional compute elements in a SSD drive can carry out computation on data coming from or going to the flash array. The goal is to bring synergic system benefits, including early data filtering, late data incrementing and automatic computation capacity scaling benefits that would reduce traffics on the storage interface and thus address the interconnect and compute bottlenecks.

Presuming data center operators want to preserve the modern data center architecture (as shown in Figure 2.2) and associated usage model (how the compute resources are allocated and how the storage is shared), computational storage enriches storage systems with broader capabilities to alleviate the compute and interconnect bottlenecks in modern data centers, but also invites numerous questions.

First, what kinds of operations are suitable to be offloaded to the compute engines? Although existing studies considered several workloads for offload, without systematic understanding of offload benefits, many of them fall into local optimum of just seeking compute speedup. Missed by much existing work considering computational storage, the criterion here is really "Is computational storage the most efficient approach to implement this workload" rather than "Does computational storage increase application performance". As for some applications, despite being acceleratable by the computational storage, compute-side acceleration may provide even higher performance and/or system efficiency. A good example is compression, different computational storage work [41, 68, 73, 86, 99, 102, 108] considered offloading compression to storage before data is written to flashes. Although acceleratable through special hardware, compression offloaded to storage increases the data traffic between compute and storage as uncompressed data have to travel to storage to be compressed. Compute-side acceleration would be a better solution for compression in terms of system efficiency, as data traveling on the link between compute and storage are compressed which leads to

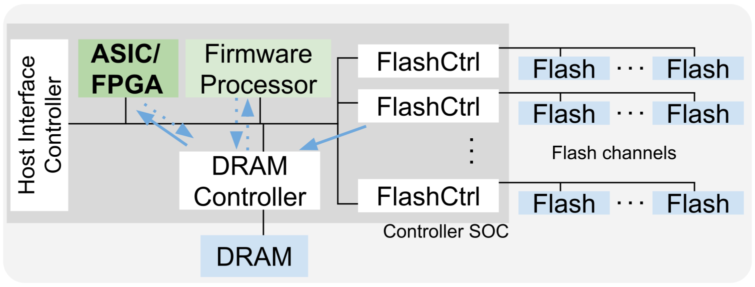


Figure 2.3: Computational storage drives considered in existing work

lower bandwidth utilization and lower latency. As a result, the first research question to be understood is how to systematically consider workloads for computational storage offload. A good understanding of this workload question would also later help shape architectures.

Second, what SSD architecture is beneficial for the integration of computing elements? State-of-the-art work considers using additional cores in the firmware processor [65, 75, 78, 82, 103] for in-SSD computation or adding other computing elements (e.g. FPGA [73, 99, 113] or ASIC [41, 79, 90]) as peers of the firmware processor. As shown in Figure 2.3, the SSD DRAM, which originally serves as a buffer for firmware data structures and storage data, would additionally serve the in-SSD compute traffic from the added compute elements. Not only would SSD DRAM become a memory wall when hit with these different requests, but the increase in requirements on DRAM capacity and bandwidth would also constitute as overhead to the SSD drive, raising the cost. It is worth considering a new SSD organization that is efficient for computational storage and addresses the memory bottleneck.

Third, what kinds of architecture features are beneficial for compute elements to match continuously improving flash bandwidth? To enable general support across a broad set of storage-intensive applications, in our vision, programmability is necessary for the computing elements. Much existing work considered employing general-purpose cores for in-SSD compute. However, this just shifts compute bottlenecks from the compute cluster into SSDs, because the power envelope of an SSD device limits the scaling out of general-purpose cores to provide enough computing power to match the flash bandwidths, especially for compute-intensive offloads such as parsing or cryptography. Further, if employing SIMD units, the compute performance improvement is hindered by the deficiency of existing SIMD ISAs to specify and exploit fine-grained parallelism for variable-width data elements which are extremely common for data in storage. Thus, a new SIMD ISA and its efficient implementation addressing these issues are worth investigating.

2.4 The Problem

To summarize, high performance is critical for computational storage, but it must be delivered with high flexibility to support diverse offloads and within the power budget of an SSD which is now becoming the backbone of the data centers. Existing approaches present a trade-off between performance and flexibility. ASIC approaches deliver high performance but target a single application or domain and hard if not impossible to re-purpose for other applications. Approaches employed general-purpose core can target different applications, but the in-SSD compute performance it provides cannot match the continuously improving flash array bandwidth.

The approach of this dissertation project is to address a set of focused questions. We aim to explore what characteristics a successful computational storage system should have in terms of the criterion on what workloads to be offloaded to computational storage, efficient computational SSD architectures that would address the SSD memory bottleneck, and high-performance computational storage processor architectures to match in-storage compute performance with the flash bandwidth. A feasible solution to the above problems should be general-purpose to target diverse offload, high performance, but lightweight that hides the additional power under the power envelope of an SSD. And it should also provide scalable performance that matches the continuously improving flash bandwidths.

2.5 Contributions

The contributions of the dissertation are threefold:

- A classification methodology based on workload properties to identify, and characterize most promising workloads for computational storage.
- ASSASIN: An SSD architecture providing direct computation data streams between flash channels and SSD DRAM, with a streaming-oriented memory hierarchy. It allows flash data to bypass SSD DRAM, eliminating the memory wall in SSD.
- VarVE: An instruction set architecture with native variable-width value vector support. The ISA enables high-performance and efficient processing of variable-width values and the exploitation fine-grained data-level parallelism for computational storage processors. And the ISA is also applicable and beneficial for general-purpose CPUs.

2.6 Organization of the Dissertation

The rest of this dissertation is organized as follows. First we provide backgrounds on SSD architectures and SIMD ISAs in Chapter 3, to prepare the SSD architecture and processor architecture discussions for computational SSDs in data centers. The overview of the dissertation project is provided in Chapter 4, where the research questions and the approaches to solve them are highlighted. The landscape of computational SSD and SIMD ISA is covered in Chapter 5. Then, our research to answer the hanging questions on the workload properties and classification, the SSD architecture with stream computing support to address the in-SSD memory wall, and the VarVE-full instruction set architecture for computational storage processors with support for variable-width data elements are discussed in Chapter 6, Chapter 7, Chapter 8 respectively. We discuss the integration of our high performance SSD and processor architectures and the performance results after putting everything together in Chapter 9. And then we summarize the dissertation and identify future directions in Chapter 10.

CHAPTER 3

BACKGROUND

3.1 Solid-state Drive Architecture

As shown in Figure 3.1, an SSD is composed of several flash chips organized in multiple channels, a DRAM chip and a controller chip. The controller chip comprises a firmware processor, a host interface controller, a DRAM controller and one flash controller for each flash channel.

The firmware processor runs the flash translation layer (FTL). FTL maintains the mapping between logical block addresses and physical block addresses. A physical block address specifies a physical location composed of a flash chip ID and the specific location inside the chip. FTL would consider both flash's read/write/erase granularity and the wear-leveling.

The flash chips organized in multiple channels are managed by the flash controllers, one per channel. The firmware processor would issue requests to the flash controller in charge to access a specific flash chip. And the flash controller would respond with the required flash page if needed. The flash chips of the same channel share the same bus but operate independently. It is firmware and flash controllers' responsibility to exploit the operation interleaving opportunities among the flash chips sharing the same channel, analogous to the notion of ranks and rank-level parallelism in the memory system.

There is a DRAM chip in high-performance SSDs serving as a buffer to store both page data relevant to recent requests from/to flash controllers and request queues between the firmware and flash controllers. It also buffers FTL-related data structures for the firmware.

The host interface controller implements the storage protocol the SSD would use to communicate with the host, e.g. SATA, NVMe, etc. The firmware running on the firmware processor would pull requests and send responses from/to the host via this host interface controller.

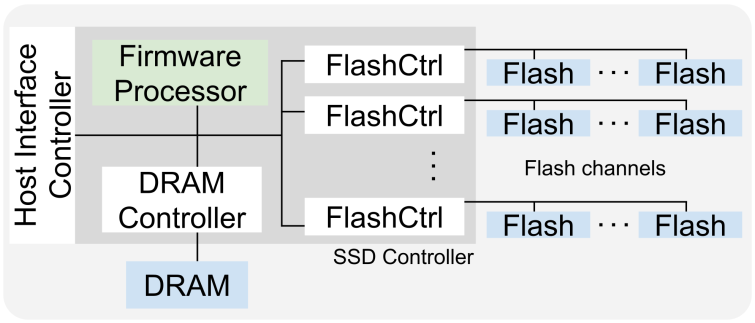


Figure 3.1: SSD architecture diagram

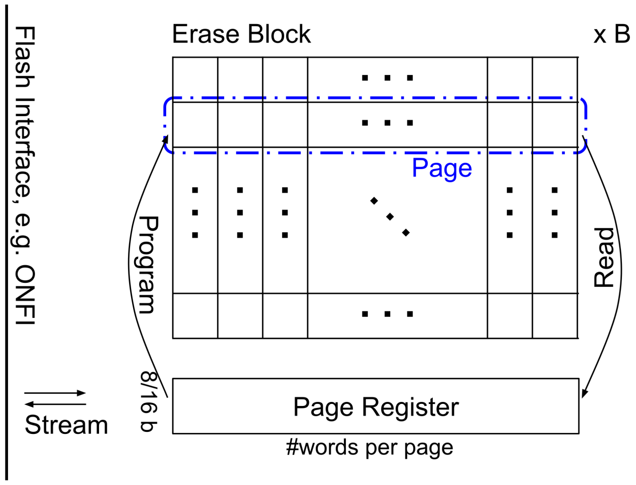


Figure 3.2: NAND flash

3.2 NAND Flash

We look deeper into the underlying NAND flash internals and flash interface. This is detailed in Figure 3.2. NAND flash is composed of multiple (B in the figure) erase blocks. Each block is an array of floating-gate MOSFET transistors sharing the substrate. The transistors on the same row sharing a selection line form the smallest read or program granularity which is called an flash page.

The NAND flash interacts with external entities with the page register through the flash interface. During a page read process, the selected row is first loaded into the page register, and the page streams out in a word-by-word (either 8b or 16b) fashion. During a page write process, data stream into the page register first, and the page register is used to program a specific row in the array. ONFI [55] is the standardization effort on the flash interface. The newest version as of writing is 5.0 which supports up to 2400 MT/s at the interface.

3.3 SIMD and Vector-length Agnostic SIMD

SIMD instruction set extensions are based on a straightforward premise. Many workloads expose data-level parallelism that the same operation sequence would be applied to a set of data. Internalizing an execution mechanism in hardware that applies the same instructed operation to a vector of values and providing this semantic interface in the ISA allows programmers to efficiently specify and exploit such data-level parallelism, delivering speedup and power efficiency gains.

Earliest SIMD extensions like MMX [94] stems from multimedia workloads. They are fine approaches at birth, limited by silicon technology then. Later extensions gradually grow the datapath width (e.g. AVX, AVX2 and then AVX-512 [3]), granted by the more and more silicon resources available via technology scaling. Although, this leads to instruction duplications. New instructions are added which perform the same operation but with different datapath bitwidths. And old instructions are kept for compatibility. Further, programs have to be rewritten for a newer wider SIMD extensions. And, at the same time, managing different versions of SIMD programs to match specific hardware capabilities is a cumbersome task.

Vector-length agnostic (VLA) SIMD ISAs like SVE [105] and RVV [22, 93] were proposed to address the above issues. In VLA SIMD, the datapath width (i.e. vector length) is determined at runtime when reported by hardware. The semantics of each VLA SIMD instruction (number of elements processed by an instruction) as well as the strip mining process both follow this runtime-determined datapath width. VLA SIMD not only addresses the instruction duplication issue that only one SIMD instruction is needed for various datapath widths for each operation, but also allows the same program binary to be reused

```
void sve_baxpy(int N, uint8_t a, uint8_t* output,
    const uint8_t* xdata, const uint8_t* ydata) {
  for (int i = 0; i < N; i += svcntb()) {
    svbool_t pred = svwhilelt_b8(i, N);
    svuint8_t x = svld1_u8(pred, xdata);
    svuint8_t y = svld1_u8(pred, ydata);
    svuint8_t z = svmul_m_u8(pred, x, a);
    svst1_u8(pred, output, svadd_u8_m(pred, z, y));
    xdata += svcntb();
    ydata += svcntb();
    output += svcntb();
  }
}
```

Figure 3.3: aX+Y for byte vectors in SVE: an example of VLA

on different hardware with different datapath widths, eliminating the mundane management process.

Figure 3.3 shows an VLA example that performs linear combination of two vectors x and y with coefficient a. In each iteration, it loads two vector strips from memory, perform vectorized linear combination of the strips and store the result back to memory. The program (or compiled binary) can run on hardware with different datapath widths (vector lengths), which is reflected by 'svcntb()'. On hardware with wider datapath, the vector strips are longer, more elements processed in one iteration, and less iterations.

CHAPTER 4

PROJECT OVERVIEW

In this chapter, we provide an overview of the dissertation project. We dissect the problem of architecting computational storage in data centers into three integral parts:

- Computational storage workload and offload opportunities
- Computational storage drive architecture
- Computational storage processor architecture

For each part, related research questions are discussed and analyzed. And the approaches to answer these questions are described.

4.1 Computational Storage Workload and Offload Opportunities

4.1.1 Research Questions

In the first part of the thesis project, the goal is to identify opportunities of system efficiency improvement and compute speedups when offloading kernels to computational storage and provide guidelines on the support priority among different workloads, as depicted in Figure 4.1. This would not only shape the contexts and roadmaps for designing a computational storage processor/accelerator, but also help application designers to make decisions whether to employ computational storage and which part to offload.

Here, the specific criterion is really ‘Is computational storage the most efficient approach to implement this workload’ rather than ‘Does computational storage increase application performance’. As for some applications, despite being acceleratable by computational storage, compute-side acceleration may provide even higher system efficiency and/or performance. Moreover, we hope that the answers to the aforementioned question could apply to workload

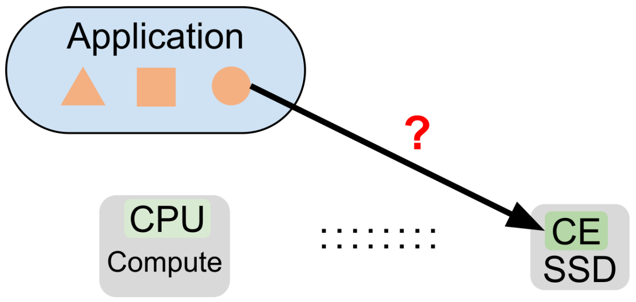


Figure 4.1: What to offload?

in the foreseeable future, not just the ones that are currently considered. As a result, although diverse workloads may exhibit different computation structures and arithmetic characteristics, the goal would be distilling common properties of workload that would justify offloading to computational storage.

To summarize, the key research agenda around the computational storage workload is to systematically characterize computational storage offload opportunities.

4.1.2 Approach

Our approach to answering this question is to perform an extensive survey to cover the workload considered for computational storage in existing research or industry products. The results of the survey represent a cross-section of computational storage studies. To ensure the soundness of our answer, we conduct a thorough characterization of each workload along with the analysis of the corresponding efficiency of each offload scenario. These results would give birth to a classification based on analyzed workload properties through careful synthesis. The classification would provide a general perspective on computational storage opportunities across both current and future workload candidates. Further, this classification would, in turn, define the workload corpus that our later design on computational storage drive architecture and computational storage processor architecture should prioritize and accelerate.

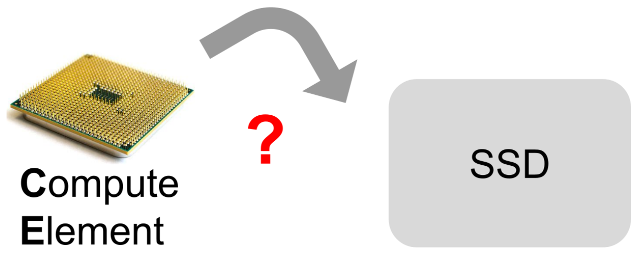


Figure 4.2: How to efficiently integrate compute into SSDs?

4.2 Computational SSD Architecture

4.2.1 Research Questions

The second part of the thesis project focuses on how to efficiently integrate compute elements into SSDs, as shown in Figure 4.2. State-of-the-art computational SSDs [65, 82, 102], as shown in Figure 4.3, employ additional compute elements which are the peers to the processor running SSD firmware to carry out the in-SSD computation. These elements all use SSD DRAM as their main memory. As a result, flash data have to first be fetched to SSD DRAM, waiting to be fetched by compute processor/cores. And in-SSD computation results have to be written to SSD DRAM before transferring to host or flash. This inefficient SSD architecture brings memory bottleneck and data access latency issues for computational storage, aggravating with the continuous improvement of flash bandwidths. As a result, we are in search of solid-state drive architectures that can eliminate the aforementioned bottlenecks and, at the same time, efficiently support the promising workloads identified in Chapter 6.

The research question regarding computational storage drive architecture is how should compute elements interact with flash data, firmware and SSD DRAM to enable low latency data access and low memory bandwidth overhead.

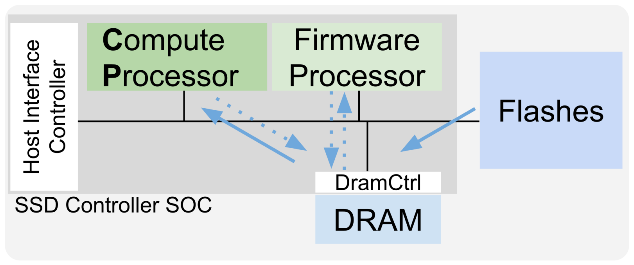


Figure 4.3: Organization of state-of-the-art computational SSDs

4.2.2 Approach

The approach is to systematically consider the needs of compute elements and how should they interact with flash data. Compute elements' needs of data access come from a thorough survey and characterization of offload opportunities from Chapter 6. Careful analysis is conducted to sift through different data access, looking for opportunities for innovative interaction mechanisms. For soundness, evaluations of different interaction mechanisms between compute elements and flash data are conducted to understand pros and cons, and to distill general knowledge on how a computational storage SSD should be architected to address the memory wall problem and the data access latency issues.

4.3 Computational Storage Processor Architecture

4.3.1 Research Questions

With a clear context of workload and SSD architecture, the third part of the thesis centers around the in-SSD processor architecture, as shown in Figure 4.4. What processor architecture features would bring compute performance that can match and exceed continuously improving flash bandwidth? At the same time efficiency and the flexibility to support different workloads should be maintained, if the acceleration is to deliver the greatest benefit.

Existing computational storage approaches [65, 75, 82, 90, 99, 113] exploit task- and block-level parallelism as shown in Figure 4.5, through scaling out the number of cores

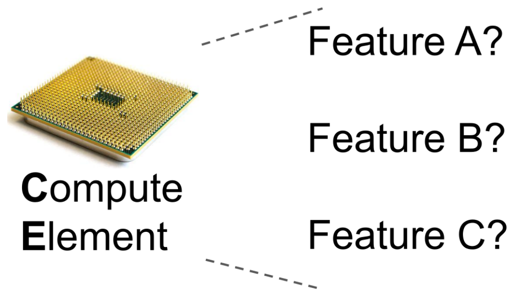


Figure 4.4: What architecture features bring high performance?

available to compute offloaded tasks inside storage. However, none of these approaches feature software programmability combined with efficient support for byte-level parallelism. Recent efforts have begun to add vector extensions to popular embedded architectures such as ARM [30] or RISCV [22]. However, these vector-length agnostic SIMD, to date, do not support sub-byte or variable-width value parallelism. On the other hand, storage data are diverse, usually packed, featuring plenty of variable-width values. Thus, another important challenge is how to extend the benefits of SIMD acceleration for packed and variable width-values stored in SSDs.

To sum up, the fundamental challenge is for an instruction set architecture to simultaneously enable efficient handling of variable-sized data elements (as exemplified in Figure 4.6) and improve datapath efficiency. Thus, the key research questions around computational storage processors are:

- What are the key elements of instruction-set architecture that enable both flexible programmability across storage workload and efficient exploitation of variable-width (sub-byte) parallelism and dependency resolution for high performance?
- What are the key design and structure elements of a microarchitecture that would efficiently implement the novel ISA?

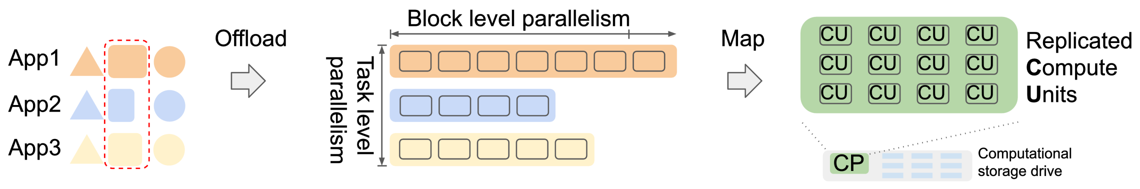


Figure 4.5: Compute unit replication for task- and block-level parallelism

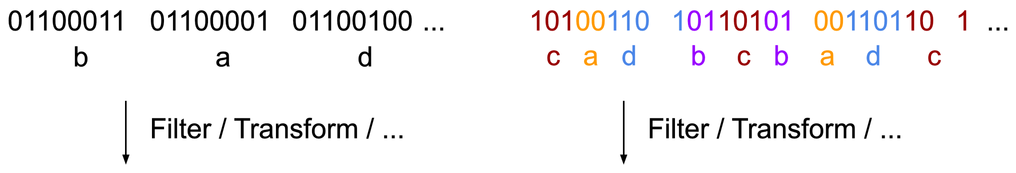


Figure 4.6: Fixed-width elements and variable-width elements

4.3.2 Approach

Our approach is to systematically explore the design of a new class of SIMD operations that exploit byte and sub-byte -level parallelism, including parallelism inside a vector of differently-sized data elements, all of which which are common in storage workloads. To ensure soundness, we characterize vastly different ways of handling variable-width data elements, including first unpack/pack these elements from/to byte-aligned elements and process them with conventional SIMD instructions, or design new SIMD instructions that directly compute on logical vectors comprised of variable-width values. These methods are compared with each other along with the conventional SIMD approach to understand the pros and cons of each. From these understandings, we will be able to distill general knowledge on handling variable-width values and exploit byte/sub-byte -level parallelism for computational storage workload.

CHAPTER 5
RELATED WORK

5.1 Computational SSD Architecture

Pioneering work includes “Active Storage” in the 1990s [39, 98] that focused on rotating hard disks in a single system, not the modern context of shared cloud storage services with high-performance flash SSDs. More recent efforts study computational storage in SSD systems. Some propose general [40, 82, 99, 102] or application-specific (i.e. data analytics) [65, 75, 108] software architecture on top of general-purpose hardware architectures as shown in Figure 7.1. Others propose application-specific hardware and associated software architectures for deduplication [41, 81], key-value storage [53, 73], deep learning [66, 90], graph analytics [77, 87], and data analytics [79, 113, 118] in computational SSDs. Such studies showcase the benefits of offloading (comparing to non-offloading) or application-specific hardware customization, but give little insight as what computation structures/properties enables system efficiency when employing computational SSD, how to build general-purpose architecture support (with associated software architecture adaptations) based on these properties and how to efficiently integrate these architecture support, which are the essence of this dissertation.

5.1.1 General-purpose Hardware Architecture for Computational SSDs

QuerySSD [57] pioneered the database-oriented function offloads and assessed the speedups and energy benefits. ActiveFlash [108] showcases similar computational SSD potential but for data reduction functions from scientific computing workloads. Biscuit [65] advances the art with a flow-based programming model for computational SSD offload. Summarizer [82] materializes the system software architecture design including a detailed NVMe command interface. YourSQL [75] provides richer software operator support to enable the offload of all TPC-H queries. BlockIF [40] argues for a software architecture for computational storage that conforms to traditional block-oriented storage interface for increased adoption. IceClave [78] proposes a low-overhead trusted execution environment for in-SSD computations and advances on the security front.

These software and system architecture advancement are all based on the general-purpose hardware architecture for computational SSDs where compute engines are embedded-class general purpose cores computing on top of the conventional cache-DRAM memory hierarchy, as depicted in Figure 7.1, and thus amenable to the in-SSD memory wall. As would be shown in Chapter 7, ASSASIN builds on top of these software architecture innovations in terms of NVMe command adaption and block-interface conformance but advances the hardware architecture integration with efficient flash data stream handling and addresses the in-SSD memory wall.

5.1.2 Application-specific Hardware Architectures for Computational SSDs

DedupInSSD [81] proposes hardware hash acceleration for in-SSD deduplication and showcase the SSD write latency reduction and lifespan improvement. CIDR [41] proposes more detailed scalable hardware architecture for deduplication which utilizes additional scratchpads for signature management and integrates compress engines. LightStore [53] augments the general purpose architecture with a hardware network module to expose a scalable key-value storage onto the datacenter network. Caribou [73] further maps the flash management into hardware modules of cuckoo hash and slab allocation manager and also supports hardware accelerated in-SSD processing primitives like select and compression. GList [87] proposes to integrate hardware sampling unit and a PE array in the SSD board for graph learning workloads to enjoy internal excessive storage bandwidth. GrafBoost [77] employs a sort-reduce accelerator in storage for vertex-centric graph processing. Deepstore [90] integrates systolic arrays at SSD, channel and chip-level for accelerated neural network inference. Thrifty [66] proposes chip-level binary compute accelerator and SSD-level training accelerator for hyper-dimension compute training.

Despite employing application-specific hardware architecture, in all above work except DeepStore and Thrifty, compute engines source flash data via SSD DRAM, thus amenable to SSD DRAM bottleneck. And the employment of hardware acceleration modules actually further increases the bandwidth requirements of SSD DRAM, as discussed in Section 7.3.2. Although channel and chip-level compute engine integration employed by Deepstore and Thrifty has the potential of addressing the SSD memory wall, it further requires application-specific control on data (SSD page) layout for parallelism exploitation. This is fine for neural network applications featuring regularly-shaped tensors which can be easily split into the flash array, but a deal breaker for general workloads with diverse basic unit sizes. ASSASIN, as will be discussed in Chapter 7, pools compute elements at the SSD-level, and thus supports different basic unit sizes by piecing a unit form pages across channels and dies. ASSASIN addresses the memory wall while leaving the flash translation layer and the SSD interface unchanged.

5.1.3 FPGA-based Hardware Architecture for Computational SSDs

There is a middle ground of computational SSDs where an FPGA chip is integrated into the SSD board and assumes the computation responsibility [58, 99, 102]. This allows the storage vendor to either support more flexible and fast iterations through hardware reconfiguration or shift the responsibility of architecting the compute engines including both general-purpose or application-specific ones in computational SSDs to the customer, but at the cost of both increased power and silicon area compared with a fixed hardware architecture, and ease of use for requiring hardware expertise for reconfiguring the FPGA.

Both Insider [99] and Access [102] try to address the ease-of-use aspects by providing system software architecture support of file system filter [99] or pipe [102] abstractions for the FPGA computing kernels to enable easy system pipelines and embracing high-level synthesis for developing these kernels. However, because the SSD FPGA still fetches the data from SSD DRAM or even requires data to be staged in an additional FPGA DRAM [58], the in-SSD memory wall is left unaddressed.

5.1.4 Disaggregated Storage

Disaggregated storage is a different approach than computational storage. It has the advantage of separate management and scaling of compute and storage, thus allowing compute to scale out and catch up with continuously improving flash bandwidth. Good examples of disaggregated storage include JBOF with NVMe-oF [10] and Amazon EBS [5]. However, comparing to computational storage, it doesn't offer the ability of matching more effective storage bandwidth to a single compute unit (a single point of consistency), and requires higher interconnect bandwidths and also puts high burdens of packets and data processing on the compute processors.

A related trend is Open-channel SSD[47] and Zoned Namespace SSD[48] that enable better control of placement and performance by externalizing management. But they do not address the system architecture aspects (driving more storage bandwidths, reducing interconnect bandwidth requirement, offloading packet and data processing) which computational storage strives for.

5.2 Compute Element Architecture

5.2.1 SIMD ISA

SIMD instruction-set architecture exploiting data-level parallelism dates back to the ILLIAC IV [49] and later popularized by the Cray-1 vector architecture [100]. Widely-used general-purpose SIMD ISA like MMX/SSE arose from multimedia workloads in the 1980s. Incremental improvements via expanding the datapath width, i.e. AVX, AVX2, and AVX512 [3], lead to the explosion of number of instructions and binary management incompatibility issues. Vector-length agnostic (VLA) SIMDs like ARM SVE [105] or RISC-V Vector [22], allow a single program to run in a variety of different vector hardware. VarVE goes one step further advancing the VLA SIMD concept to abstract away the bitwidth of each vector element. This not only eliminates instruction duplication for different vector types but also allows more efficient use of VecReg and VecALU to process more elements per instruction, improving the datapath efficiency.

5.2.2 Matrix Extension

Matrix ISA extensions are growing to drive up parallelism exploitation and improve efficiency. AMX [72] provides native BLAS-3 operations on tiled matrices for x86. SME [26] provides for AArch64 native inner product accumulation on a matrix tile. Tensor Cores [33] on NVIDIA GPUs also exploit exposing matrix multiplication, and even consider going beyond to provide application-specific operations [116]. VarVE-full is going in the opposite direction providing a simpler interface and more flexible operations to mitigate the instruction-set explosion. It is possible to extend VarVE-full to two-dimension strip mining on variable-value matrices to support matrix multiplication.

5.2.3 Stream Computing Acceleration.

Recent work proposes different acceleration for streaming workloads. UVE [60] proposes an extension that provide general padded element streaming support, mitigating the loop control and memory indexing overhead. SparseCore [96] proposed stream ISA support that abstracts sparse data as streams of key-value pairs and provides instructions that compute on these streams. Bison-e [97] proposes mechanism that embed multiple small values from two streams into two 32b/64b integer respectively, and carry out regular integer multiplication (this technique is called binary segmentation) to get inner products or convolution of the two streams. VarVE is a more general approach that provides full SIMD operation support for variable width values as an VLA SIMD extension. Sparsity or small values could also be exploited as a special case of our variable-width value support, as shown in Section 8.3.4.

CHAPTER 6
COMPUTATIONAL STORAGE WORKLOADS AND OPPORTUNITIES

6.1 The Spectrum of Computational Storage Offloads

We survey the research literature considering offloading computations into storage systems, on storage links (network/PCIe) or inside storage devices (SSDs) [40, 41, 57, 65, 66, 69, 70, 73, 75, 77–79, 81, 82, 86, 87, 90, 92, 99, 102, 108, 109, 113]. We extract the specific offloaded functions from each system and summarize them in Table 6.1. Offload functions are shown as columns and system/literature names as rows.

File system and database domain encapsulate most of the functions considered for computational SSD offloads surveyed in our literature study. As domains providing durability and associated analysis, management as well as performance optimization capabilities, these two domains are highly contingent on the storage functionality as well as storage interface changes. As a result, they are popular domains to be involved in considerations for function relocation between compute and storage. We group the functions under their corresponding domain in the table. The table collectively represents a cross-section of system research on computational storage.

For file system domain, functions considered offloading to computational SSDs include Cryptography, Compress, Deduplicate, Erasure coding, and Replicate They provides security, efficiency, capacity-saving, and failure-recovery for the file system respectively. Existing work consider offloading these functions to enjoy compute acceleration (e.g. ASIC for Galois Field operations), parallelism exploitation (e.g. multiple compress or deduplicate FPGA kernels), constraining the storage traffic (e.g. replicate to present a eventual consistent distributed storage interface).

For database domain, functions considered are mostly for data analytics workloads,

	File system					Database					Other			
	Cryptography	Compress	Deduplicate	Erasure Coding	Replicate	Filter	Select	Parse	Statistics	Write-ahead Log Replay	Transpose	Statistical Modeling	Neural Networks	Graph Analytics
Access [102]	x	x				x								
ActiveFlash [108]		x				x			x		x	x		
Aurora [109]					x					x				
Azure [70]				x										
Biscuit [65]				x		x								
BlockIF [40]				x										
Caribou [73]		x	x		x	x	x							
CIDR [41]		x	x											
DedupInSSD [81]			x											
DeepStore [90]													x	
Glist [87]														x
Grafboost [77]														x
Ibex [113]						x	x	x						
IceClave [78]	x					x	x		x					
Insider [99]		x				x	x		x			x		
Lepton [69]		x												
MithriLog [79]						x	x	x	x					
QuerySSD [57]						x	x							
Skyhook [86]		x				x	x		x					
Summarizer [82]						x	x		x					
Thrifty [66]												x	x	
YourSQL [75]						x	x	x						

Table 6.1: Functions from different application domains proposed for computational storage

including parse, select, filter, statistics. Parse decode structures and values both specified by plain texts and structure them to database structures of binary data. Offloading Select enables column pruning to get table data out of SSDs selectively. Offloading Filter enables row-wise predicate pushdown to trim down rows. And offloading statistics tasks enable aggregation inside SSD, transferring only the collected stats out, largely reducing the data size. Finally, write-ahead logging (WAL) is from transnational DBMS rather than analytical ones. Offloading WAL replay, allows distributed SSDs storing database to contain traffics and present to the upper-level software as a highly available eventually consistent durability storage. This is similar to the Erasure coding and Replicate offloading in file system domain.

Despite numerous studies considering offloading functions to SSDs, systematic considerations of computational storage offload opportunities is sparse. This can lead to local optimum of pursuing performance increase, in lack of considerations for costs of additional computational storage engines or the alternative of compute-side acceleration (FPGAs/ASICs/GPUs). For example, multiple research promote compression offload on the write-path, based on the computation speedups delivered by ASIC or FPGA compression accelerators [41, 73, 99] or simply CPU-saving benefits [86, 108]. However, they failed to recognize that compute-side acceleration of compression could deliver the same or better acceleration and CPU-saving benefits with the same accelerators. Further, compression at the compute-side (CPU or accelerator) would make the data transfer on data center interconnect to the storage cluster as compressed data instead of original data, saving interrconnect traffics. But offloading compression to computational SSD would instead transfers uncompressed data on the interconnect, wasting interconnect traffics. In this sense, compute-side offload of compression is strictly better than offloading to computational SSD> As a result, our goal is to provide a systematic way to reason about offloading benefits for a function and determine its feasibility of computational SSD offload.

6.2 Offload Function Classification

To form a methodology that systematically evaluate whether a function is good for computational storage offloading, we start from qualitative analysis. The analysis consider multifaceted metrics and function features. And to reiterate, the criterion is really whether computational storage offloading brings the most performance speedups and system efficiency improvement, not just whether computational storage offloading brings compute speedups.

Qualitatively, the benefits we are seeking in computational storage are offloading (free up precious compute CPUs), interconnect traffic reduction (mitigate interconnect bottleneck and reduce energy), and compute speed improvement (from architecturally exploiting acceleration). The offloading benefits are automatic after move the functions to storage. But interconnect traffic reduction and compute speed improvement are not a given. Computation speedup benefit is less distinctive. Generally, if one function can enjoy computation speedup from in-SSD acceleration, it can also enjoy same maybe higher (because of looser power constraints) speedups with compute-side acceleration. It is the interconnect traffic reduction benefits that uniquely warrant a function to be offloaded to the storage, as compute-side acceleration cannot generate this benefit. And then, computation speedup benefit would be a nice plus. It would not only improve throughput of the in-SSD compute, but also reduce the possibilities of in-SSD computation requests reducing the availability of SSD, which is critical as a shared resource (e.g. slow compute requests blocking the internal request queues of SSD).

Based on the understanding from the qualitative analysis, a classification is proposed to identify and prioritize computational storage offload opportunities. The classification considers three workload properties: Data Size Change Offloading Direction and Vectorizable. **Data Size Change** captures whether the size of the output of the function is smaller, the same, larger than its input. Or, in other words, whether the function reduces/keeps/increases the size of the data after applied. **Offloading Direction** captures the direction data flows through. A function can either be on read path, where input data come from flash array and gets processed by the function with results are transferring out the SSD, or on the write path, where input data comes from outside of the SSD and gets processed by the function with results writing into the flash array. Offloading Direction is coupled with Data Size Change to reflect traffic reduction benefits. A size-increase function on the write-path or a size-reduce function on the read-path would lead to SSD interface and interconnect (PCIe links or networks) traffic reduction, if the function is offloaded to computational storage. **Vectorizable** reflects whether the function expose data-level parallelism that could be exploited by SIMD architectures for high-performance computing. Vectorization is a natural match to computational storage as storage data are usually set of objects, amenable for SIMD acceleration. Further, SIMD ISA improves instruction expressiveness and in turn improves compute efficiency by reducing instruction traffics, making itself a good fit for computational SSD with stringent power constraints. And the community have good understanding on architectures exploiting vectorized operations, which priorities the corresponding architecture support.

The results of applying the classification on the functions from Table 6.1 is shown in Figure 6.1. The x and y axis are for Data Size Change and Offloading Direction respectively. And we futher dashed-understore or full-underscore the functions that are moderately difficult to vectorize or highly difficult to vector respectively.

For filesystem functions, Compress and Deduplicate are considered offloading on the write path to reduce data written into flashes, saving storage and provide effectively higher storage bandwidth. Reversely, Decompress restores the original data on the read path, increasing data size. Decrypt and Encrypt are dual functions that take place on read and write path respectively to securely store the data in storage, without changing data size. Erasure Coding and Replicate enables failure recovery and increase availability by introducing redundancy, increasing data size on the write path. All these filesystem functions feature data-level parallelism at least on the block level. However, it is hard to vectorize compress, blocked by the dynamic huffman tree creation process which is highly branchy. Branches creates divergence, leading to low utilization of SIMD datapath and high overhead for branch management, eliminating the acceleration benefits from vectorization. For decompress, there are less branch kernels involved. And thus there are sporadic work [13, 76, 112] vectorizing it. Other functions feature straightforward vectorization opportunities, including specialized SIMD support for Cryptography [42].

For database functions, read-paths functions like Filter, Select, Parse and Statistics all reduce data size through reducing either the data amount or size per data element.

Write-ahead-log (WAL) Replay reapplies transactions records the updated entries to to the database stored in storage. It is size-increasing for having to writeback updated entries by the blocks. Filter, Select, Statistics all feature tuple-level parallelism for row-oriented databases or element-level parallelism for columnar databases, amenable to vectorization. Parse was slightly hard to vectorize, due to its automaton-like structure. But we have seen widely-adopted vectorized json parsing research [74] and software library [84]. WAL Replay is hard to vectorize due to the parsing job embedded, immensely scattered data, and the stringent transactional requirements.

Statistical modeling and Neural networks usually happens on read-path where the data to be mined or to train the neural network stream from flashes. And they provide hugely smaller amount of results to the outside compared to the mining/training data, thus greatly reducing the data size. They are also highly vectorizable due to regular computational structures.

As discussed in the qualitative analysis, functions that provides interconnect traffic reduction benefits and are easily vectorizable to match the in-SSD compute performance to the flash array to avoid SSD availability reduction are the first candidates to consider for offloading to computational SSDs. In other words, the offloading functions should ideally be a size-reducing one on the read path, or size-increasing one on the write path, or one that operates alone in storage and does not communicate outside the storage. And they should be vectorizable to to exploit data-level parallelism through SIMD, the most accessible architecture approach to bring high performance with high efficiency. These first candidates are shaded in green in Figure 6.1.

6.3 Workload Properties and Architecture Opportunities

Dwelling on the functions identified as most suitable to be offloaded to computational SSD, i.e. the ones shaded in green in Figure 6.1, key properties can be identified that further

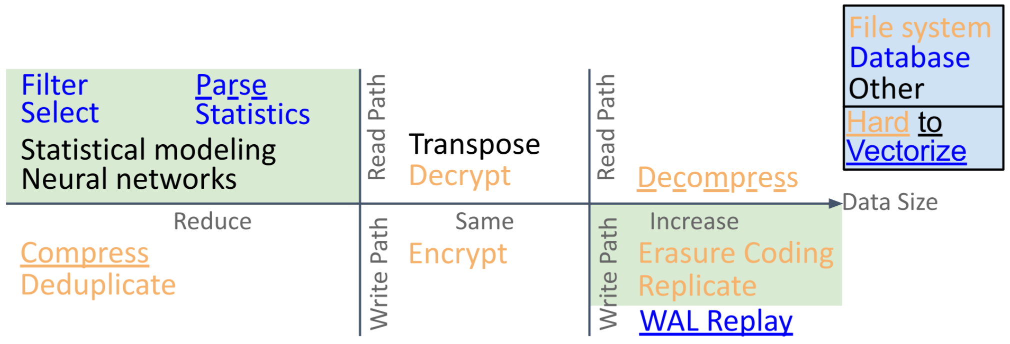


Figure 6.1: Function classification

encapsulate architecture opportunities.

The first observation is that these functions are diverse, calling for general-purpose architecture supports instead of ASIC approaches. Functions may process text input (Parse), binary input (Filter, Select, Statistics) or even treat input in a content-agnostic way (Erasure coding, Replicate). The inputs may be organized in be tuples, binary columns, matrices, or just fix-sized data blocks. The computation could take place in This observation align with our vision of that general-purpose high-performance architectures are needed for computational SSDs. Further, in chapter 8, we explore processor architecture support for efficient computation on these diverse inputs, improving efficiency along with performance.

The second observation is that all these functions feature streaming accesses to input data, with almost no reuse of input data elements. For example, database functions stream in tuples or binary data. Neural network functions stream in both input images and layer weight matrics. And Erasure Coding and Replicate stream in data blocks to general additional data blocks for availability, reliability and failure-recovery. This observation is the root insight for our SSD architecture support exploration in Chapter 7

6.4 Summary

To summarize, in this section, we explore a methodology to systematically determine whether a function is good candidates to offload to computational SSD, and enables the projection of the offloading benefits at the same time. Through qualitative analysis on fundamental benefits of computational SSD offload and comparative advantages against compute-side offload, we propose to classify offloads based on three properties: data size change, offload direction and vectorizable.

Offloading data-size-reducing function on the read-path and data-size-increasing function on the write-path would lead to SSD interface and interconnect traffic reduction, producing the comparative advantages against other compute-side offload. Further coupled with vectorizability, functions offloaded to storage would have the potential to enjoy compute performance matching or excceeding the bandwidth of the flash array, providing further compute speedup benefits.

This classification not only answer the fundamental question on what functions to offload but also provide a group of high-benefits functions as the workload base to guide us making architectural decisions for SSD architecture and processor architecture in computational SSDs, as will be discussed in in Chapter 7 and Chapter 8 respectively.

CHAPTER 7

ASSASIN: AN SSD ARCHITECTURE WITH EFFICIENT STREAM COMPUTING SUPPORT

7.1 Motivation: Memory Wall in Cost-effective SSDs

In this section, we show that to support in-SSD compute with continuously increasing flash bandwidth capacity, the bandwidth requirement on the SSD DRAM would have to scale accordingly in the state-of-the-art general-purpose computational SSD architecture. However, the DRAM scaling requirement not only is contingent on the silicon technology development of whether DRAM improves better than flash throughput-wise towards future, but also goes against the goal of high efficiency and low power for cost-effective SSDs, which are the backbone of modern data centers. As a result, exploring new SSD architectures to provide efficient support for in-SSD compute is necessary.

7.1.1 A Motivating Example

Let us consider an exemplar function, Filter, offloaded to computational storage from data analytics workload to understand the performance characteristics inside the state-of-the-art general-purpose computational SSD architecture shown in Figure 7.1, which is employed in multiple computational SSD studies [40, 65, 75, 78, 82, 99, 102, 108].

The function filters tuples based on given predicates on certain fields. Tuples come from the database (to be specific, TPC-H lineitem table) stored in the SSD flash array of which the schema is known and no parsing is needed. This Filter function features early data reduction benefits when carried out with computational SSD because unselected data would not come out of the SSD interface. Further, the function features tuple-level parallelism, such that it is easy to exploit scaled-out compute engines inside SSD for high performance, as each engine could process tuples in a small batch of flash pages. Thus, this Filter function

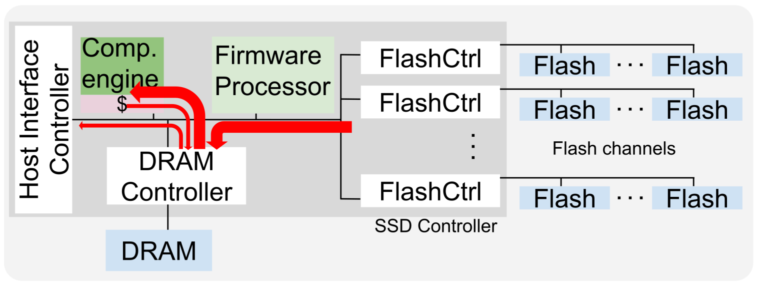


Figure 7.1: State-of-the-art computational storage architecture

is very suitable for offload to computational SSDs, and is considered in most computational storage studies (see Table 6.1).

In the state-of-the-art general-purpose computational SSD architecture, which is depicited in Figure 7.1, offloaded functions execute on compute engines (embedded-class cores). Data is first staged in the SSD DRAM, and then accessed by compute engines through the cache-DRAM memory hierarchy for processing. With a 1GHz in-order RISC-V scalar core on top of a 32KB 8-way L1 data cache and a 256KB 16-way L2 cache, the offloaded Filter function runs at 0.63 GB/s. The simulation is done in Gem5 [46] with cache performance measured through Cacti [45] @ 14nm.

Although the Filter function is light on computing intensity, the achieved performance is far from the bandwidth of a flash channel (1.6GB/s or 3.2 GB/s depending on channel width as defined in ONFI 4.2 [55]). Digging deeper, we find that the performance is hindered by memory access stalls, as shown in the cycle decomposition detailed in Figure 7.2. Even if we assume the L1 cache is tiny (no extra delay for memory accesses) but perfect (no cache misses except compulsory). Compulsory misses and DRAM accesses would still slow down the performance by three times.

At the SSD level, let us consider the setting of eight 8-bit flash channels delivering 12.8GB/s to maximally utilize the current and future NVMe over PCIe storage interface (4 lanes of PCIe 4.0 combined peaks at 8GB/s). The memory bandwidth requirement of the SSD DRAM is at least 25.6GB/s (firmware processor load pages from flash controllers into

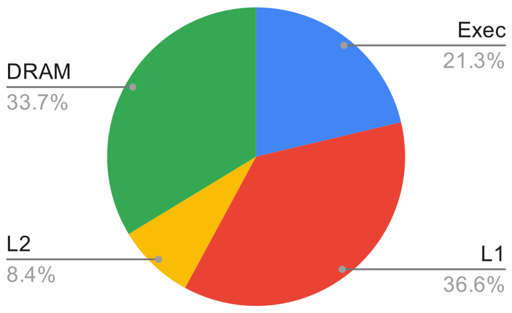


Figure 7.2: Cycle decomposition for ‘Filter’

DRAM, and compute engines read pages from DRAM to perform in-SSD computing). This is shown with thick red arrows in Figure 7.1. The full requirement will be higher, as there is additional traffic for storing and transferring the in-SSD computing results or firmware DRAM access needs. These requirements exceed the capabilities of LPDDR4 DRAM used in current SSD products [24], and even exceed a DDR4 DRAM of 16GB/s sustained bandwidth.

7.1.2 The Problem

Our example illustrates the *memory wall* problem inside a computational SSD. State-of-the-art computational SSDs read data from the flash array into the SSD DRAM. And compute elements then process these data on DRAM through the cache. This makes SSD DRAM bandwidth a critical performance limit for hosting both compute and flash traffics. As flash bandwidth scales, so must the computation, producing increasing demands on DRAM bandwidth and creating a hot spot in the SSD architecture.

CPU solutions to the *memory wall*, including caching and increase on memory parallelism (multi-channel or HBM), do not readily apply. Caches do not work well for streaming data due to low reuse. Increased memory parallelism is both expensive and high-power. These costly CPU approaches are not viable in the extremely cost and power-sensitive SSD storage space (backbone of modern data centers).

The computing and memory hierarchy architecture inside computational SSDs should be lightweight but effective. It should enable compute engines inside to access data with low latency, low power, but high bandwidth. It should also feature moderate consumption of silicon resources. We dive into the workloads for insights that may help us resolve the *memory wall* scaling problem in computational SSDs.

7.2 ASSASIN Design

7.2.1 Workload Feasibility for Stream Computing

To find a solution to the memory latency issue and the memory wall problem we uncovered, we go back to the workload for insights. Dwelling on functions that are first candidates for computational storage offload as discussed in 6.3, we find that implementations of these functions feature streaming access to storage data and random access to relatively-limited function states (relative to the storage data that are streaming through).

Starting with file system functions, for Cryptography, the key schedule is usually determined at the beginning and thus suitable for storing as function states. Encryption or decryption would be applied to the input data or code blocks in a streaming fashion. For compression and decompression, besides the input data streaming in, the recent history of the data streaming in (compression) or out (decompression) needed as the dynamic dictionary could be seen as function states. Different implementations all have an explicit upper bound on the history size, which also limits the size of an additional suffix tree or a hash table for indexing this dynamic dictionary. Deduplication is similar to compression except the dictionary is metadata on seen blocks. For erasure coding, it reads in multiple streams of data blocks and generates extra coded blocks through various Galois field operations before streaming out. There are no states but a Galois field multiplication lookup table used across erasure coding operations for different blocks.

For database functions like Filter, Select, Parse, these workloads feature a highly parallel nature where computation is applied to each row of data independently. As a result, aside

	Streaming	**Function States**
Cryptography	Data blocks / Code blocks	Keys & GF table
(De)compress	Data and history	Dictionary indexes
Erasure coding	Data blocks / Code blocks	Galois Field (GF) table
Filter	Tuples	–
Select	Tuples	–
Parse	Tuples	–
Statistics	Tuples	Accumulators
NN Training	Training data	Model parameters
NN Inference	Inference sample	Model parameters

Table 7.1: Stream computing implementation of computational storage workload

from temporaries and streaming input/output of table data and results, there are no function states for state transfers. It is similar for the Statistics function which generates statistical summaries from data tuples, it only needs additional accumulators as the function states.

For neural network training and inference, it is sensible for either a general-purpose processor or an accelerator to keep weights of the model stationary as function states and streaming in the inference or training data. We summarized how functions are mapped to stream computing in Table 7.1.

7.2.2 ASSASIN SSD: Stream Computing Between Flash Controllers and DRAM

The ASSASIN SSD architecture is shown in Figure 7.3. Contrast to a regular SSD architecture as shown in Figure 3.1, ASSASIN adds scalable stream computing cores (ASSASIN cores) on the SSD controller chip logically between the SSD DRAM and the flash array. ASSASIN cores carry out offloaded functions as inline stream computing on the data stream (denoted by the blue arrows in Figure 7.3) between SSD DRAM and the flash array. Note that flash controllers hide the electrical complexity of flash, such that ASSASIN cores can be implemented with standard CMOS VLSI methodologies.

Comparing to the state-of-the-art general-purpose computational SSD architecture as

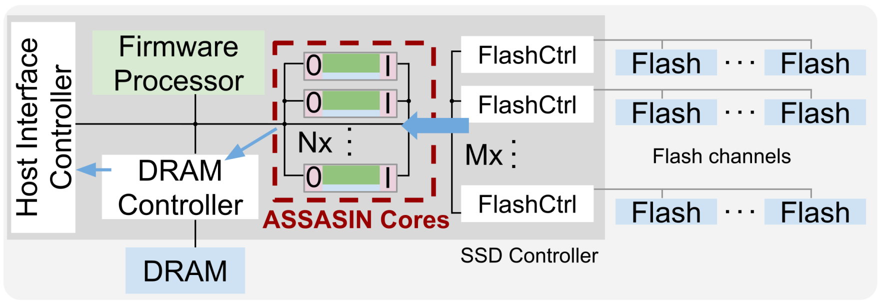


Figure 7.3: ASSASIN SSD (Compare with Figures 7.1 and 7.4)

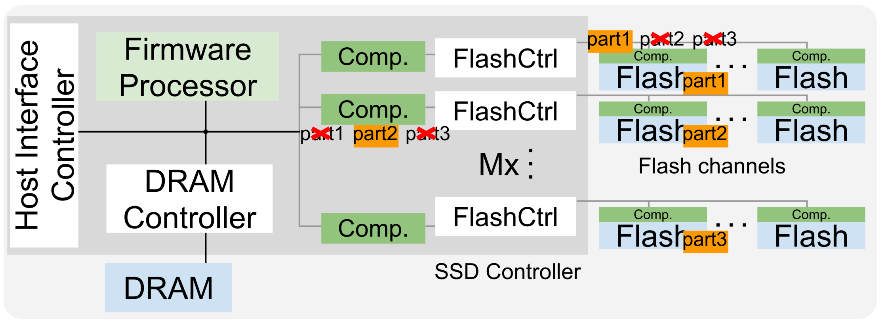


Figure 7.4: Application-specific computational storage proposals [66, 90, 91] bind acceleration to channels, and thus cannot flexibly share compute and compose data from across the flash array

shown in Figure 7.1, where compute engines fetch storage data through a traditional cache-DRAM memory system only after data are first staged in SSD DRAM, ASSASIN saves at least half of the SSD DRAM traffic. Further, function offloads to computational storage often reduce data (read) or increment data (write), generating system benefit by decreasing traffic at the storage interface. ASSASIN allows these functions to be implemented between SSD DRAM and the flash array via ASSASIN cores, harvesting these traffic reduction benefits for the SSD DRAM as well. As a result, ASSASIN SSD architecture largely reduces SSD DRAM traffic and bandwidth requirement, addressing the memory bottleneck issue raised in Section 7.1.

ASSASIN also differs significantly from proposed application-specific computational SSD architectures (Figure 7.4) that add specialized compute engines to each channel/controller or to each flash die [66, 90, 91]. ASSASIN has two key advantages: First, ASSASIN flexibly shares compute engines across the SSD with a crossbar interconnect, delivering

Table 7.2: Instruction set extension for stream access and management

Instruction	**Description**
StreamLoad	Hangs if empty, i.e. IHead[rs1] == ITail[rs1]. Otherwise: R[rd] = IStream[rs1][IHead[rs1]], IHead[rs1] += width
StreamStore	Hangs if full, i.e. OHead[rs1] + 8 > OTail[rs1]. Otherwise: OStream[rs1][OHead[rs1]] = R[rs2], OHead[rs1] += width
ReadIStream	R[rd] = IStream[rs1][ITail[rs1] - R[rs2]]
ReadOStream	R[rd] = OStream[rs1][OHead[rs1] - R[rs2]]
Instruction	**Encoding**
StreamLoad	unused[11:0] rs1[4:0] width[2:0] rd[4:0] opcode[6:0]
StreamStore	unused[11:5] rs2[4:0] rs1[4:0] width[2:0] unused[4:0] opcode[6:0]
ReadIStream	unused[6:0] rs2[4:0] rs1[4:0] width[2:0] rd[4:0] opcode[6:0]
ReadOStream	unused[6:0] rs2[4:0] rs1[4:0] width[2:0] rd[4:0] opcode[6:0]

robust performance even with uneven data distribution across channels. Second, ASSASIN's crossbar interconnect enables aggregating pages from different channels and presenting them to the compute engines. This allows FTL placement and management decisions to be completely independent (and thus, no customized FTL is required for ASSASIN). As a result, ASSASIN can support flexible interleaving of read/write requests that do not exploit computational storage with computational storage operations.

7.2.3 ASSASIN Core: Efficient Streaming

An ASSASIN core is based on a general-purpose core (like the firmware processor) but extends it in following aspects.

Hybrid Hierarchy for Inline Streaming Each ASSASIN core employs a hybrid memory hierarchy consisting of input/output streambuffers, a scratchpad and a cache, as shown in Figure 7.5. Input and output streambuffers are for low-latency storage stream access. Scratchpad is tightly integrated with core pipeline and thus offers low-latency random access to function state. The cache which is further backed by SSD DRAM is for holding data structures larger than scratchpad's limit (i.e. a fallback capacity memory).

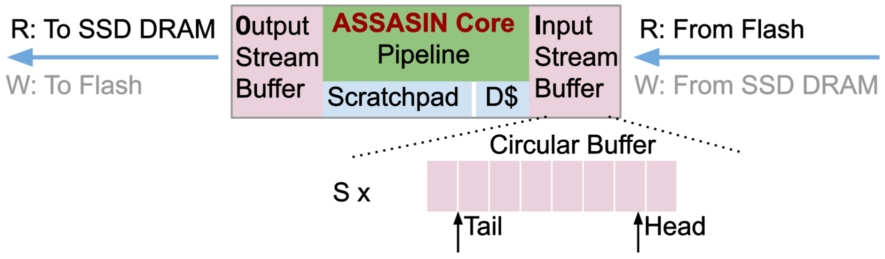


Figure 7.5: ASSASIN core

Stream Buffer Under a Microscope As drawn in Figure 7.5, a stream buffer can hold up to S streams. For each stream, there is a circular buffer with the capacity of P flash pages. Here, P and S are both microarchitecture parameters. There are two pointers on each circular buffer, Head and Tail, which are both control status registers (CSR) of an ASSASIN core. Head points to a core's current position on the stream. The word at the Head position could be easily prefetched into the core pipeline, allowing low-latency access. The Tail CSR acts as a doorbell register to be used by the SSD firmware. The firmware would update (advance) this Tail register to let an ASSASIN core know that new data are fetched into the circular buffer, ready to be processed.

Instruction Set Extension for Streaming Besides the CSRs described in previous paragraphs, the ASSASIN core augments a general-purpose ISA (we use RISC-V [110] in evaluation) with additional instructions to access (and automatically manage) input and output streams, which we summarized in Table 7.2. The first two instructions StreamLoad and StreamStore feature automatic stream pointer increments (i.e. Header CSR) based on the 'width' immediate, while the latter two have no effects on streambuffer pointers.

7.2.4 Flexible Interconnect: Scalable Compute

We architect ASSASIN to be scalable with the flash array bandwidth through pooling cores at the SSD-level connected with a crossbar interconnect. This allows a flexible N:M pairing (as the 'N' and 'M' in Figure 7.3), between the ASSASIN cores and flash channels to scale

Listing 7.1: ‘compute’ specified as a function

```
void compute(char* scratchpad) {
    while (true) {
            // An iteration processes one object, e.g. a tuple.
        char input = StreamLoad(0, 1); // InStreamId == 0, width == 1.
        char output;
        // Compute on input or fetch more data
        //     from input stream to produce output.
        // Maybe also use scratchpad in the process.
        // ...
        StreamStore(0, 1, output); // OutStreamId = 0, width = 1.
    }
    // The loop exists when StreamLoad hangs,
    //     i.e. input stream is exhausted.
    // Firmware would reset ASSASIN core (PC & pipeline)
    //     before next compute request starts.
}
```

out compute performance, as opposed to the fixed 1:1 pairing for channel-level compute engines [66, 90].

One may wonder whether ASSASIN is just shifting the traffic and bandwidth requirements from the SSD DRAM to the interconnect. The insight here is that the interconnect and stream buffers are streaming-oriented (small and restrictive thus allowing optimizations). They can be scaled to high bandwidth at much lower power when compared with the NV-DDR [55] facility that transfer pages from channels to the SSD DRAM (huge and random accesses). This advantage is the root of the system efficiency of ASSASIN. As a confirmation, a 16-bit per input/output lane, 16x16 crossbar is implemented as the interconnect in SystermVerilog. It easily closes timing at 2GHz with 14nm SAED library and is only one eighth of a core in silicon area (see Section 7.3.7).

7.2.5 ASSASIN Programming Model

Software Stack Extensions Generally, computational SSD requests are specified in the form of ‘(compute, pData, List[List[LPA]])’. ‘compute’ represents a stream computing function, the specification of which is detailed in the third part of this subsection. ‘pData’ is a host

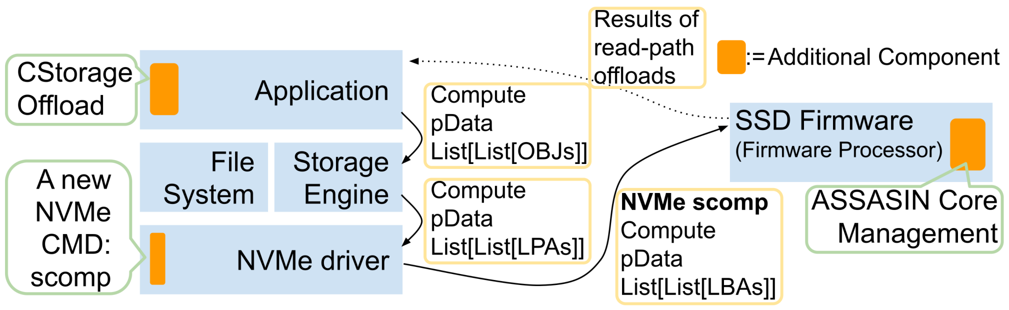

CStorage Offload
Application
File System
Storage Engine
A new NVMe CMD: scomp
NVMe driver
Results of read-path offloads
:=Additional Component
Compute pData List[List[OBJs]]
Compute pData List[List[LPAs]]
SSD Firmware (Firmware Processor)
NVMe scomp Compute pData List[List[LBAs]]
ASSASIN Core Management

Figure 7.6: Software stack extensions for ASSASIN

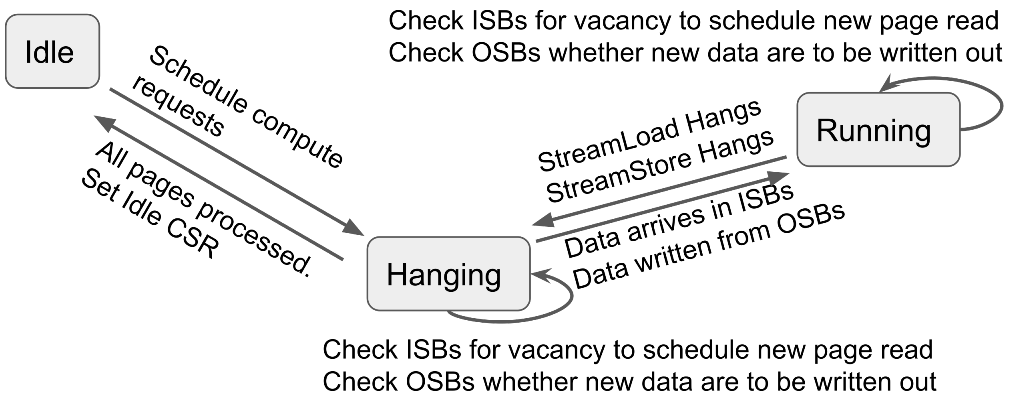

Idle
Schedule compute requests
All pages processed. Set Idle CSR
Check ISBs for vacancy to schedule new page read
Check OSBs whether new data are to be written out
Running
StreamLoad Hangs
StreamStore Hangs
Data arrives in ISBs
Data written from OSBs
Hanging
Check ISBs for vacancy to schedule new page read
Check OSBs whether new data are to be written out

Figure 7.7: ASSASIN core management FSM in firmware

pointer to either the input data for write-path computational SSD requests the results of which would be writing to storage, or the destination to store the results for read-path computational SSD requests. The 2-dimensional array specifies the logical page addresses (LPA) that the output stream(s) of a write-path offloaded function should be written to or that form the input stream(s) for a read-path function. And the size of the outer dimension corresponds to the number of input/output streams. A computational SSD request would be wrapped as a new NVMe command ‘scomp’, as shown in Figure 7.6.

If upper-layer applications would rather specify compute inputs in List[List[objects]], a storage engine would be responsible of transforming that in to List[List[LPA]]. This is the original responsibility a storage engine (a file system or a DBMS storage engine) would assume even without computational storage, so no changes are required here. Further, this is where task decomposition would take place to exploit the task-level parallelism through the multiple ASSASIN cores at the SSD-level. Large compute requests can be decomposed into multiple smaller ones with consistent splitting of each object/LPA stream.

ASSASIN Core Management in the Firmware Following the insights of control plane and data plane separation [99], the firmware processor (as shown in Figure 7.3) runs firmware (i.e. the control plane) independent of compute-oriented ISA/uArch innovations employed by ASSASIN cores. Compared to conventional SSDs, the firmware is extended with the capability of managing ASSASIN core resources. This includes schedule stream computing on any ASSASIN core and schedule page read and write to/from any input/output streambuffer (ISB/OSB).

As shown in Figure 7.7, the firmware periodically checks control status registers (Head/Tail for each ISB and OSB) of each ASSASIN core, performing state transitions (hanging avoids overflow) and schedules pages in and out streambuffers. This is where the construction of streams from pages at specified LPAs in computational storage requests (see the first part of this subsection) takes place. ASSASIN cores only process streams, without the need of knowing any flash array data layout or LPAs (preserving generality). The management process is similar to that of in-DRAM buffers which firmware originally assumes, thus no new challenges are involved.

Specify Streaming 'compute' 'compute' needs to be written in a streaming fashion, as shown in Listing 7.1. It reads from the input data stream(s) using StreamLoad instructions, performs required computation and appends output to the output data stream(s) via StreamStore instructions.

For existing applications, offload functions will in general need to be rewritten in a streaming fashion with ASSASIN ISA extension instructions (wrapped in intrinsics for high-level programming languages). However, compiler support to automate this task is feasible via automatic streaming access pattern identifications [106].

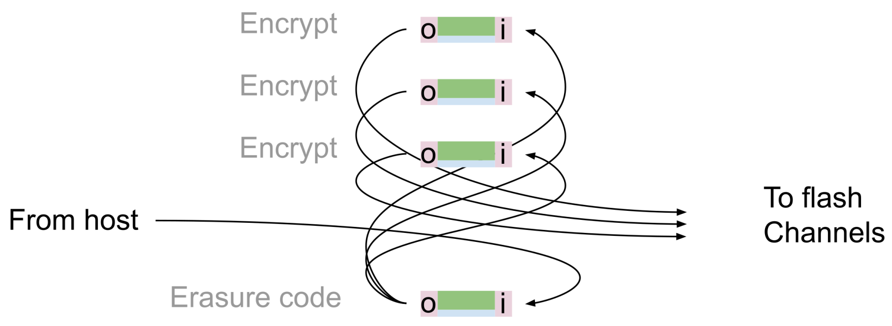


Figure 7.8: Erasure coding and encryption pipeline

Composing Complex Pipelines ‘compute’ could actually be generalized into a tree of compute functions, where the function at a tree node would read input stream(s) from its parent and provide output stream(s) to its children. This generalization only expands the notion of output stream destination to additionally include input stream buffers of all ASSASIN cores. Recall that ASSASIN cores’ stream buffers are all mapped in the SSD address space, no actual changes are needed in hardware. The generalized programming model allows us to compose complex function pipelines that will map to several ASSASIN cores with the help of ASSASIN’s flexible architecture and interconnect.

Let’s discuss an example of a write-path erasure encoding and encryption pipeline. It first applies a Hamming(3, 2) code in the granularity of an SSD page that adds a code page for every two data pages for additional reliability over a single flash chip failure. Then each page including the additional code page is symmetrically encrypted with a block cipher before writing to flash chips. This example is detailed in Figure 7.8. The generalized programming model allows load-balancing decisions to be specified. One light-weight Hamming(3,2) running on a core could generate input for two cores running the computing-intensive encryption function.

A complex neural network inference pipeline could also be programmed. Following the insights from inference accelerator work [67, 89], it is better to keep the neural network weights stationary inside the scratchpad to enjoy low-latency high-bandwidth memory access and reduce DRAM traffics. However, with limited scratchpad sizes, all weights of a model

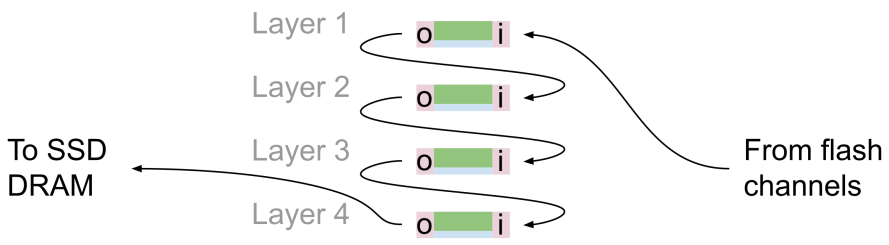


Figure 7.9: Neural network inference pipeline

Table 7.3: Configurations of in-SSD compute engines (8 1GHz cores in each configuration)

	Data Source	**ISA**	**MemArch per Core. 32KB L1I omitted**
Baseline	DRAM (8GB/s)	RISCV32IM	L1D: 32KB, 8 way, 64B cache line L2: 256KB, 16 way, 64B cache line
UDP [61]	DRAM (8GB/s)	UDP ISA	256KB scratchpad
Prefetch	DRAM (8GB/s)	RISCV32IM	L1D: 32KB, 8 way, 64B cache line L2: 256KB, 16 way, 64B cache line DCPTPrefetcher [63] (best in Gem5)
AssasinSp	**S**cratch**p**ad	RISCV32IM	64KB scratchpad 64KB I + 64KB O ping-pong scratchpads
AssasinSb	**S**tream**b**uffer	RISCV32IM +Stream ISA	64KB scratchpad 64KB I + 64KB O streambuffer (S=8 P=2)
AssasinSb$	**S**tream**b**uffer + Cache	RISCV32IM +Stream ISA	64KB scratchpad, 32KB 8W L1D 64KB I + 64KB O streambuffer (S=8 P=2)

may not be able to fit in the scratchpad. Our flexible architecture makes it possible to form a pipeline as shown in Figure 7.9 where each ASSASIN core handles one layer of inference and only stores one layer of weights in the scratchpad. This is analogous to the cross-node pipeline considered for distributed neural network training [71].

7.3 Evaluation

We simulate several computational storage kernels and full-system TPC-H data-analytics pipelines and compare ASSASIN against state-of-the-art SSD architectures.

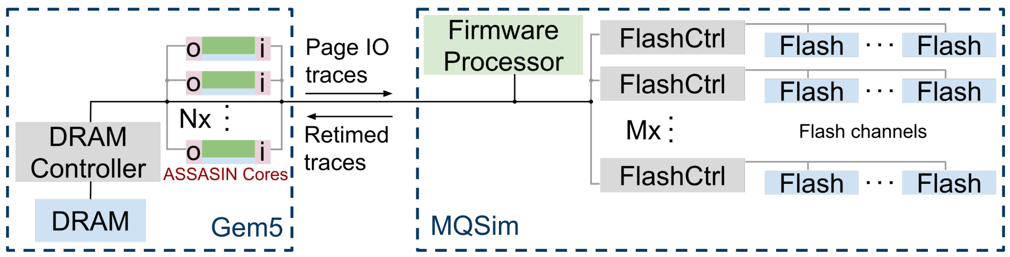
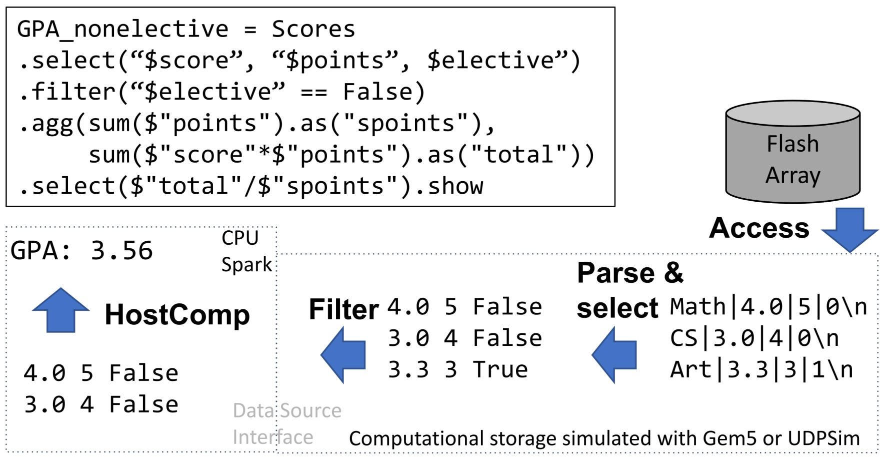


Figure 7.10: Hybrid simulation infra with Gem5 and MQSim



```
GPA_nonelective = Scores
.select("$score", "$points", $elective")
.filter("$elective" == False)
.agg(sum($"points").as("spoints"),
     sum($"score"*$"points").as("total"))
.select($"total"/$"spoints").show
```

```
GPA: 3.56
```

```
4.0 5 False
3.0 4 False
```

```
4.0 5 False
3.0 4 False
3.3 3 True
```

```
Math|4.0|5|0\n
CS|3.0|4|0\n
Art|3.3|3|1\n
```

Figure 7.11: Compstor-accelerated analytics pipelines

7.3.1 *Methodology*

Configurations We compare six computational SSDs that differ in compute engines and their integration in the SSD architecture, as summarized in Table 7.3. In other aspects the SSDs are similar, employing an 8-channel flash array (each channel has 1GB/s read/write performance), a 2GB LPDDR5 DRAM [24] (8GB/s effective bandwidth), and a PCIe Gen4 x4 host interface. In all cases, the host CPU driving the computational SSD is four-core eight-thread with 32GB main memory.

Compute engines in each SSD (except the UDP, see Table 7.3) are eight in-order scalar RISC-V cores with different memory hierarchies. RISC-V cores are selected because of the availability of open-source designs (we use ibex cores [101]) and toolchain [36]. **Baseline**, as drawn in Figure 7.1, represents the state-of-the-art general-purpose computational SSD architecture [40, 65, 75, 82, 99, 102, 108], where compute engines fetch data from SSD DRAM before computing on it. The other variants (Prefetch, AssasinSp, AssasinSb and AssasinSb$) add varied memory hierarchies at each core. **Prefetch** adds a prefetcher from SSD DRAM for latency reduction. There are three ASSASIN variations. In **AssasinSp**, conventional **S**cratch**p**ads are used to double-buffer storage data in a 'ping-pong' fashion. Storage data is fetched from flash into the 'pong' scratchpad, bypassing the SSD DRAM while the compute engines process data already in the 'ping' scratchpad and vice versa. **AssasinSb** employs a **S**tream**b**uffer which not only enables direct computation on storage data streams bypassing SSD DRAM, but also automatically manages the stream pointers through the core-level stream ISA extension (Section 7.2.3). **AssasinSb$** adds a data cache (backed by DRAM) for increased flexibility, with graceful degradation if the scratchpad size is not large enough for offload functions' needs of random access.

UDP [61], designed to accelerate data analytics, represents another dimension of computational SSD architectures [41, 73, 79, 90, 91, 113] where application- or domain- specific accelerator(s) are employed. A UDP lane computes using a private scratchpad. The firmware processor copies data to be processed from SSD DRAM into these scratchpads.

Simulation We adopt a hybrid simulation methodology as shown in Figure 7.10. Gem5 [46] is employed to model different memory hierarchy configurations and the compute performance of each. MQSim [107] is used to model flash performance. By combining one of the best simulators of the two worlds, the simulation results should be representative.

Gem5 is extended with scratchpad and streambuffer module [6] that features single cycle access to model AssasinSp, AssasinSb, and AssasinSb$ configurations. And the best-performant prefetcher, DCPTPrefetcher [63], in our benchmark is employed to evaluate Prefetch. At the same time of the simulation, access to scratchpads or streambuffers are traced to generate timed page-level IO traces which are further used as inputs to MQSim. Extended MQSim [7] simulates SSD internals and calculates the completion of each IO request, which we use to retime the computing process simulated by Gem5, i.e. adding additional latency if a page doesn't come out of the flash as early as compute engines modeled by Gem5 first access the specific page.

For UDP, UDPSim [61], the cycle-accurate simulator, is used instead of Gem5, but the SSD simulation and retiming by MQSim is the same.

Workload We first evaluate ASSASIN with four standalone function offload to a computational SSD: Statistics, RAID4 erasure coding, RAID6 erasure coding, AES encryption. Each of these functions is in its own an application. And we program these functions in C++ through the programming model we discussed in Section 7.2.5.

Then we consider TPC-H [37], which featured in much computational storage research [65, 73, 75, 82, 86, 99, 108, 113]. Our SparkSQL implementation [4] of TPC-H offloads Parse, Select and Filter operation through the datasource API [28] to computational storage simulated with Gem5 [46] or UDPSim [61]. Here we assume that the storage containing the TPC-H data would employ systematic coding (as in most erasure coded systems) so that source data are available even after coding, and that erasure coding blocks are large, reducing the boundary overhead (piecing together an object across SSDs/nodes).

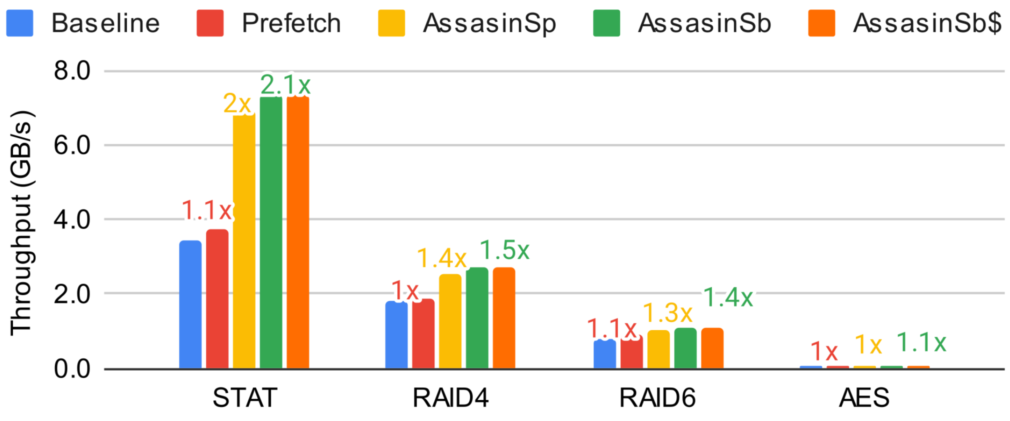


Figure 7.12: Throughput of offloaded standalone functions

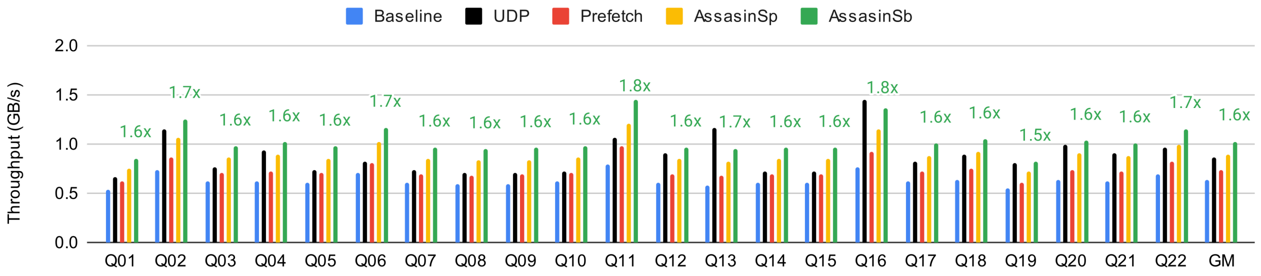


Figure 7.13: Computational SSD's throughput of offloaded PSF pipeline from TPC-H queries

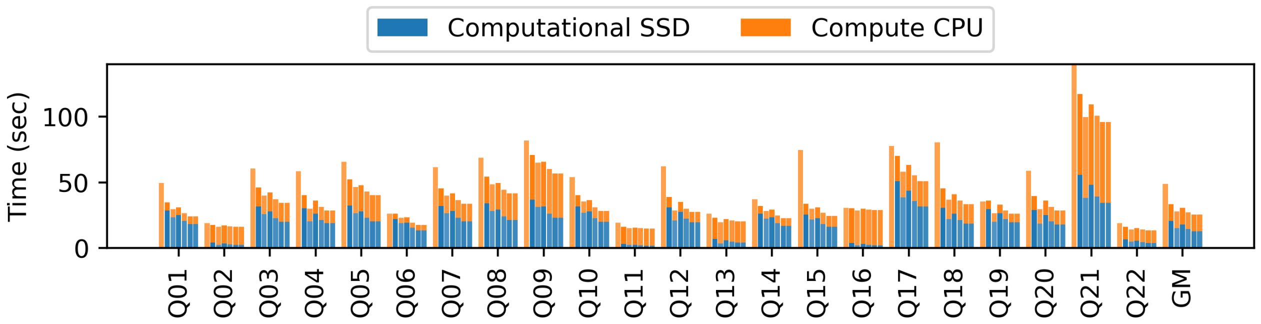


Figure 7.14: End-to-End latency. In each cluster: PureCPU, Baseline, UDP, Prefetch, AssasinSp, AssasinSb, AssasinSb$

7.3.2 Single-function Offload

We evaluate the performance of different computational SSD configurations running Stat (summing a column), RAID4 and RAID6 erasure coding and AES encryption over the same 8 GiB data array serialized in binary flatly (No deserialization or parsing is needed). Figure 7.12 shows the achieved throughput, with speedups over baseline labeled.

With the best prefetcher from Gem5, Prefetch is effective on reducing access latency. However, it only brings limited performance improvement because of the memory wall. The theoretically required DRAM bandwidth of Stat and RAID4 exceed what the LPDDR5 DRAM offers (~ 8 GB/s), leading to stalls for Prefetch. AssasinSp and AssasinSb address the memory wall issue through bypassing DRAM with Ping-Pong scratchpads or streambuffers, which leads to 1.3x-2.0x speedup for the first three functions and little to none memory bandwidth requirement. AssasinSb further outperforms AssasinSp by 10% on these cases with the stream ISA extension that enables automatic stream pointer management. AssasinSb and AssasinSb$ achieve the same performance as function states all fit in the scratchpad. Effectively the L1D cache provides no benefit (is not exercised by the program).

Please note that Stat, RAID4, RAID6 and AES are a sequence of functions with progressively greater computation intensity (ops per bytes). Generally, ASSASIN gives greater benefit for less compute-intensive functions, alleviating a memory bottleneck. As compute intensity increases, computing becomes the performance bottleneck. But it's worth pointing out that by employing acceleration, some originally compute-intensive functions (e.g. AES-NI for AES) could become memory-intensive and thus benefit from ASSASIN.

7.3.3 Database Function Pipeline Offload

Now we look at computational SSD offload of a database function pipeline consisting of Parse, Select and Filter (PSF) from TPC-H workload with scaling factor 10 ($\sim$10GiB data), the system architecture of which is shown in Figure 7.11. Performance of offloaded functions is detailed in Figure 7.13.

UDP ISA [61] employs multiway dispatch and instructions that fuse operations to accelerate branch executions and unstructured data computation. This is the reason that UDP achieves on average (GeoMean) 1.3x speedup on offloaded PSF database function pipeline (this workload is well supported by UDP ISA specialization) over the Baseline.

On the other hand, we consider the progression of changes on Baseline's memory hierarchy while keeping general-purpose cores. PSF, bottlenecked by the Parse function, is moderate in terms of compute intensity. Prefetch achieves small speedup averaging (GeoMean) at 15% by hiding DRAM latency for accessing storage data. Adopting the idea of inline computation of storage data streams, AssasinSp matches UDP's speedup without UDP's exotic ISA customization. This comes from the low latency access to function states as well as storage data streams. Finally, both AssasinSb and AssasinSb$ further improve performance by 18% through the use of streambuffers and the stream ISA extension that eliminates the address

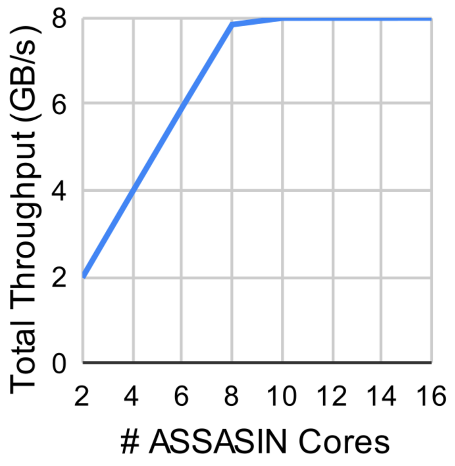


Figure 7.15: Compute performance scales linearly with cores.

calculations and pointer management instructions. This is 1.5x - 1.8x higher throughput than Baseline. The variation in speedup generally reflects the memory intensity of each offloaded pipeline.

Finally, we look into the overall data analytics performance for all 26 TPC-H queries, which stacks the host compute latency and computational SSD latency together. We also include the pure-CPU performance without computational SSD offload for comparison which essentially represents disaggregated storage architecture. Figure 7.14 details overall latency. Comparing to the Baseline, AssasinSb's 1.5x - 1.8x higher performance on in-storage compute translates to 1.1x - 1.5x end-to-end speedup which averages (GeoMean) at 1.3x. And please notice this is on top of the 1.9x speedup Baseline already brings over the no-computational-SSD pure-host-CPU (i.e. disaggregated storage) scenario.

7.3.4 Performance Scalability

The ASSASIN SSD features an all-to-all interconnect between ASSASIN cores and flash channels (through flash controllers) as discussed in Section 7.2.4 to enable flexible performance scaling. Here we evaluate scalability and whether the crossbar interconnect potentially creates hot spots at a flash channel or causes ASSASIN cores to stall and wait for requested

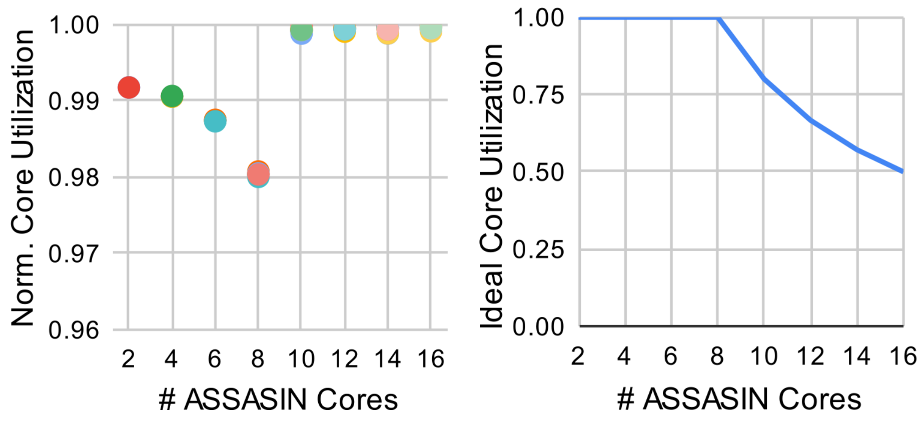


Figure 7.16: Normalized core utilization (left), normalized by ideal utilization (right).

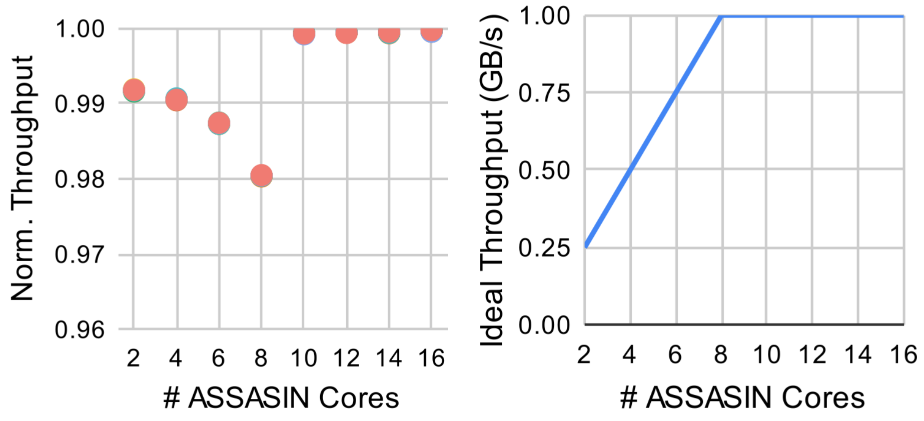


Figure 7.17: Ext4/MQSim layout combined with Xbar produces even load across flash channels

storage data. We consider a dummy workload where each ASSASIN core (AssasinSb variation) scans each byte of input, using the TPC-H datasets. If input data is always available, a 1 GHz core achieves 1 GB/s.

Various numbers of ASSASIN cores are considered to evaluate scalability. Figure 7.15 shows the achieved compute throughput of the computational SSD during scaling. Figure 7.16 shows the corresponding core utilization (normalized by the ideal utilization, derived by considering nominal bandwidth relationships between cores and channels). As shown, the interconnect allows linear scaling of compute performance until bounded by flash array throughput (8GB/s in total). Cores have high utilization, more than 98%, reflecting that the

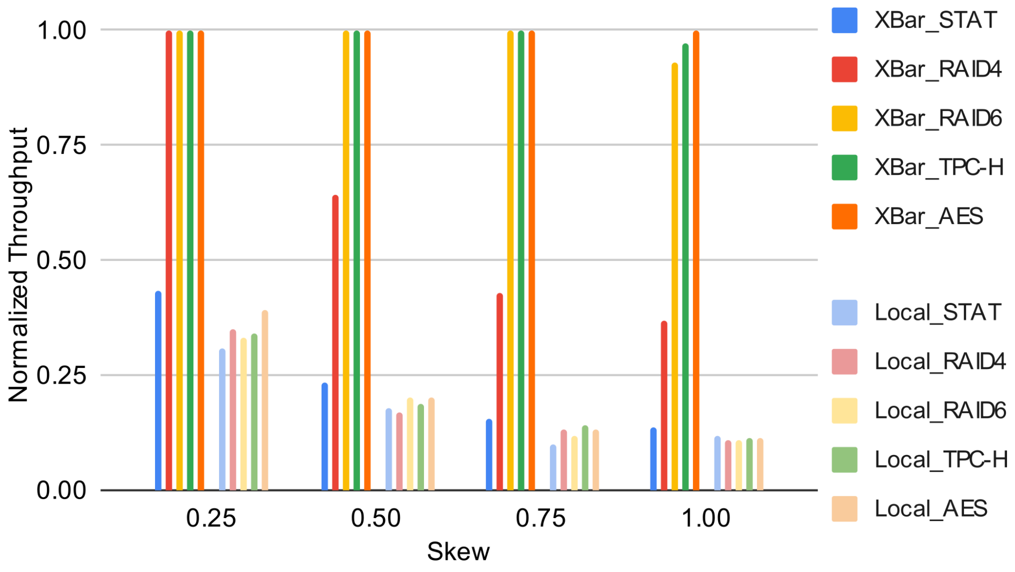


Figure 7.18: Performance sensitivity to skew across channels (flash data layout)

flash array and the interconnect deliver pages in time to keep the cores busy. Further, flash channels also feature balanced high throughput as shown in Figure 7.17. This is because independent FTL (FTL modeled by MQSim is employed here) already aims to distribute pages evenly across the channels for better storage performance, separate from computational SSD considerations.

7.3.5 Sensitivity to Flash Array Data Layout Skew

We further evaluate ASSASIN's (AssasinSb variation) performance sensitivity to flash array data layout skew, which is defined as (D_i denotes the amount of the to-be-processed data in the i-th channel):

$$Skew = \frac{1}{n-1} max_i(\frac{D_i}{avg_i(D_i)}) \qquad Skew \in [0,1]$$

ASSASIN with the SSD-level crossbar interconnect to redistribute flash data to compute engines is compared with the alternative architecture (Figure 7.4) from application-specific computational storage [66, 90, 91]. Ignoring FTL generality issues of the alternative architecture

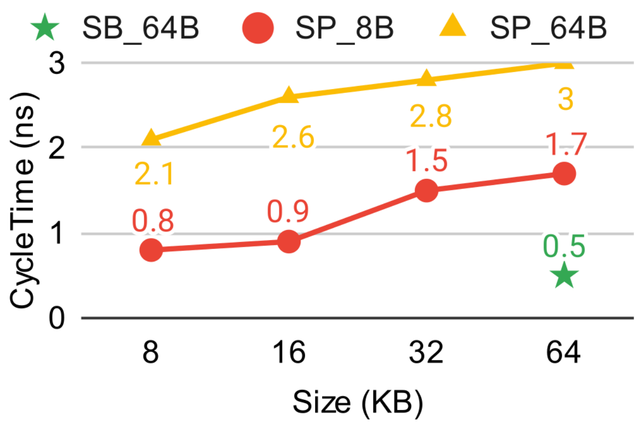


Figure 7.19: Timing for ASSASIN MemArch extensions (SB = Streambuffer, SP=Scratchpad)

discussed in Figure 7.4 and Section 7.2.2, ASSASIN cores are switched in at each channel to perform channel-local compute. Four layouts with varied skew for requested data are evaluated (from low skew (Skew=0.25) to extreme skew (Skew=1)) in additional to the no skew senario. Respective performance normalized to no skew one are shown in Figure 7.18.

The two clusters of bars show that the crossbar architecture consistently outperforms the channel-local compute architecture in the presence of skew. Further as skew increases the benefits increase to as much as 8-fold.

Within the XBar cases, for flash-read limited functions like Stat, uneven layout aggravates the storage read bottleneck, and XBar-based global compute helps less. But for compute-limited functions like RAID6, TPC-H and AES, XBar-based global compute can source pooled compute engines effectively for the data read out from the most-skewed channel, matching the overall throughput with that of the even data layout scenario.

We conclude that ASSASIN's XBar interconnect architecture achieves robust performance, thereby enabling flexible, independent FTL layout mananagement. Benefits compared to channel-local proposals (as in Figure 7.4) can be as large as 3-8x.

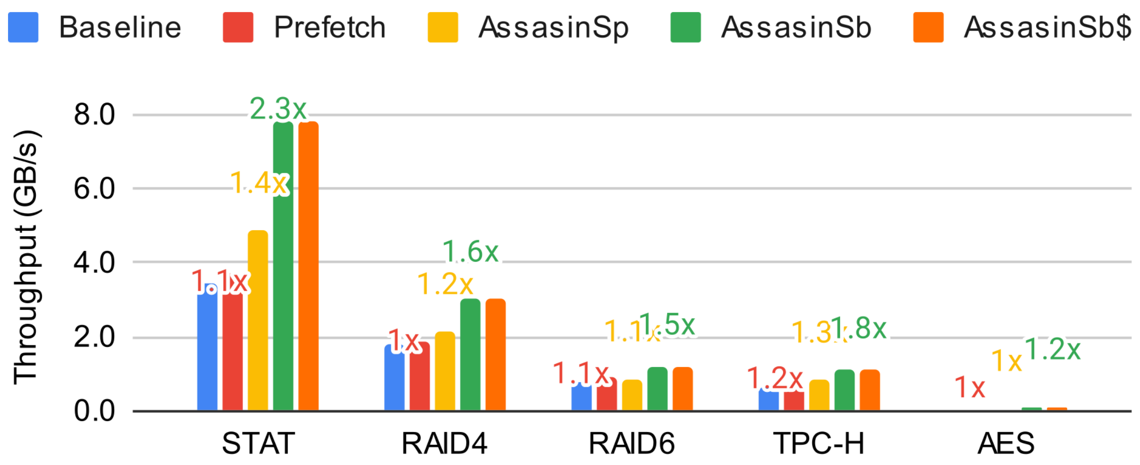


Figure 7.20: Throughput after timing adjustment (TPC-H is GeoMean across queries))

7.3.6 Clock-speed Benefits and Adjusted Performance

To assess the clock-speed benefits of ASSASIN's streaming architecture, we implement the ISA extension described in Section 7.2.3, using SystemVerilog and with SAED14nm [25]. The implemented streambuffer (for AssasinSb) provides 1B-64B access to the stream heads in the 'MEM' pipeline stage of each ASSASIN core. As a comparison, scratchpads (for AssasinSp) with varied widths (8B for a scalar core, 64B for SIMD extensions) and sizes are implemented.

Our results (see Figure 7.19) show AssasinSb's streambuffer achieves 0.5ns per cycle even with a wide, i.e. 64B, interface provisioned for SIMD. A small and prefetched FIFO, built upon restricted streaming semantics (head-only) of 'StreamLoad' and 'StreamStore' instructions, on top of streambuffer with coarse-grained (128B aligned) accesses is the key to this high speed.

In contrast, the scratchpad implementations are significantly slower because they require random access (large MUXes / access trees). At 64KB, the scratchpad even with a narrow 8B interface, requires 2 cycles for each access in a 1GHz core. Thus, AssasinSp where most accesses are served by the scratchpads would take performance penalty.

	Power (mw)	**Area (mm2)**
Ibex Core	0.241	0.039
UDP Core	1.210	0.059
32KB 8way $	0.379	0.044
256KB 16way $	1.136	0.282
64KB SRAM	0.145	0.030
Crossbar	0.439	0.005
Baseline	14.048	2.923
UDP	14.333	1.444
ASSASIN	12.625	1.625

Table 7.4: Power and silicon area

We consider these results in the context of a RISC-V core, with a classical five-stage pipeline (IF, DE/RR, EX, MEM, WB). In this design, the prefetch FIFO for the streambuffer is added where the dcache access would occur. Substituting dcache to much faster streambuffer allows the clock period to be reduced by 11% (the critical path shifts to 'IF'). On the other hand, scratchpads would have to be timed for 2-cycle accesses, and without any cycle time benefits.

Finally, we adjust the performance based on above findings, as shown in Figure 7.20. Overall, AssasinSb's throughput improves to 1.5x-2.4x (from 1.4-2.1x) over Baseline, resulting from the cycle time reduction. AssasinSp, with one additional cycle needed for scratchpads' accesses, degrades to only 1.1x-1.4 (from 1.3 to 2x). In short, on top of the DRAM bypassing feature, AssasinSb's streaming memory architecture and instructions delivers a further 1.5x increase in performance.

7.3.7 Power and Area Efficiency

We evaluate the cost of the in-SSD computing engines and their memory hierarchy in terms of power and silicon area of different configurations. Synopsys design compiler and 14nm SAED technology library [25] are used to evaluate the costs of logic. The in-order RISC-V cores we use in baseline and ASSASIN (i.e. AssasinSb variation) are based on ibex [101].

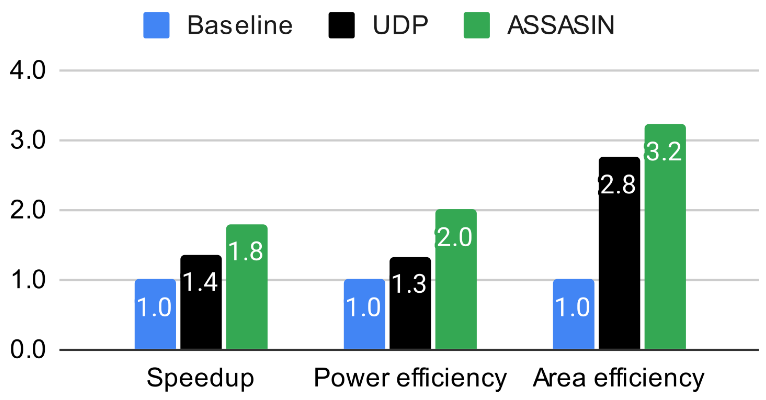


Figure 7.21: Relative speedup, relative power efficiency and relative area efficiency

For UDP, we evaluate with its SystemVerilog-based ASIC implementation. And Cacti [45] is used for caches, streambuffers and scratchpads.

The power and silicon area costs for subcomponents and three evaluated configurations are summarized in Table 7.4. One thing worth pointing out is that a L1 cache or similar-size SRAM are at the same order of magnitude with the compute logic of a core in area and power. This shows the significance of memory hierarchy innovation as pursued by ASSASIN. The speedup over baseline, relative power efficiency (speedup per unit power) and relative area efficiency (speedup per unit area) of each configuration are plotted in Figure 7.21. ASSASIN (i.e. AssasinSb) achieves 2.0x and 3.2x higher power and area efficiency comparing to Baseline through its streambuffer and scratchpad based memory hierarchy and streaming instruction-set extension. And ASSASIN(i.e. AssasinSb) with general-purpose RISCV cores also outperforms a UDP accelerator which employs an exotic customized ISA for unstructured data processing.

7.4 Summary

The memory wall problem in the computational storage emerges when in-SSD compute tries to match the continuously improving flash bandwidth. It would produce increase demands on the SSD DRAM bandwidth, creating a hot spot int he SSD architecture. With further limitations of memory scaling by the power-efficiency and cost-efficiency requirements on SSDs, the backbone of data centers, in-SSD memory wall surfaces.

To dissect the problem clearly, we studied a broad range of research proposals for domain-specific properties that allow architecture innovations. Key insight arises that computational storage functions all feature implementations with streaming accesses to storage data and random accesses to function states of limited size.

Inspired by this insight, we designed ASSASIN, which allows inline stream computing on flash data streams through a hybrid high-bandwidth memory hierarchy composed of stream buffers and scratchpad and a streaming instruction-set extension. ASSASIN allows flash data streams to completely bypass SSD DRAM, decoupling DRAM bandwidth requirements from the flash array bandwidth scaling, thus fundamentally addressing the in-SSD memory wall. As a result, ASSASIN achieves 1.5x - 2.4x speedup on functions offloaded in storage which translates to 1.1x - 1.5x overall end-to-end speedup, compared to the state-of-the-art general-purpose computational SSD architecture.

However, with memory wall addressed by ASSASIN, the compute performance of in-storage compute elements becomes the new bottleneck, especially for compute-intensive offloads as shown in Section 7.3.2. The hope is that the third part of the thesis project would address the bottleneck with an efficient high-performance computational storage processor design.

CHAPTER 8
VarVE: BRINGING SIMD PERFORMANCE TO VARIABLE-WIDTH VALUES

Within the SSD, compute engines are a critical bottleneck for computational storage systems. This is particularly true for compute-intensive tasks like parsing or cryptography today, and for future tasks such as data analytics employing machine-learning inference. In the third part of the thesis project, the goal is to architect a high-performance computational storage processor to meet these needs. Specifically, we address the deficiencies identified in conventional processors in computing on variable-width data elements, a staple of sensor data, and efficient information storage, both of which are critical limits towards matching and exceeding the continued increase in flash bandwidth.

As an approach, we focus on a single-instruction multiple-data (SIMD) extension of conventional CPU architectures. SIMD has been an extraordinarily successful instruction-set architecture approach in scaling the performance of both large-scale parallel and microprocessor architectures [2, 3, 15, 49, 95, 100]. In particular, SIMD's benefit combination of higher performance and lower energy per operation make it the most attractive option for computational storage processors.

Conventional SIMD approaches are unsuitable for computational storage, as storage data are diverse in width and often stored in unaligned layout. Thus, existing SIMD designs produce little benefit in this setting, with variable width data leading to not only divergence that eliminates the SIMD performance benefits, but also low datapath utilization as many of the bits are data pads for alignment. To overcome these problems, we consider a new type of vector ISA extension, VarVE, that builds on recent advances in the vector-length agnostics (VLA) SIMD, and extend it to capture computing over vectors of variable-width values. This extension can be done gracefully within the VLA context, and yields significant benefits for both performance and the integration of SIMD capabilities into the conventional

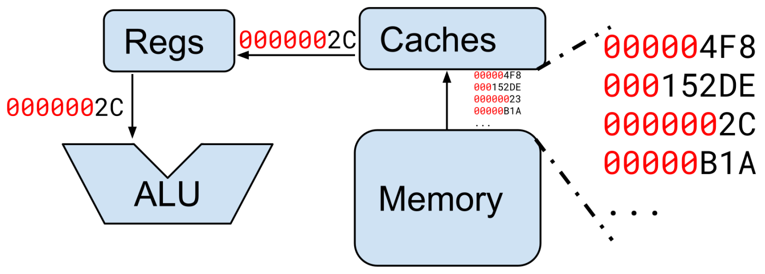


Figure 8.1: Wastes from value-datapath mismatch

ISA generally. We evaluate this extension using a variety of storage workloads from our workloads study (Section 6.3).

8.1 Problem and Approach

8.1.1 Observations on Datapath Efficiency

Real world data is diverse, and much of them does not need 64 bits to represent. The diversity usually comes from the biases, precisions and ranges embedded in the digitization process. Moreover, good programming practice considers the maximum value that could occur, even as a possible intermediate result, and type the variable accordingly. Execution of such programs on a conventional x64 processor leads to waste in memory traffic and bandwidth, cache capacity and bandwidth, and ALU, as in Figure 8.1.

For example, we consider a TPC-H filter [37]. To generate predicates on an enum column ‘c_mktsegment’ (the cardinality is five), all 64 bits of the datapath are involved in the compare instruction when only three bits are doing the essential work, producing a low datapath efficiency of 4.7%. Ideally, all of the datapath and register bits would be used for essential compute and storage. Here, 95% of them are wasted.

SIMD can partially mitigate the problem. MMX [94], was developed on this exact premise. By grouping a set of 8-bit values into a 64-bit register, one MMX instruction could apply the same operation to each value, both increasing performance, and wasting

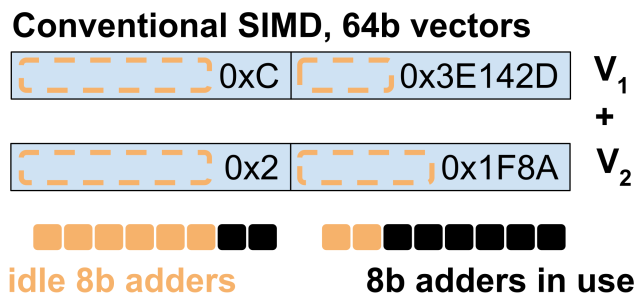


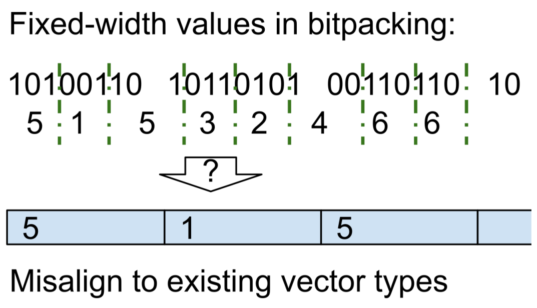
Figure 8.2: Padding wastes both the VecReg and datapath

far fewer ALU and register bits. Of course, MMX has evolved in to a long series of SIMD instruction set extensions for x86: SSE, AVX, AVX2, AVX-512 [3].

Most SIMD instruction sets support four vector types, with uniform bitwidths b ($b \in \{8, 16, 32, 64\}$) for each element. However, they still do not deliver high datapath efficiency. The primary reason is that programming must be conservative. When a programmer selects an element type, they must account for the largest outlier inputs, results, or even intermediate results, i.e. the worst case. This causes poor efficiency. In our motivating example, the TPC-H filter, the predicates need to be applied to the 'c_custkey' column, which is defined in schema to hold 32-bit integers. As a result, all predicates from the 'c_mktsegment' column have to be calculated in 32-bit-per-value vector types in ARM SVE. So even with multiple vector bitwidths, the program only achieves 9.4% datapath efficiency. Thus, existing SIMDs lead to huge padding wastes in both vector registers and datapath, as in Figure 8.2. Our observations exemplify the low datapath efficiency of existing architectures – in both SIMD and scalar processor architectures.

8.1.2 The Problem

The fundamental problem that gives rise to low datapath efficiency is the software-hardware interface, i.e. the instruction set, which have been defined around fixed-width datatypes for more than seven decades [43, 111]. If an instruction-set can only address and describe


Fixed-width values in bitpacking:
10100110 10110101 00110110 10
5 1 5 3 2 4 6 6
?
5 1 5
Misalign to existing vector types
(a) Bitpacked values misalign

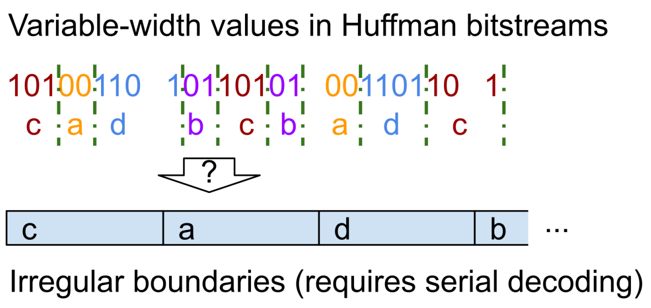

Variable-width values in Huffman bitstreams
10100110 10110101 00110110 1
c a d b c b a d c
?
c a d b ···
Irregular boundaries (requires serial decoding)
(b) Huffman's irregular boundaries

Figure 8.3: Misalignment between values and SIMD vectors

operations on fixed-width datatypes, to allow for dynamic 'fluctuation' and outliers, an application (and compiler) has little leeway to select the datatype size conservatively, producing wastes. The more variable the values in the computation, the greater the waste – in memory and cache, data movement, and computation.

8.1.3 Approaches and Challenges

To address the problem, we aim to design a variant of the ARM SVE instruction-set architecture. The new SIMD extension, VarVE, should allow programmers to address and express computation for sets of variable-width values, improving memory and datapath efficiency while maintaining the exposure of parallelism. Several challenges need to be addressed in order to deliver this expressive SIMD extension.

First, there is the question of how to capture variable-width values into vectors. Large datasets with such values are often coded or packed [18, 19, 21], creating a mismatch between the packed data values and SIMD datatypes. As in Figure 8.3a and Figure 8.3b, such packed values cannot easily align and load to a vector register. VarVE needs an efficient mechanism to provide flexible loading and conversion for diverse values.

Second, traditional architectures (scalar or SIMD) adopted fixed-width datatypes to simplify hardware. Computation on variable width values would require more complex management. In modern technology environment, VarVE must show that it can be implemented with acceptable complexity and achieve performance (e.g. clock rates) compatible with existing pipelines. This is key to delivering performance increase.

Third, over the past decade, commercial processor architects have turned increasingly to SIMD as perhaps the most important technique to increase performance. The complexity of expressing vector and matrix parallelism has produced an explosion in the number of SIMD instructions (e.g. ARM SVE alone has 800+ instructions [31] with hundreds slated to be added). The requirement of instruction variants for element widths of $\{8, 16, 32, 64\}$ compounds this problem. In adding support for variable width values, VarVE must solve the datapath and memory inefficiency challenges without exacerbating the SIMD instruction set explosion.

Finally, any extension of SIMD instruction-set architectures should keep in mind that ISA complexity is a difficult challenge. Programming (or code generation) for SIMD is already a difficult proposition, so additions to ISA should be easy to generate code for (e.g. align with SVE codegen).

8.2 VarVE ISA Design

The VarVE ISA extension consists of two types of features, those that support 1) efficient packing/unpacking to load/store variable-width data and 2) efficient computation on variable-width data. In both cases, these designs leverage, and are designed to gracefully integrate with modern vector-length agnostic (VLA) architectures [105]. Two ISA variants are proposed. VarVE-pup is a minimal extension that contains only the pack/unpack feature to allow conventional SIMD easily load variable-width data for compute. VarVE-full is a full variable-width value extension that not only inherits the vpack/vunpack support for efficient loading, but also performs SIMD computation on native variable-width vector type.

8.2.1 Efficient Loading of Variable-width Values: Pack/Unpack

Data are diverse (ints, strings, enums, codes, etc.). Incoming values can feature arbitrary bitwidths that are packed or coded for storage or transfers. Existing SIMD ISAs require lengthy and complex bit operations to capture a stream of variable-width values, placing them into a vector register. In Figure 8.4, we show an example of unpacking (and loading) bitpacked values. ARM SVE need ∼10 instructions to first generate byte positions for each value, gather the values, and then do bit operations to mask out the extra bits gathered.

In VarVE-pup, we introduce a set of universal pack & unpack instructions to solve this problem (see first two rows of Table 8.1). These instructions convert between packed values in a stream to fixed-width elements in conventional SIMD vectors. An additional interface to describe input streams are also added, including fixed-width-value streams (FRStream), variable-width-value streams (VRStream) and huffman (HRStream) streams, as well as those for output streams: FWStream, VWStream, HWStream.

VarVE-pup instructions significantly reduce the instruction counts through ISA support as shown in Figure 8.4, to one instruction per iteration. And it enables performance improvement through microarchitecture optimizations like prefetch and preunpack [60]. Once a stream is specified with stream description instructions, unpacking could happen in the background and hiding its latency among the compute. Further, VarVE-pup instructions remove the roadblocks (performance penalty of loading) of using non-padded variable-width value format in storage and transfers, improving memory system efficiency compared to using padded fixed-width format.

VarVE-pup	
FRSTREAM	Specify an input fixwidth stream at a memory location
HRSTREAM	Specify an input huffman stream at a memory location
VRSTREAM	Specify an input variable-width stream at a memory location
FWSTREAM	Init an output fixwidth stream to a memory location
HWSTREAM	Init an output huffman stream to a memory location
VWSTREAM	Init an output variable-width stream to a memory location
CRSTREAM	Consume stream elements (lockstep stripmining)
ERSTREAM	Ending an input stream
EWSTREAM	Ending an output stream
UNPACK	Unpack a vector from an input stream and consume corresponding bits
PACK	Pack to output stream with vector values

VarVE-full		
Stream Description and Management		
FRSTREAM	Specify an input fixwidth stream at a memory location	
HRSTREAM	Specify an input huffman stream at a memory location	
VRSTREAM	Specify an input variable-width stream at a memory location	
FWSTREAM	Init an output fixwidth stream to a memory location	
HWSTREAM	Init an output huffman stream to a memory location	
VWSTREAM	Init an output variable-width stream to a memory location	
CRSTREAM	Consume stream elements (lockstep stripmining)	
ERSTREAM	Ending an input stream	
EWSTREAM	Ending an output stream	
VUNPACK	Populate vector from input stream but not consume	
VPACK	Pack to output stream with vector values	
Operations for Variable-width Value Vectors		
VADD	Add vectors	
VSUB	One vector substracts the other	
VMAX	Signed maximum	
VMIN	Signed minimum	
VMUL	Multiply vectors	
VDIV	One vector divide the other	
VAND	Btiwise and	
VBIC	Bitwise Vx and not Vy	
VERR	Bitwise xor	**Operand Variants** [31]:
VORR	Bitwise or	
VCMPEQ	Compare vectors: equal	Predicated Vectors
VCMPNE	Compare vectors: not equal	Unpredicated Vectors
VCMPLT	Compare vectors: less than	Predicated Immediate
VCMPLE	Compare vectors: less equal	Unpredicatd Immediate
VASR	Arithmetic shift right	
VASL	Arithmetic shift left	
VTVS	Fixwidth vector to VarWidth vector	
VCPY	Copy scalar to vector	
VDUP	Duplicate immediate to vector	
VINDEX	Index generation	
VNUM	Num of elements in vector	
VRCNUM	Read & clear NumE register	

Table 8.1: Summary of VarVE instructions

```
void sve_unpack(int w, int bits, const void *data) {
  // read 2B around each value. truncate and align
  for (int i = 0; i < bits / w; i += svcntw()) {
    svbool_t pg = svwhilelt_b32(i, bits);
    svuint32_t vid = svdup_u32(0);
    for (int j = 0; j < w; j++)
      vid = svadd_u32_m(pg, vid, svindex_u32(0, 1));
    svuint32_t vbid = svlsr_n_u32_m(pg, vid, 3);
    svint32_t vx = svld1sh_gather_offset_s32(pg, data, vbid);
    vid = svand_n_u32_m(pg, vid, 7);
    vx = svasr_s32_m(pg, vx, vid);
    vx = svand_n_s32_m(pg, vx, (1 << w) - 1);
    // vx parsed and aligned in 32b-per-element vector
  }
}
void varve_unpack(int w, int bits, const void *data) {
  int sid = frstream(data, w, bits);
  for (int i = 0; i < bits / w; i += svcntb()) {
    svint8_t vx = unpack_s8(sid);
    // vx parsed and aligned for compute
  }
}
```

Figure 8.4: SVE vs VarVE-pup: unpack and load bitpacked values

8.2.2 Efficient Computation on Variable-width Values: Native Vector Support

In VarVE-full, we leverage the high-level view pioneered by VLA SIMD (SVE), that abstracts away datapath width[1] (see Section 3.3). The key is that with the VLA approach, the same binary program works with different hardware datapath widths; the number of vector elements for each iteration is determined at runtime. Thus the datapath width still determines performance, but it is no longer embedded in the programs.

A new vector type where the bitwidth of each element in a single vector can vary, as shown in Figure 8.5, is added. This allow denser packing of variable-width values into vector registers. In terms of architectural states, variable-width values vectors reuse the existing vector registers. New vector metadata registers are added, one metadata register per vector register, storing the bitwidth of each element for the associated variable-width values vector.

1. This is a marked contrast to AVX or Neon where the datapath width is explicit in the ISA, and embedded in every program.

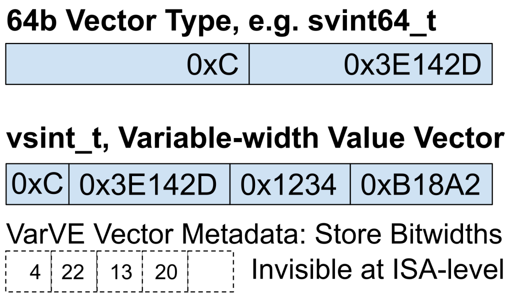

64b Vector Type, e.g. svint64_t
0xC
0x3E142D
vsint_t, Variable-width Value Vector
0xC
0x3E142D
0x1234
0xB18A2
VarVE Vector Metadata: Store Bitwidths
4
22
13
20
Invisible at ISA-level

Figure 8.5: Native support for variable-width values in VarVE-full

Vector metadata register are read/written when carrying out new vector operations defined below. No standalone instruction can read/write/change them.

Further, new vector operations defined on the variable-width-value vector type are added, as summarized in last five rows of Figure 8.13. These instructions still carry out element-wise operation between vectors. The number of output vector elements produced by each instruction is dynamic, as shown in Equation 8.1. The rule reflects that output vector elements cannot be more than the input vector elements in either operand. And if there is backpressure when adapting corresponding input vectors to the datapath to compute the output vectors, the number of output elements may further decline. This dynamic property gracefully hides into the VLA scheme, as we will show in Section 8.2.4.

$$E_{op(u,v)} <= min(E_u, E_v) \tag{8.1}$$

In VarVE-full, the pack & unpack instructions are upgraded as vpack & vunpack to target variable-width-value vectors. vunpack adds a scalar argument *mb* and a predicate argument *pred*. Argument *mb* constrains the maximum width vector elements can grow to (default is 64b). This maximum constraint is transitively applied through VarVE-full arithmetic operations. If the result of an operation is larger than the maximum, normal SIMD overflow behavior occurs. The *pred* argument is needed for an optimization discussed in Section 8.2.5.

The benefits of VarVE-full vector type and instruction design are two fold. First,

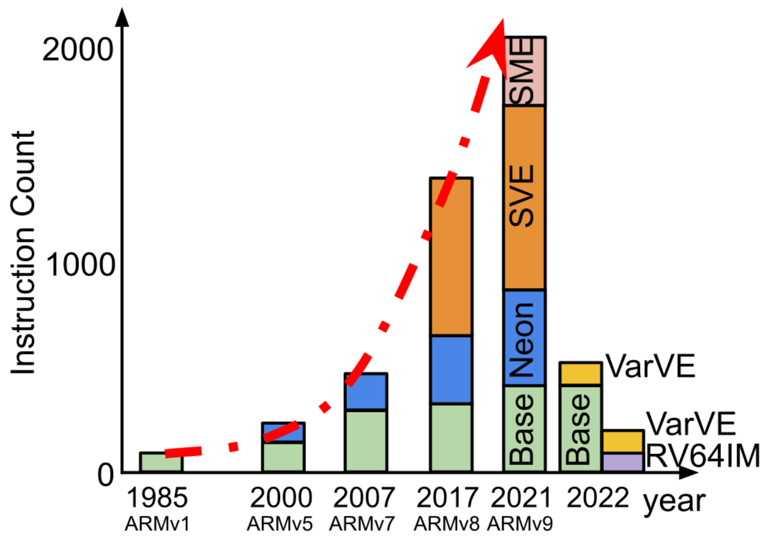


Figure 8.6: VarVE-full mitigates SIMD instruction count explosion

VarVE-full avoid VecReg wastes on padded values. Second, dynamic adapting of value vectors to datapath allows increased data-parallelism by putting more vector elements through the datapath each cycle.

8.2.3 Reversing the SIMD ISA Instruction Count Explosion

All SIMD ISA's (e.g. AVX, SVE) are suffering an instruction set count explosion. This is because not only are new operations continually being added to support BLAS2 (Matrix-vector) and BLAS3 (Matrix-matrix) operations [26, 72], but each of these instructions are *multiplied* by the number of required datatypes (int8, int16, int32, int64, fp8, fp16, fp32, fp64, bfloat16, and other new machine learning datatypes). This has produced the explosion in instructions depicted in Figure 8.6. For example, ARM, often called a RISC instruction set has over 1000 instructions, with at least another thousand more in planning. The growth of instructions presents several challenges – decode complexity (area and energy), instruction encoding space (causing extensions to 64-bit formats in RISC-V), and pipeline complexity (additional pipeline stages and interlock checking). The design of VarVE-full has the potential to help with this situation.

VarVE-full, as a flexible class of vector instructions, subsumes many fixed-width datatype instructions. For example, a single *vadd* instruction for variable-width values subsumes existing instructions for vector-add-int8, vector-add-int16, vector-add-int32, vector-add-int64. For the same semantics, a program need only select the appropriate maximum value size at initialization. This means those four instructions could be removed without any loss of expressive power. Thus, VarVE-fullrepresents a potential 4 to 8-fold reduction in many classes of SIMD instructions – while increasing expressive power to variable width vectors.

Because SIMD instructions are the largest and fastest growing part of modern instruction sets, VarVE-full has the potential to dramatically reduce the total number of instructions. For example, if the 800 SIMD instructions as depicted in Figure 8.6, could be reduced to 100 instructions, the overall instruction set size would be significantly reduced with concomitant benefits in decoding and pipelining described above. Perhaps as important, opcode space can be freed up for other customization such as machine learning datatypes (eg. bfloat16).

8.2.4 VarVE-full Strip Mining: Lockstep Advancing

The management of strip mining pacing is the responsibility of any VLA architecture. In existing VLA SIMDs, the strip size, i.e. the number of elements processed for one stream each iteration, is abstracted from the program. It is instead a static property of implementation hardware, and is a runtime constant in programming. This allows all VLA instructions in the strip mining loop to advances in *lockstep*, when processing multiple streams with element-wise correspondence among.

However, in VarVE-full, the strip size is no longer a runtime constant. the feasible strip size is dependent on the dynamic size of the values. In principle, the strip size could be different for each iteration of the strip mining loop. Further, the appropriate strip size may not be determined until the last instruction, as shown in Figure 8.7, where the effective strip size is successively constrained by the downstream instruction.

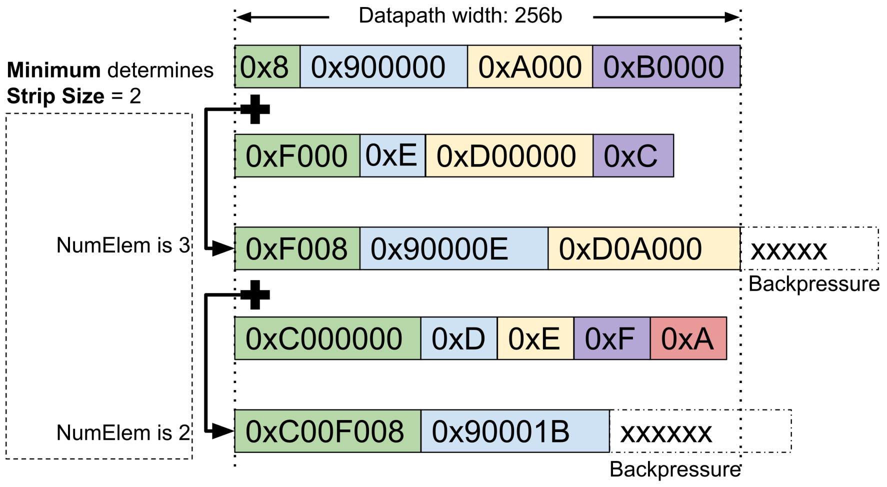


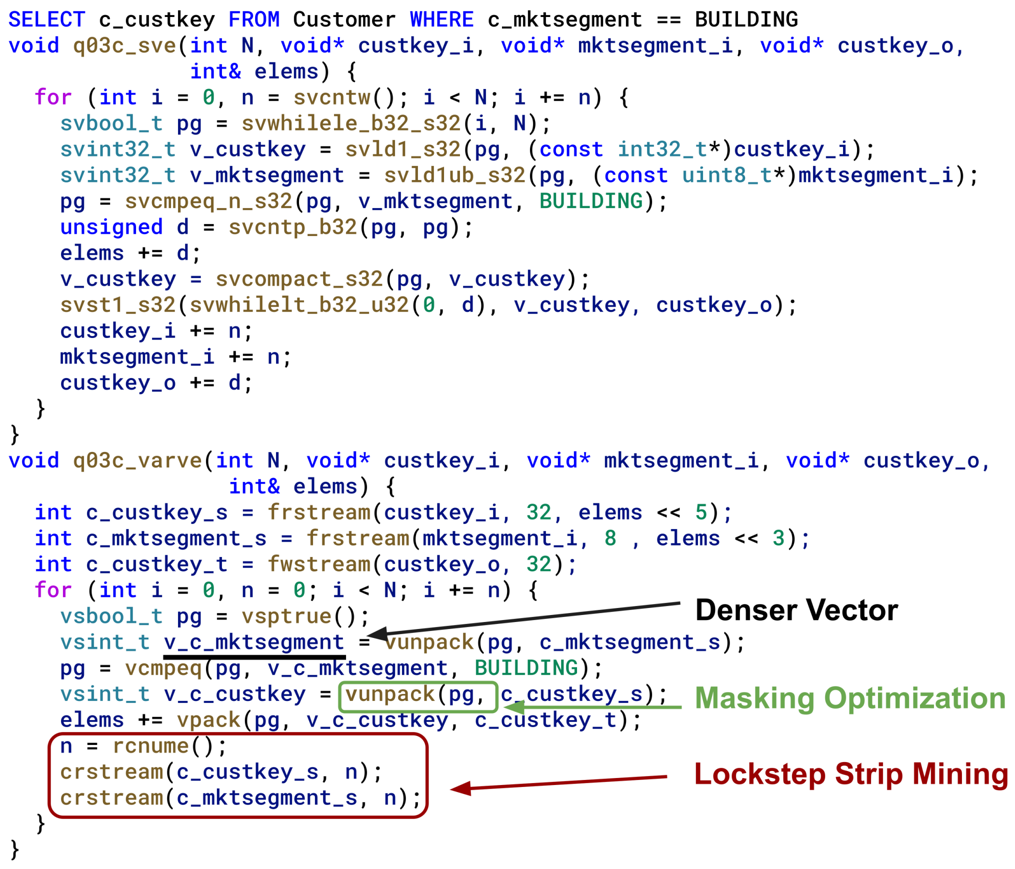
Figure 8.7: Strip size determined at the end of the iteration

To efficiently determine the maximum feasible strip size, we introduce the ‘NumE’ misc register. This register is updated by each of the VarVE-full instructions, using the min function. After a sequence of VarVE-full instructions in a a strip mine loop, it contains the lowest #elements produced by any operation, which is exactly the maximum feasible strip size to deliver lockstep advance, i.e. maintaining element-wise correspondence across streams. This ‘NumE’ register is similar to the ARM NZCV conditional flags [17].

For the overall scheme to work, the semantics of ‘vunpack’ are modified. It now just populates the vector with values from a stream, but no longer consumes the corresponding bits. At the end of the iteration, after the strip size/advance is determined, new ’crstream’ instructions would be called to advance the stream, consuming bits representing ‘NumE’ values, as shown in the end of Figure 8.8.

8.2.5 Masking Optimization

Figure 8.8 provides a concrete example of VarVE-full lockstep strip mining and its comparison with existing SVE strip mining on a filter kernel from the TPC-H benchmark [37]. It also exemplifies a new optimization opportunity in VarVE-full which we call ‘Masking Optimization’.



```
SELECT c_custkey FROM Customer WHERE c_mktsegment == BUILDING
void q03c_sve(int N, void* custkey_i, void* mktsegment_i, void* custkey_o,
              int& elems) {
  for (int i = 0, n = svcntw(); i < N; i += n) {
    svbool_t pg = svwhilele_b32_s32(i, N);
    svint32_t v_custkey = svld1_s32(pg, (const int32_t*)custkey_i);
    svint32_t v_mktsegment = svld1ub_s32(pg, (const uint8_t*)mktsegment_i);
    pg = svcmpeq_n_s32(pg, v_mktsegment, BUILDING);
    unsigned d = svcntp_b32(pg, pg);
    elems += d;
    v_custkey = svcompact_s32(pg, v_custkey);
    svst1_s32(svwhilelt_b32_u32(0, d), v_custkey, custkey_o);
    custkey_i += n;
    mktsegment_i += n;
    custkey_o += d;
  }
}
void q03c_varve(int N, void* custkey_i, void* mktsegment_i, void* custkey_o,
                 int& elems) {
  int c_custkey_s = frstream(custkey_i, 32, elems << 5);
  int c_mktsegment_s = frstream(mktsegment_i, 8 , elems << 3);
  int c_custkey_t = fwstream(custkey_o, 32);
  for (int i = 0, n = 0; i < N; i += n) {
    vsbool_t pg = vsptrue();
    vsint_t v_c_mktsegment = vunpack(pg, c_mktsegment_s);
    pg = vcmpeq(pg, v_c_mktsegment, BUILDING);
    vsint_t v_c_custkey = vunpack(pg, c_custkey_s);
    elems += vpack(pg, v_c_custkey, c_custkey_t);
    n = rcnume();
    crstream(c_custkey_s, n);
    crstream(c_mktsegment_s, n);
  }
}
```

Figure 8.8: TPC-H filtering: SVE vs VarVE-full

One problem with VarVE-full is that one stream with wide values (in bitwidths) can slow an entire loop. That's because the minimum #elements of all vector operations will determine the rate of progress (lockstep value) for the entire loop. Thus the fixed-width datapath will be poorly utilized.

We observe that sometimes if we have a mask for a slow input stream, we can accelerate it by pushing the mask into the stream. This idea is akin to predicate push-down in database queries. VarVE-full allows the mask to be supplied to the vunpack instruction (as in Figure 8.8, for v_c_custkey) to transform unneeded values to zeroes, such that they consume neither vector register bits, nor datapath bit positions in later computation. This optimization would produce larger strip size and thus higher datapath efficiency.

This optimization is particularly useful when mask/filtering could be calculated from

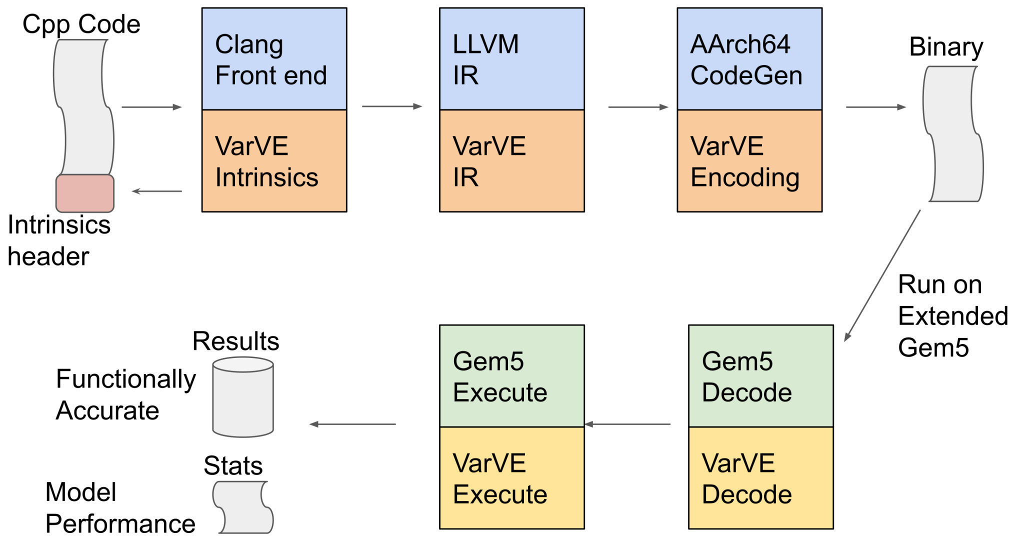


Figure 8.9: Compilation and Simulation Infrastructure

input streams with narrower (on average) values, and then the mask is applied to the unpacking of streams with wider values. With masking optimization, the reduction of strip size during instructions operating wider values could be smaller or even zero.

8.3 Evaluation

8.3.1 Methodology

Modeling and Software Infrastructure Gem5 [46] is extended with VarVE instructions as shown in Figure 8.9. Gem5 ARM-Ex5 in-order core model [8] is employed as the performance model. The memory hierarchy is 8-way 32KB L1I, 8-way 32KB L1D, 16-way 256KB L2, all with 64B cache blocks and a DDR4-4800 16GB DRAM. Since most of our workloads feature streaming accesses [117], caches have little impact. Further, we added intrinsic support (for C/C++) and code generation for VarVE instructions to LLVM [85], using unused opcodes in the ARM SVE encoding space.

ISA Variants Four ARM ISA variants, AArch64 Scalar, SVE, VarVE-pup, VarVE-full are evaluated. For three SIMD variants, the datapath widths (i.e. vector length), are all

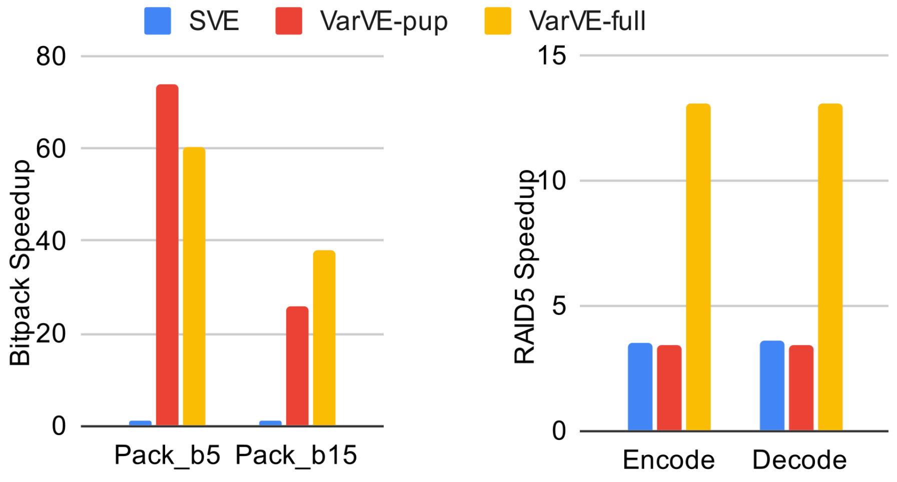


Figure 8.10: Speedup for bitpacking and RAID5 (vs. Scalar)

256 bits. As a quick recap, VarVE-pup is a partial extension that only adds the pack/unpack instruction pair converting between packed streams and conventional vectors with fixed-bitwidth elements as discussed in Section 8.2.1 (see Table 8.1). VarVE-full features full-fledged vector support of variable-width values, as well as the pack/unpack instruction pair converting between streams and vectors with variable-width values (see Table 8.1). VarVE-full's model considers the pressure from hardware (Section 8.4.2), that corresponding elements from two input vector would align to the next 8-bit boundary. The performance modeled include the negative effects resulting from this implementation limit.

Workloads Workloads include file system kernels of bitpacking and RAID5 erasure coding, database analytics kernels of filters extracted from TPC-H [37] queries, and a neural network inference kernel to show VarVE's generality.

Memory efficiency improvement is investigated if the input data exploits the similar principle: eliminating bit wastes on zeros through packing. Performance scalability with increasing datapath width is also investigated. And we show that VarVE's performance gain is robust with different datapath widths.

Metrics

- **Runtime:** Execution time based on Gem5 cycle-accurate simulation
- **Speedup:** ratio of scalar execution time to that of one SIMD implementation
- **Instruction Count:** Number of executed instructions.
- **Memory Traffic:** Bytes read/written from/to DRAM

8.3.2 File System Kernel: Bitpacking and RAID5

We study VarVE, using filesystem kernels as exemplar applications optimized for efficient information encoding, where the use of variable-sized values is a critical technique for efficient use of storage and memory. In Figure 8.10, we show the speedups for bitpacking and RAID5 erasure coding, compared to scalar implementation.

In the bitpacking evaluation, two TPC-H columns 'p_brand' and 'o_orderdate' which are 8-bit and 32-bit unpacked integers respectively are bitpacked to fixwidth streams of 5-bit or 15-bit per value. Lack of byte-unaligned (bit-level) scatter support, bitpack implementation in SVE has to fall back to the Scalar implementation, achieving the same performance as the Scalar baseline. **VarVE-pup matches this workload well, and achieves 78x and 26x speedup** respectively through vectorization, ISA and microarchitecture support of latency-hiding write-path packing (the duo of preunpacking). The 3x differences in speedup for two columns results from the input type width: More elements can be processed in one iteration for narrower input type. **VarVE-full further improves the vectorization density for the 15-bit per value bitpacking, achieving 38x speedup.** For the 5-bit per value bitpacking, even with the same vectorization density as VarVE-pup, **VarVE-full was limited to 60x speedup over Scalar by the additional auxiliary work (determine advance and consume input streams) in the lockstep strip mining.**

For RAID5, unpacked data from 'l_linenumber' in 32-bit integers are streamed in, generating additional parity blocks. The encoding and decoding processes are duo of each

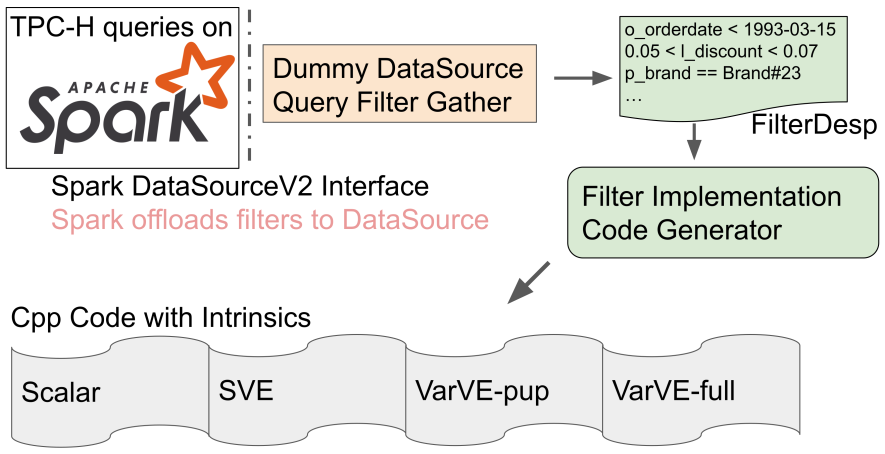


Figure 8.11: TPC-H filters offload structure

other, thus the performance of each ISA variant are roughly the same across the two processes. Because inputs and outputs are already unpacked 32-bit integers, VarVE-pup, with no unpacking and packing acceleration benefits to exploit, performs roughly the same with SVE at 3.5x speedup over Scalar. This speedup comes from wider datapath (256b vs. 32b), less the auxiliary work of the conventional strip mining. VarVE-full benefits from denser usage of vector register and vector ALU. Despite the schema definition of 'int', values in the column never use full 32 bits but average at 8 bits. As a result, **VarVE-full first 'vunpack' more elements from input into dense vectors, do the parity computation, and then 'vpack' back to the original thin form stream, achieving 13.1x speedup over Scalar, 3.7x over SVE or VarVE-pup**.

8.3.3 Database Analytics Kernel: TPC-H Filters

Then we evaluate the performance of four ISAs on the filter kernels offloaded from TPC-H data analytics workloads that is studied in the majority of computational SSD work [57, 65, 73, 75, 78, 79, 82, 86, 99, 102, 108, 113]. As shown in Figure 8.11, the filters are collected from the Spark DataSource interface where a large variety of filters [29] are pushed down to data sources. We implemented an ad hoc code generator with functionality similar to

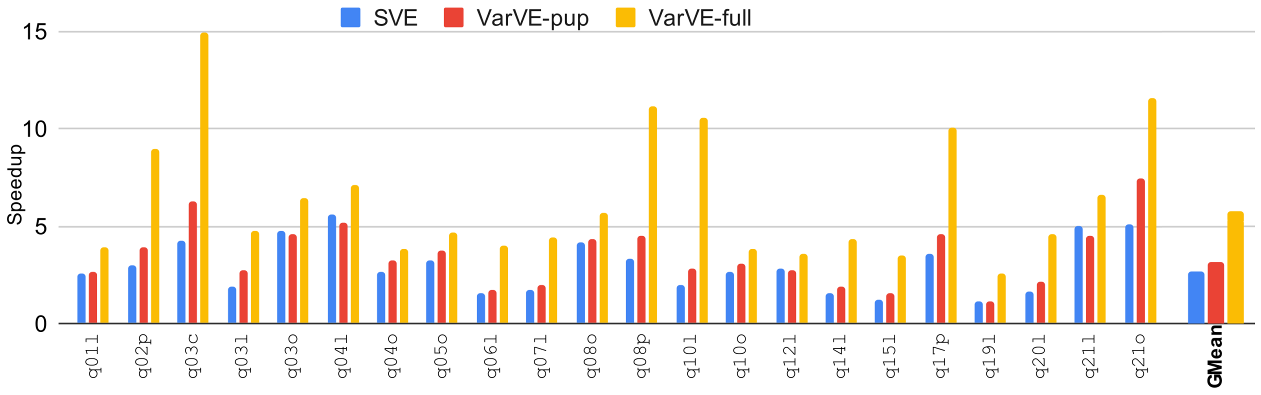


Figure 8.12: Speedup for TPC-H filters. TaskName = QueryID + TableNameLeadingCharacter (vs. Scalar)

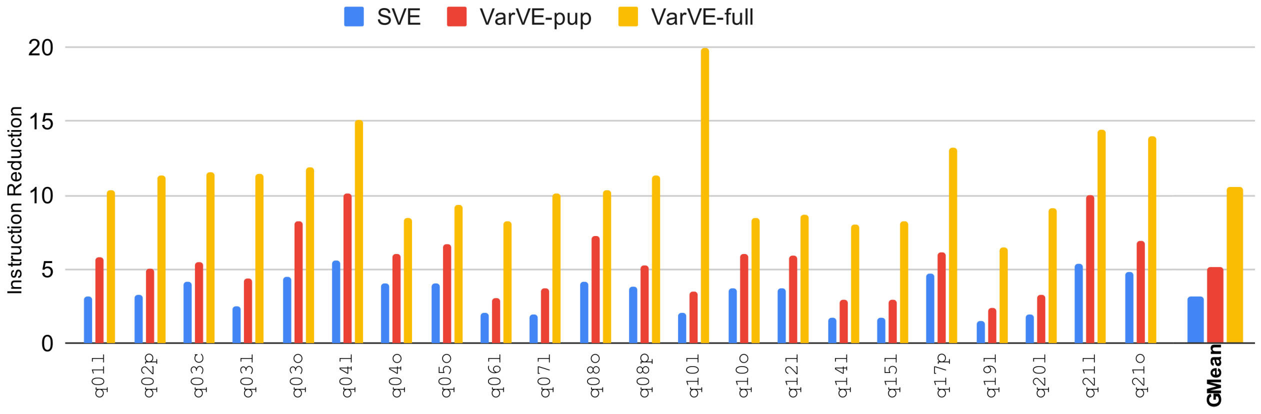


Figure 8.13: Instruction count reduction (vs. Scalar)

that of Spark Tungsten [38] to generate four variants of filter implementations in C++ with corresponding intrinsics. All input data are unpacked as types defined by TPC-H schema [37].

Figure 8.12 and Figure 8.13 depict the speedup and dynamic instruction count reductions over that of the Scalar implementation respectively. **VarVE-pup achieves 1.2x - 7.5x speedup over Scalar, which is 1.5x over SVE by GeoMean.** This performance benefits come from elimination of pointer increments. **VarVE-full achieves 2.6x - 15x speedup which averaged (GeoMean) at 5.7x over Scalar. This is also 1.3x - 5.4x speedup over SVE which averaged (GeoMean) at 2.1x.** The performance benefits come from the denser representation that fits more elements in the SIMD width (256 bits). And VarVE-full also brings the benefits of masking optimization as discussed in Section 8.2.5

for multi-predicate filtering. The values for the rows filtered out in the first predicate are set to zero during predicated unpack of the second column, saving VecReg and datapath bits. Further, in terms of instruction count, **VarVE-full is 10x reduction from Scalar, and 3.4x from SVE.** This showcase the instruction expressiveness of VarVE-full that one VarVE-full instruction on average specify 3.4x more operations than SVE.

8.3.4 Neural Network Inference

Here we evaluate a neural network inference workload on MNIST [56] classification to show that VarVE's benefits are broadly applicable to general SIMD computing. This neural network is small but representative as it covers important layer types like Conv2D, Relu, MaxPooling, FullyConnected as shown in Table 8.2. It is trained with quantization awareness [35] in float64. The trained model is then quantized to 32-bit integer while retaining 97.7% inference accuracy. Separately, the same trained model is first pruned [34] to 80% sparsity in its weights, and then quantized to 32-bit integers while retaining 97.58% inference accuracy.

Figure 8.14 depicts the speedup of three SIMD variants on inference with either the quantized or the pruned model over the inference of Scalar on the quantized model. For quantized model, **VarVE-pup delivers 2.7x speedup over Scalar through vectorization.** But that's only 3% performance edge over SVE, as inputs are already unpacked data, VarVE-pup only saves on stream pointer increments. **VarVE-full achieves 3.7x speedup over Scalar, that is 1.4x over SVE and VarVE-pup.** The performance edge comes from higher datapath efficiency of native variable-width value vector support. For the pruned & quantized model, SVE and VarVE-pup cannot benefit from the sparsity of weights as the conventional predicated execution does not improve the datapath efficiency. Also, going for an exotic execution schedule trying to skip spontaneous zero weights creates divergence and hurts performance. As a result, their performance doesn't change. On the other hand, VarVE-full can exploit the sparsity in weights with the normal computation schedule as those

Layer	**Configuration**	**OutputShape**
Input		(28, 28, 1)
Conv2D	kernel: (3, 3), padding: 'VALID'	(26, 26, 32)
Relu		(26, 26, 32)
MaxPooling2D	pool: (2, 2)	(13, 13, 32)
Flatten		(5408,)
Dense	weights: (5408, 10)	(10,)

Table 8.2: Architecture of the MNIST neural network

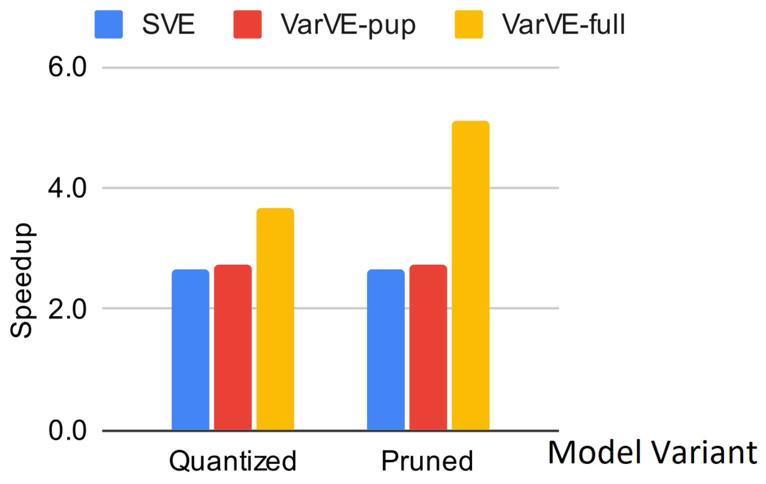


Figure 8.14: Inference speedup (vs. Scalar on Quantized)

zero weights does not consume VecReg/VecALU bits just like the situation in the masking optimization as shown in Section 8.2.5. Thus, **VarVE-full improves inference speedup to 5.1x over Scalar, and that is 2x better than SVE or VarVE-pup.**

8.3.5 Memory Traffic Improvement

All previous evaluation assumes that inputs are unpacked data, i.e. each value is padded to 8, 16, 32, or 64 bits. As observed in Section 8.1.1 and Figure 8.1, padded variable-width values would not only incur efficiency loss in datapaths, but also in memory systems (both main memory and caches). With VarVE's ability to load and process diverse values, data are no longer needed to be preprocessed (decoded and padded) before employing SIMD accelerations. In this section, we investigate what kinds of memory system efficiency improvement

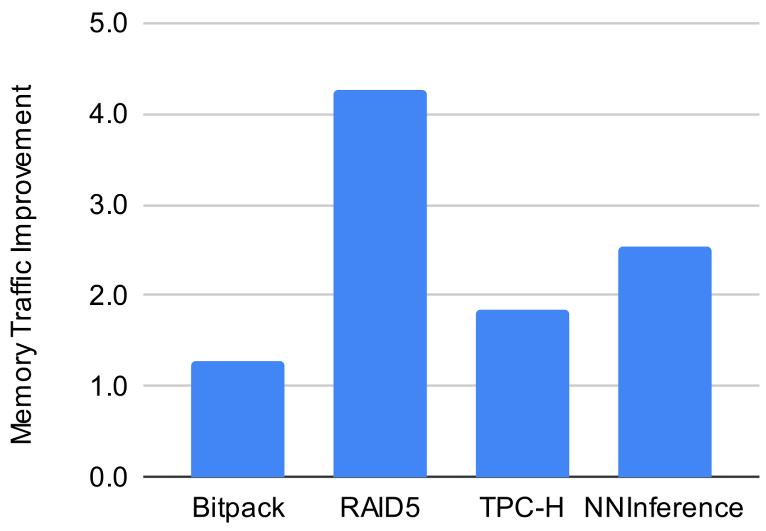


Figure 8.15: Memory traffic improvement (vs. padded input)

can be delivered, if the input data are in VarWidth streams, which is a format similar to VarVE-full vectors: A tightly bitpacked data stream associated with a metadata stream, where each byte in the metadata stream marks the bitwidth for a value in the data stream. The only difference between a VarWidth stream and the data stored in a VarVE-full vector register and vector metadata register pair is that the VarWidth bitstream can grow indefinitely to store as many values as needed, while the number of values stored in VarWidth vector is limited by the vector datapath width (vector register width).

Figure 8.15 shows that input in **VarWidth streams could achieve 1.3x-4.3x memory traffic improvement** compared to the scenario where inputs are unpacked padded data. The improvement comes from the elimination of the wastes on bits that are zeros in memory system traffic (main memory to cache, and cache to registers), as shown in Figure 8.1, less the consumption of bits used in metadata.

Further, we would like to point out that we deliberately use unpacked (i.e. padded) data as inputs in previous experiments to focus the evaluation on core datapath efficiency. If packed data like VarWidth streams are used as inputs, the performance of SVE would take a huge toll for unpacking those values from streams, as we have seen in the bitpacking evaluation. On the other hand, the performance of VarVE would be mostly unchanged, with the ISA and hardware support of packing and unpacking (Section 8.2.1), as long as the prefetching could keep up with the computation pipeline.

8.3.6 VarVE Performance vs Datapath width

In this section, we further investigate how VarVE performance gain changes as the datapath width is increased. This is enabled by vector-length agnosticism of SVE and VarVE. That is, the same VarVE program can be run on implementations with different datapath widths.

Figure 8.16 demonstrates the GeoMean speedup of three SIMD variants with different datapath widths over Scalar on TPC-H filtering workloads. The labels on top of the bars for VarVE-pup and VarVE-full further present their respective speedup over the SVE. As shown in the figure, **VarVE-full's speedup scales linearly with the datapath width. And it also provides robust speedup (i.e. 2.2x) against SVE,** the state-of-the-art vector-length agnostic SIMD ISA. Performance gain still come from the the higher datapath efficiency with VarVE-full's native supports for variable-width values. In fact, if thinking abstractly, higher datapath widths may actually help reduce the ratio of leftover bits in VarVE-full.

8.3.7 Summary

Evaluation using kernels from multiple domains including file system, data analytics and neural network inference suggest that VarVE delivers significant performance increase. These improvements arise from greater datapath efficiency and memory traffic reduction. These results show the promise for more general computing. Comparing VarVE-pup with VarVE-full, we find that even sophisticated pack/unpack is not sufficient to capture this benefit. Native support for variable-width values is the key to the greatest performance benefits.

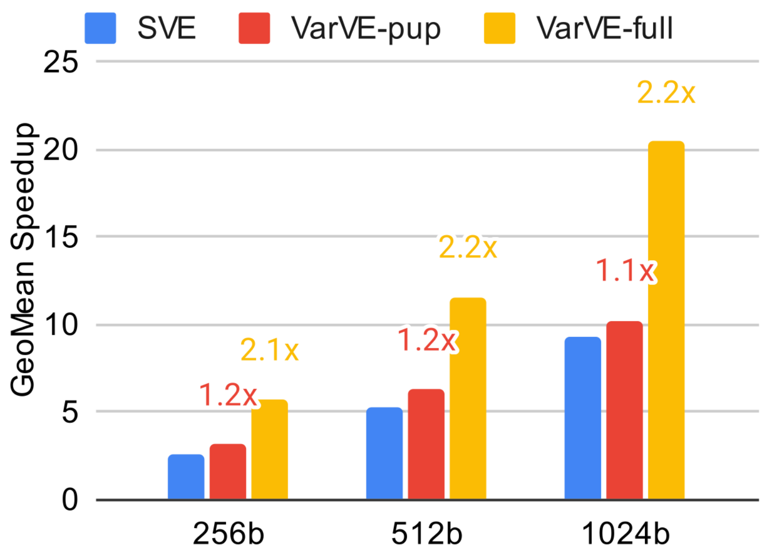


Figure 8.16: TPC-H filter speedup with varying datapath widths

8.4 Implementing VarVE-full

Our goal of this section is to show that VarVE-full could be implemented with same clock frequency and limited area/power increase, compared to conventional SIMD engines.

8.4.1 VarVE-full Microarchitecture

A representative SIMD microarchitecture based on previous work [44, 51, 94, 95, 104], is shown by light orange components in Figure 8.17. Generally, the vector register(VecReg) provide operands for the the vector ALU (VecALU) and accepts the output vector from the VecALU.

As discussed in Section 8.2.2, vector metadata registers are added in VarVE-full, one per vector register, to store the bitwidth of each vector element in the vector storing variable-width values. Vector metadata registers accept writes from VecALU and service reads for VecALU (through EAU, discussed later in Section 8.4.2) by VarVE-full instructions.

Moreover, because elements are of variable widths, corresponding elements between two vectors in a vec-vec operation are no longer aligned. VarVE-full adds an Element Alignment

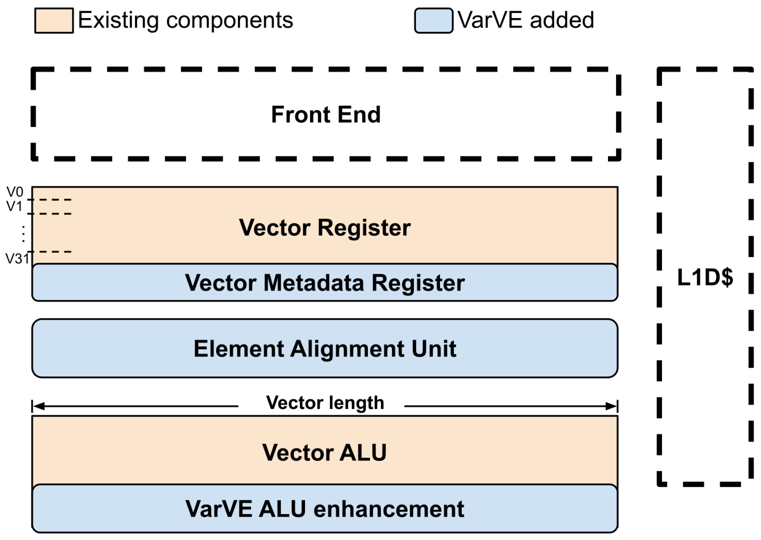


Figure 8.17: Microarchitecture for SIMD and VarVE-full

Instruction	**Width**
Add/Sub	$Max(W_a, W_b) + 1$
Mul	$W_a + W_b$
Logical	$Max(W_a, W_b)$
Comparison	$Max(W_a, W_b)$

Table 8.3: Alignment width, two inputs denoted as a and b

Unit (EAU) as a VecALU pipeline step as shown in Figure 8.17.

Finally, to provide operations on two elements aligned at an arbitrary bit position, the vector ALU has to be enhanced, especially for those operations that involves passing carry signals across element boundaries, as the element boundaries are now dynamic with variable-width values. We will concentrate on these two changes in hardware for VarVE-full, to show that VarVE-full is feasible in hardware implementation.

8.4.2 Element Alignment Unit

The Element Alignment Unit (EAU) aligns the corresponding elements from two variable-width-value vectors to compute in VecALU. The alignment process not only considers value widths from two vectors, but also considers intermediate results after applying the operations specified

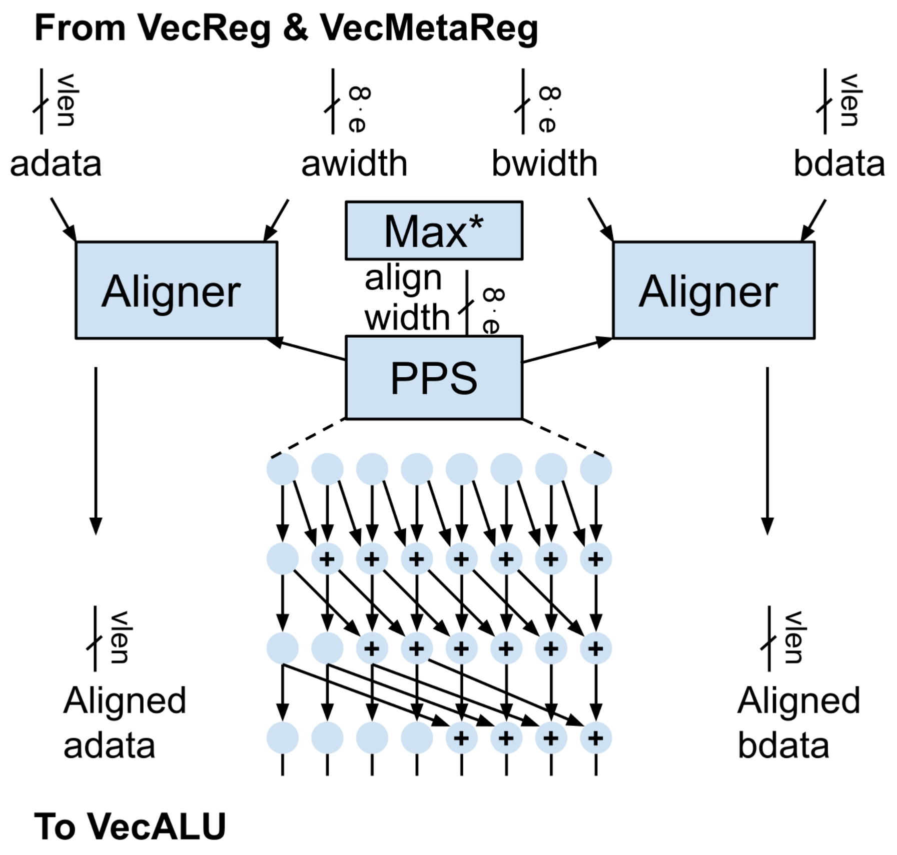


Figure 8.18: Element Alignment Unit

by the instructions. We summarize the rules of alignment width for different VarVE-full instructions in Table 8.3.

Figure 8.18 depicts how the EAU is implemented in hardware. A vanilla Parallel Prefix Sum (PPS) unit is used to calculate element boundaries from the alignment widths determined according to Table 8.3. And vector elements from each vector are aligned on these boundaries in an Aligner, which is mostly a hardware gather (i.e. per-position MUX) on the input vector.

To reduce circuit implementation cost and circuit latency, we opt to align vector elements only to rounded-up 8-bit boundaries. This reduces the circuit scale of the PPS unit and the MUXes in the Aligner by 8x and 64x respectively, compared to bit-level alignment. We closed the timing of EAU implemented in SystemVerilog at 1GHz on SAED 14nm silicon technology [25] via Design Compiler, fast enough as a pipeline step in a SIMD unit, e.g. in Ara [51]. Please notice that the round-up of alignment is invisible to the software

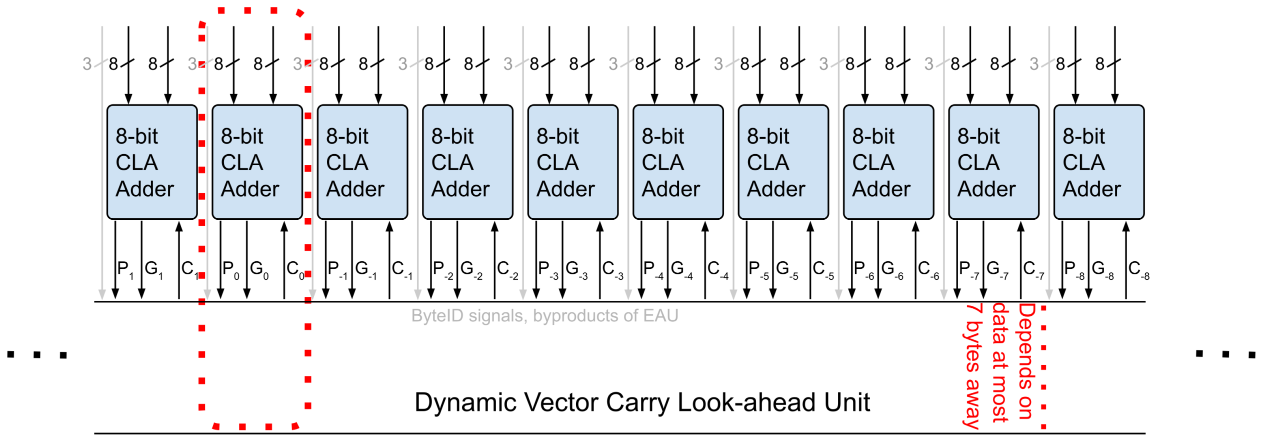

8-bit CLA Adder
P_1 G_1 C_1
P_0 G_0 C_0
P_-1 G_-1 C_-1
P_-8 G_-8 C_-8
ByteID signals, byproducts of EAU
Dynamic Vector Carry Look-ahead Unit
Depends on data at most 7 bytes away

Figure 8.19: Dynamic Vector Carry Look-ahead Adder (subscriptions follow the order of significance)

programming, because the strip mining process is already dynamic depending on the values processed, as discussed in Section 8.2.4. The round-up of alignment hides delicately as if each value is slightly larger. For the same reason, other and future implementations is free to choose finer alignment granularity without worries of breaking ISA promises. This is a similar mechanism as the vector-length agnosticism [22, 105] that allows implementation freedom while maintaining the compatibility of an already-compiled binary program.

With the EAU, instructions without cross-bit carries can be implemented with conventional VecALU unchanged. This is a significant milestone on hardware feasibility of VarVE-full.

8.4.3 Dynamic Carry Look-ahead Adder

Adaptations are needed for instructions that have carries across the vector to account for the fact that element boundaries are now dynamic depending on the sizes of variable-width values in each vector. Specifically, we dive deep into the adder hardware implementation here as an example. The adapted carry-lookahead vector adder is drawn in Figure 8.19.

In traditional SIMD like AVX or state-of-the-art VLA SIMD like ARM SVE, the carry-lookahead unit for vector adding should already be runtime-configurable, because vector element type could be byte, half, word, double -sized. As a result, there are four different logics to choose

	Size/ByteID	**Carry Logic** ‘|’ denotes ‘logical or’
AVX	Byte	0
NEON	Half	G_{-1}
SVE	Word	$G_{-1} \mid P_{-1}G_{-2} \mid P_{-1}P_{-2}G_{-3}$
(MSB)	Double	$\|_{i=1}^{7} G_{-i}\Pi_{j=1}^{i-1}P_{-j}$
VarVE-full	0	0
	1	G_-1
	2	$G_{-1} \mid P_{-1}G_{-2}$
	3	$G_{-1} \mid P_{-1}G_{-2} \mid P_{-1}P_{-2}G_{-3}$
	4	$\|_{i=1}^{4} G_{-i}\Pi_{j=1}^{i-1}P_{-j}$
	5	$\|_{i=1}^{5} G_{-i}\Pi_{j=1}^{i-1}P_{-j}$
	6	$\|_{i=1}^{6} G_{-i}\Pi_{j=1}^{i-1}P_{-j}$
	7	$\|_{i=1}^{7} G_{-i}\Pi_{j=1}^{i-1}P_{-j}$

Table 8.4: Carry input logic for each 8-bit adder
P_i and G_i are propagate and generate signals [12]

from when calculating the input carry signal for each 8-bit adder, as summarized in Table 8.4 with the logic for the most significant byte (MSB) shown. For VarVE-full, this list grows to eight different variants, with the selection signals being the in-element ByteIDs for each 8-bit adder position, which are byproducts of the Element Alignment Unit. Please notice that the carry at most comes from the 8-bit adder 7 bytes away, which is the same with conventional SIMD or VLA SIMD ISAs. As a result, the adaptations should have limited impacts on the circuit critical path, timing or silicon area/power.

8.5 Summary

Existing processor architectures suffer from low datapath efficiency when computing on variable-width values for requiring padding each value to 64 bits. However, variable-width values are central to coding and storage efficiency, and thus critical for computational storage.

Existing SIMDs partially address the low-efficiency problem by providing four vector types. Not only does single instruction would specify more operations, thus reducing the instruction traffic but also vector types allow better tailored compute to be specified for dynamic values. However, conservative programming accounting for outliers still lead to padding for values, leading to low datapath efficiency.

We propose VarVE, a vector instruction set architecture extension that provides native variable-width value vector support to compute directly without padding. VarVE delivers 1.3x - 5.4x speedup over ARM's current best, scalable vector extension (SVE), on popular file system and database computational SSD kernels by delivering higher datapath efficiency. And that is 3x - 15x over 64bit Scalar with 6x GeoMean, 50% higher than the 4x datapath width improvement, signifying significant compute efficiency improvement. As will be shown in Section 9.1, SIMD approaches, especially VarVE, are compatible with our ASSASIN SSD architecture, and its streaming memory hierarchy, providing multiplicative performance benefits. And VarVE combined with ASSASIN elevate the in-SSD compute performance to match and exceed the flash array bandwidth.

Further, VarVE builds on the vector-length agnostic (VLA) approach, which is gaining widespread adoption. As a result, VarVE has broad potential impact as a general SIMD extension for all processors, increasing datapath efficiency and mitigating the SIMD instruction count explosion.

CHAPTER 9
DISCUSSION

In this section, we aim to discuss three matters. First, how does ASSASIN and VarVE combine and whether the combined general-pupose computational SSD deliver high performance and high efficiency at the same time? Second, what are the implications of ASSASIN and VarVE towards emerging computational SSD products and NVMe specifications? Third, how does ASSASIN and VarVE apply beyond computational SSDs?

9.1 Putting it Together

9.1.1 Multiplicative Performance

Both VarVE (or generally most SIMDs) and ASSASIN target streaming workloads. Thus, VarVE equipped processor is compatible to the ASSASIN scheme to compute on top of the streams from/to the flash array, bypassing the SSD DRAM Apparently, ASSASIN cores have to be upgraded to support VarVE ISA extension, as shown in Figure 9.1. Caches are already with cache blocks of 64B, providing enough bandwidth for VarVE. And, as shown in Section 7.3.6, streambuffers, timing of which are closed at 1GHz, are already provisioned for 64B-wide accesses. As a result, the memory system is also ready for the introduction of VarVE.

Thus, VarVE would bring multiplicative benefits to ASSASIN SSD. We summarized the

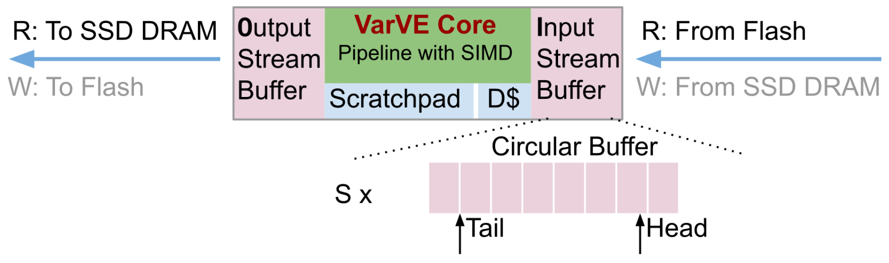


Figure 9.1: VarVE equipped ASSASIN core

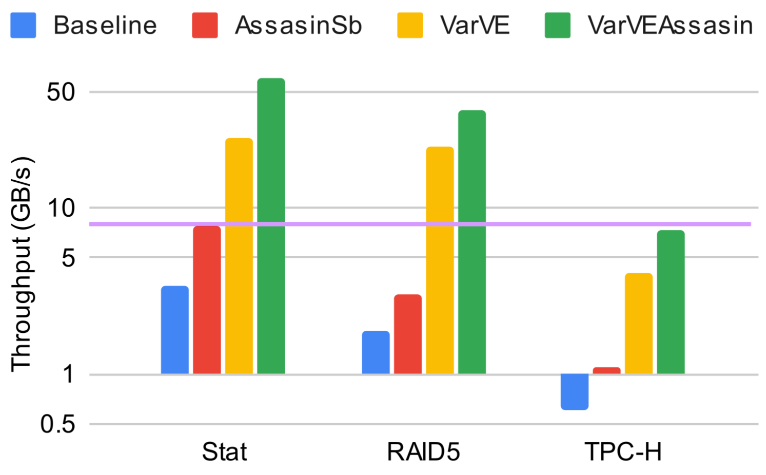


Figure 9.2: Performance of putting ASSASIN and VarVE together

performance comparisons in Figure 9.2, where we assume there are 8 cores, datapath width of each core is 256b and the flash array bandwidth capacity is infinite. If we further consider the flash array throughput limit, it just bounds the performance at the flash array throughput (e.g. 8 GB/s, the green bar in the figure) if the performance shown in Figure is higher.

As one can gather from the figure, VarVE combined with ASSASIN deliver the compute performance that matches and exceeds the throughput of the flash array, with general architecture support and high flexibility from programmability maintained, fulfilling our goal of the dissertation project.

9.1.2 Cost in Silicon Area and Power

We project VarVEAssasin's cost in silicon area and power, based on Ara [51] and Ariane [115], as shown in Table 9.1. As we can see, 8 core VarVEAssasin each with 256b datapath consume only 1w. This power consumption hides well under the ∼5w power budget of an SSD device. But as depicted in Figure 9.3, this small power and silicon area cost translates to 17x speedup, 4x power efficiency and 32x area efficiency compared to scalar baseline we evaluated in ASSASIN.

	Assasin	VarVE 256b	8 VarVEAssasin core
Area (mm^2)	0.203	0.303	4.1
Power (mW)	22	103.111	1000.9

Table 9.1: Area and Power for ASSASIN Core, VarVE Core, and 8 VarVEAssasin Core

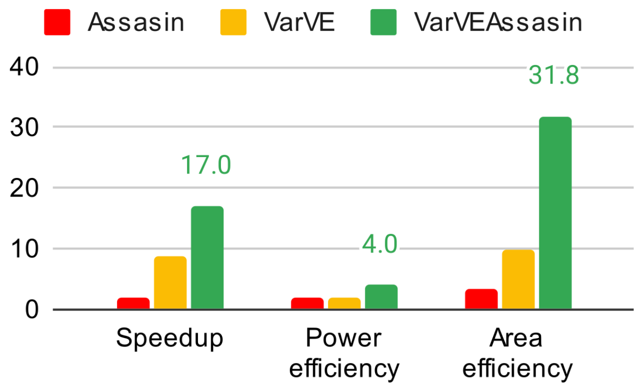


Figure 9.3: Speedup, area efficiency and power efficiency vs ASSASIN's scalar baseline

9.2 Implications for NVMe and Computational SSD Products

NVMe 2.0 [16] added a new copy command, and comment set support for key value SSD devices (e.g. KVSSD [80]) among other changes. The new copy command could be seen as the first computational storage function standardized in NVMe spec, despite of its simple functionality. And the new KV command set is an extension to SSDs' current block interface to support native key-value storage inteface. In some sense, it could be seen as the offload of the adapter between the SSD block interface and the key-value storage interface required by upper-level applications, originally implemented in the storage engine software. Further, SNIA is working on a list [64] of computational storage functions which currently include compress, filter, encrypt, erasure code, regex, pipline, hash, deduplication. These functions could be the candidates for future additions to NVMe command space.

The architecture pursued in this dissertation is a direct contrast to these standardization efforts. ASSASIN and VarVE pursue general-purpose high performance architectures, where each ISA is designed to provide programmability that allows the offloads of arbitrary functions. While the standardization efforts are pursuing incremental extensions with fixed functionality commands, which largely limits the offload freedom on application.

We venture general support is the way to encourage adoption. As shown in Section 7.2.5, only one command 'compute' is needed to be added to the NVMe command space to cover general computation, with diverse programs sent as arguments of the 'compute' NVMe command. This not only largely simplifies the NVMe command interface but also preserves application's freedom to offload diverse functions.

9.3 Beyond Computational SSD

9.3.1 ASSASIN for Stream Computing

As discussed in Section 7.2.5, with control and data plane separation, ASSASIN cores do not assume the responsibility of flash data layout and management. The firmware processor takes care of flash data layout, wear-level management and so on. ASSASIN cores only know about and compute on the streams formed by firmware, just like the response to conventional read requests prepared by SSD firmware. Thus, the compute and DRAM bypassing ability of ASSASIN are agnostic to any facility or constraint from the storage/flash array.

As a result, the ASSASIN design can be applied to any computing scenario that features lots of streaming accesses. ASSASIN process these streams with only streambuffers, bypassing caches and main memory, reducing memory traffics and cache thrashing. The streambuffer design is also a lower-latency higher-efficiency memory hierarchy with the priori knowledge of accessing only the stream heads compared to the cache design.

Specifically, ASSASIN can be coupled with the SIMD approaches targeting streaming data. The StreamLoad/StreamStore instruction semantics couple well with the strip mining programming model of both conventional fix-datapath-width SIMDs and the state of the art vector-length agnostic (VLA) SIMDs. And this instruction pair, coupled with the streaming hierarchy, could reduce the overheads of managing the loop and incrementing pointers, as shown in a similar proposition UVE [60].

Further, there are software frameworks for distributed systems [11, 50] exploiting streaming computing paradigms. These software frameworks organize input data as stream of events and arrange computation specified as operators on top of dataflows of streams. ASSASIN can provide architectural support for these stream dataflows workloads, bypassing cache hierarchies and provides low-latency accesses.

9.3.2 VarVE for General SIMD Computing

As discussed in Section 8.1, real world data are more diverse than just being 32 bits or 64 bits. The diversity most comes from the biases, precisions and bounds in the quantization process. As a result general computing tackle variable-width values all the time. Good programming practice advocates selecting types conservatively, accounting for outliers in inputs, outputs and even intermediate results. And this applies to programming for both Scalar and SIMD architectures, leading to to padding and thus datapath wastes in general computing.

VarVE is designed to handle general SIMD computing. This is partly showcased in Section 8.3.4 where VarVE is applied to neural network inference workload and VarVE showcase its capabilities to exploit sparsity in weights of the pruned neural network as a special case of variable-width values. With its root in vector-length agnostic SIMD and similar striping model as discussed in Section 8.2.4, VarVE is expected to be applicable to other workloads in general SIMD computing, as general as VLA SIMD ISA like RISC-V Vector and ARM SVE, to bring improvement in datapath efficiency for computing on variable-width values.

CHAPTER 10
SUMMARY AND FUTURE WORK

10.1 Summary

In this dissertation, we addressed three fundamental questions on architecting general-purpose computational storage. First, what are key opportunities for computational storage acceleration? Second, what SSD architecture provides the most efficient support for computational storage? Third, what computational storage processor architecture enables high performance that can match the continued rapid improvement in flash bandwidth?

Based on a broad survey of computational storage proposals and research, we show that function properties of 'data size change', 'offload direction' and 'vectorizable' are indicators of the system efficiency and compute speedups of computational SSD offloads and thus determine the priority of offloading. The diversity of the functions identified as best offloads call for the general-purpose high performance architectures. With workload insights that best functions feature streaming accesses, we propose ASSASIN SSD architecture that provides efficient integration of compute into SSDs, and allow flash data traffics bypassing SSD DRAM, mitigating the SSD DRAM memory wall. Further, VarVE is proposed to bring native vector support for variable-width values, which are popular in storage, eliminating padding wastes to improve datapath efficiency. Evaluation suggests the SSD-level (ASSASIN) and processor-level (VarVE) architectures bring 1.9x and 8.9x throughput increase on GeoMean respectively. Further, the two architectures are well compatible with each other. Collectively, they bring 17x multiplicative benefits, enabling in-SSD compute performance to match and exceed the flash array bandwidth.

10.2 Future Work

Our work on ASSASIN and VarVE have laid the foundation for flexible, power-efficient, high-performance computational storage. However, we believe this is just the beginning. Interesting future direction include:

- **Data Center Software Architecture.** The flexibility from general high-performance architectures (ASSASIN and VarVE) enables an ambitious view of system and application partition for computational storage, as in in-SSD compute has the hardware capability to support almost all kernels originally execute on compute CPUs. However, the data center software architecture was not built to support this kind of free movement of function kernels. Interesting questions include 1) how to provide abstraction and naming for both flash data and compute engines across distributed storage engines virtual machines, etc? 2) how to perform performance isolation and accounting for the offloaded kernels from different tenants? 3) how to manage security, e.g. access permissions, for offloads?

- **Expressing Computational Storage Functions.** The streaming structure of ASSASIN and flexibility of VarVE are different from classic memory-oriented computing. While some have proposed programming models based on UNIX file system filters [99] for computational storage, much more flexible parallel and even distributed computation is possible, and perhaps desirable in a cloud storage environment where data are replicated and distributed across devices, nodes, and even data centers. Some interesting questions include 1) how to specify compute on a logical data unit composed of consecutive data pages scattered across storage nodes and devices? 2) how to enable the specification of parallelism and auto-scaling among logic data units across storage nodes and devices?

- **Automating use of Computational Storage.** The classification for projecting offloading benefits based on function properties presents a systematic methodology to determine what functions to offloads would bring high system efficiency and compute speedups. In principle, this classification could be applied inside the compiler and automatically identify offload opportunities through static program analysis. However, multiple research questions remain to be resolved: 1) where should the classification integration in the compiler, and how does the corresponding static analysis span through basic blocks? 2) how to identify data size change property through static analysis? 3) how to identify dataflow directions from abstract syntax tree or IR? 4) how to identify vectorization opportunities?

- **Advanced Computational SSD Architecture.** While ASSASIN and VarVE reflect significant organizational advances for computational SSD architecture to deliver high performance while maintaining flexibility, significant challenges remain in terms architecture support to manage computational SSD Some interesting questions include 1) how to support multi-tenant sharing and security in SSD? 2) how to provide hardware support for performance isolation? 3) what are architecture opportunities for key management or trusted execution?

REFERENCES

[1] Alibaba Cloud Global Infrastructure. https://www.alibabacloud.com/global-locations.

[2] Intel Advanced Vector Extensions Programming Reference. https://www.intel.com/content/dam/develop/external/us/en/documents/36945, .

[3] Intel Advanced Vector Extensions 512. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html, .

[4] SparkSQL TPC-H implementaion with Computational Storage Offload. https://github.com/compstorassasin/compstor.

[5] Amazon EBS: SSD. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html.

[6] Extended Gem5 for ASSASIN evaluations. https://github.com/compstorassasin/gem5.

[7] Extended MQSim for ASSASIN evaluations. https://github.com/compstorassasin/MQSim.

[8] Gem5 Ex5 Core. https://gem5.googlesource.com/public/gem5/+/refs/heads/stable/configs/common/cores/arm/ex5_LITTLE.py.

[9] Google data center locations. https://www.google.com/about/datacenters/locations.

[10] NVMe-OF JBOF. https://nvmexpress.org/wp-content/uploads/NVMe-202-1-Part-1-JBOFs_Final.pdf.

[11] Apache Kafka. https://kafka.apache.org/.

[12] Lookahead carry unit. https://en.wikipedia.org/wiki/Lookahead_carry_unit.

[13] An LZ Codec Designed for SSE Decompression. http://conorstokes.github.io/compression/2016/02/15/an-LZ-codec-designed-for-SSE-decompression.

[14] Microsoft Azure global infrastructure. https://infrastructuremap.microsoft.com/explore.

[15] Neon. https://developer.arm.com/Architectures/Neon.

[16] Changes in nvm express revision 2.0. https://nvmexpress.org/changes-in-nvm-express-revision-2-0/.

[17] AArch64 NZCV Condition Flags. https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Registers/NZCV--Condition-Flags.

[18] Apache ORC. https://orc.apache.org/.

[19] Apache Parquet. https://parquet.apache.org/.

[20] Samsung PM9A3 Data Center SSD. https://www.samsung.com/semiconductor/ssd/pm9a3/.

[21] Protocol Buffers. https://developers.google.com/protocol-buffers.

[22] riscv-v-spec. https://github.com/riscv/riscv-v-spec.

[23] S3 Select. https://aws.amazon.com/blogs/aws/s3-glacier-select/.

[24] Samsung 980 Pro SSD. https://s3.ap-northeast-2.amazonaws.com/global.semi.static/Samsung-NVMe-SSD-980-PRO-Data-Sheet_Rev.2.1.pdf.

[25] Synopsys teaching resources. https://www.synopsys.com/community/university-program/teaching-resources.html.

[26] The Scalable Matrix Extension (SME), for Armv9-A. https://developer.arm.com/documentation/ddi0616/latest.

[27] What is computational storage. https://www.snia.org/education/what-is-computational-storage.

[28] Introducing Apache Spark Data Sources API V2. https://databricks.com/session/apache-spark-data-source-v2, .

[29] Spark Filters. https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/sources/filters.scala, .

[30] Scalable Vector Extension. https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-instruction-emulator/resources/tutorials/sve, .

[31] ARM SVE Encoding. https://developer.arm.com/documentation/ddi0602/2022-06/Index-by-Encoding/SVE-encodings?lang=en, .

[32] Tencent Cloud Global Infrastructure. https://intl.cloud.tencent.com/global-infrastructure, .

[33] NVIDIA Tensor Cores. https://www.nvidia.com/en-us/data-center/tensor-cores/, .

[34] Pruning in Keras example. https://www.tensorflow.org/model_optimization/guide/quantization/training_example.

[35] Quantization aware training in Keras example. https://www.tensorflow.org/model_optimization/guide/quantization/training_example.

[36] RISC-V GNU Compiler Toolchain. `https://github.com/riscv/riscv-gnu-toolchain`.

[37] TPC-H dataset. `http://www.tpc.org/tpch/`.

[38] Spark Tungsten. `https://www.databricks.com/glossary/tungsten`.

[39] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 81–91. ACM, 1998.

[40] Ian F Adams, John Keys, and Michael P Mesnier. Respecting the block interface–computational storage using virtual objects. In *11th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[41] Mohammadamin Ajdari, Pyeongsu Park, Joonsung Kim, Dongup Kwon, and Jangwoo Kim. Cidr: A cost-effective in-line data reduction system for terabit-per-second scale ssd arrays. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 28–41. IEEE, 2019.

[42] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. Breakthrough aes performance with intel aes new instructions. *White paper, June*, page 11, 2010.

[43] Gene M Amdahl, Gerrit A Blaauw, and Frederick P Brooks. Architecture of the ibm system/360. *IBM Journal of Research and Development*, 8(2):87–101, 1964.

[44] Krste Asanovic. *Vector microprocessors.* University of California, Berkeley, 1998.

[45] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2): 1–25, 2017.

[46] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, aug 2011. doi:10.1145/2024716.2024718.

[47] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel {SSD} subsystem. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 359–374, 2017.

[48] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, DL Moal, G Ganger, and George Amvrosiadis. Zns: Avoiding the block interface tax for

flash-based ssds. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*, 2021.

[49] Wendell J Bouknight, Stewart A Denenberg, David E McIntyre, JM Randall, Amed H Sameh, and Daniel L Slotnick. The illiac iv system. *Proceedings of the IEEE*, 60(4): 369–388, 1972.

[50] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[51] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(2):530–543, 2019.

[52] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, page 205–218, 2006.

[53] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. Lightstore: Software-defined network-attached key-value drives. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 939–953. ACM, 2019. doi:10.1145/3297858.3304022.

[54] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, page 251–264, 2012.

[55] Intel Corporation, Micron Technology, Phison Electronics Corporation, Western Digital Corporation, SK Hynix, and Sony Corporation. Open nand flash interface specification revision 4.2. `https://www.onfi.org/specifications`, 2020.

[56] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6):141–142, 2012.

[57] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. Query processing on smart ssds: Opportunities and challenges. In

Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pages 1221–1230, 2013.

[58] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. Query processing on smart ssds: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1221–1230. ACM, 2013.

[59] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. Programmable solid-state storage in future cloud datacenters. *Communications of the ACM*, 62(6):54–62, 2019.

[60] Joao Mario Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. Unlimited vector extension with data streaming support. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 209–222. IEEE, 2021.

[61] Yuanwei Fang, Chen Zou, Aaron J Elmore, and Andrew A Chien. Udp: a programmable accelerator for extract-transform-load workloads and more. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017.

[62] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[63] Marius Grannaes, Magnus Jahre, and Lasse Natvig. Storage efficient hardware prefetching using delta-correlating prediction tables. *Journal of Instruction-Level Parallelism*, 13:1–16, 2011.

[64] Computational Storage Technical Work Group. Computational storage architecture and programming model. *SNIA Standard*, 2022.

[65] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 153–165, 2016.

[66] Saransh Gupta, Justin Morris, Mohsen Imani, Ranganathan Ramkumar, Jeffrey Yu, Aniket Tiwari, Baris Aksanli, and Tajana Šimunić Rosing. Thrifty: Training with hyperdimensional computing across flash hierarchy. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[67] 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. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[68] Daniel Reiter Horn, Ken Elkabany, Chris Lesniewski-Lass, and Keith Winstein. The design, implementation, and deployment of a system to transparently compress hundreds of petabytes of image files for a file-storage service. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 1–15, 2017.

[69] Daniel Reiter Horn, Ken Elkabany, Chris Lesniewski-Lass, and Keith Winstein. The design, implementation, and deployment of a system to transparently compress hundreds of petabytes of image files for a file-storage service. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 1–15, 2017.

[70] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, 2012.

[71] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf`.

[72] R Intel. Architecture instruction set extensions and future features programming reference. `https://www.intel.com/content/dam/develop/external/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf`, 2021.

[73] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: intelligent distributed storage. *Proceedings of the VLDB Endowment*, 10(11):1202–1213, 2017.

[74] Lin Jiang and Zhijia Zhao. Jsonski: Streaming semi-structured data with bit-parallel fast-forwarding. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 200–211, 2022. doi:10.1145/3503222.3507719.

[75] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.

[76] Beau Johnston and Eric McCreath. Parallel huffman decoding: Presenting a fast and scalable algorithm for increasingly multicore devices. In *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, pages 949–958. IEEE, 2017.

[77] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Grafboost: Using accelerated flash storage for external graph analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 411–424. IEEE, 2018. doi:10.1109/ISCA.2018.00042.

[78] Luyi Kang, Yuqi Xue, Weiwei Jia, Xiaohao Wang, Jongryool Kim, Changhwan Youn, Myeong Joon Kang, Hyung Jin Lim, Bruce Jacob, and Jian Huang. Iceclave: A trusted execution environment for in-storage computing. In *Proceedings of the 54th International Symposium on Microarchitecture*. ACM, 2021.

[79] Seongyoung Kang, Jiyoung An, Jinpyo Kim, and Sang-Woo Jun. Mithrilog: Near-storage accelerator for high-performance log analytics. In *Proceedings of the 54th International Symposium on Microarchitecture*. ACM, 2021.

[80] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel DG Lee. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 144–154. ACM, 2019.

[81] Jonghwa Kim, Choonghyun Lee, Sangyup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu-ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. Deduplication in ssds: Model and quantitative analysis. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2012.

[82] Gunjae Koo, Kiran Kumar Matam, Te I, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 219–231. ACM, 2017.

[83] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[84] Geoff Langdale and Daniel Lemire. Parsing gigabytes of json per second. *The VLDB Journal*, 28(6):941–960, 2019.

[85] 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.

[86] Jeff LeFevre and Noah Watkins. Skyhook: Programmable storage for databases. *Vault*, February 2019.

[87] Cangyuan Li, Ying Wang, Cheng Liu, Shengwen Liang, Huawei Li, and Xiaowei Li. {GLIST}: Towards {In-Storage} graph learning. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 225–238, 2021.

[88] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. Leapio: Efficient and portable virtual nvme storage on arm socs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 591–605, 2020.

[89] Sean Lie. Wafer-scale deep learning. *HotChip*, 2019.

[90] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia de Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mwei Hwu. Deepstore: In-storage acceleration for intelligent queries. In *Proceedings of the 52th International Symposium on Microarchitecture*. ACM, 2019.

[91] 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 '22, page 635–654, 2022. doi:10.1145/3503222.3507702.

[92] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. Graphssd: graph semantics aware ssd. In *Proceedings of the 46th international symposium on computer architecture*, pages 116–128, 2019.

[93] Patterson, David. SIMD Instructions Considered Harmful. `https://www.sigarch.org/simd-instructions-considered-harmful`.

[94] Alex Peleg and Uri Weiser. Mmx technology extension to the intel architecture. *IEEE micro*, 16(4):42–50, 1996.

[95] Srinivas K Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming simd extensions on the pentium iii processor. *IEEE micro*, 20(4):47–57, 2000.

[96] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. Sparsecore: stream isa and processor specialization for sparse computation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 186–199, 2022.

[97] Enrico Reggiani, Cristóbal Ramírez Lazo, Roger Figueras Bagué, Adrián Cristal, Mauro Olivieri, and Osman Sabri Unsal. Bison-e: a lightweight and high-performance accelerator for narrow integer linear algebra computing on the edge. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 56–69, 2022.

[98] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*, pages 62–73. ACM, 1998.

[99] Zhenyuan Ruan, Tong He, and Jason Cong. Insider: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, 2019.

[100] Richard M Russell. The cray-1 computer system. *Communications of the ACM*, 21 (1):63–72, 1978.

[101] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8. IEEE, 2017. URL `https://github.com/lowRISC/ibex`.

[102] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. Accessible near-storage computing with fpgas. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–12, 2020.

[103] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, 2014.

[104] James E Smith, Greg Faanes, and Rabin Sugumar. Vector instruction set support for conditional operations. *ACM SIGARCH Computer Architecture News*, 28(2):260–269, 2000.

[105] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The arm scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017. doi:10.1109/MM.2017.35.

[106] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor Mudge, and Ronald Dreslinski. Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 654–667. IEEE, 2021. doi:10.1109/HPCA51647.2021.00061.

[107] 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, 2018.

[108] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 119–132, 2013.

[109] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052. ACM, 2017.

[110] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 116, 2011.

[111] Martin H Weik. The eniac story. *Ordnance*, 45(244):571–575, 1961.

[112] André Weißenberger and Bertil Schmidt. Massively parallel huffman decoding on gpus. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–10, 2018.

[113] 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, 2014.

[114] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, 2012.

[115] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27 (11):2629–2640, 2019.

[116] Yunan Zhang, Po-An Tsai, and Hung-Wei Tseng. Simd2: a generalized matrix instruction set for accelerating tensor computation beyond gemm. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 552–566, 2022.

[117] Chen Zou and Andrew A Chien. Assasin: Architecture support for stream computing to accelerate computational storage. In *2022 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022.

[118] Chen Zou, Hui Zhang, Andrew A Chien, and Yang-Seok Ki. Psacs: Highly-parallel shuffle accelerator on computational storage. In *ICCD*, pages 480–487. IEEE, 2021.