Design for Computational Storage Simulation Platform

Peter Wilcox

University of California, Santa Cruz
Baskin School of Engineering
Santa Cruz, CA, USA
pcwilcox@ucsc.edu

Heiner Litz

University of California, Santa Cruz
Baskin School of Engineering
Santa Cruz, CA, USA
hlitz@ucsc.edu

Abstract

Data movement between storage and compute resources represents a bottleneck in data-driven applications. This performance bottleneck can be mitigated by leveraging inherent parallelism in the user application and offloading component tasks, called compute kernels, for execution at the storage layer instead of on the host CPU. Performing computation on the storage device instead of moving the data through the memory hierarchy to the CPU cache provides opportunities for increased throughput and reduced energy consumption by exploiting data independence present in the application. Previous work in this domain requires either specialized hardware for each type of application, a restrictive set of operations executable on the storage device, or significant engineering effort to program the compute kernel offloaded to the device. While prior work has proposed accelerating certain compute tasks (e.g. encryption/decryption, compression/decompression) the application space has been very limited, because applications need to fulfill certain properties to be a good candidate. Currently there exists no way to figure this out. To extend the applicability of computational storage we propose a new simulator platform that enables large design space explorations for storage accelerators and applications. By reducing the cost of offload we can leverage more fine-grained compute kernels, increasing the applicability of computational storage. Researchers can use our simulator design to explore different approaches for accelerating applications, including different mechanisms for offloading compute tasks to the device and consuming results, different methods of interfacing between user applications and the device, and different device characteristics and capabilities. This system enables us to evaluate the scalability of compute offloading techniques without significant investment in hardware or engineering effort.

CCS Concepts: • **Hardware** → *Communication hardware, interfaces and storage*; **Hardware accelerators**.

This work is licensed under a Creative Commons Attribution International 4.0 License

CHEOPS '21, April 26, 2021, Online, United Kingdom
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8302-8/21/04.
<https://doi.org/10.1145/3439839.3459085>

Keywords: Computational Storage, FPGA, SSD, NVMe, Near-Data Processing, Device Simulation, Accelerator, Compute Kernel

ACM Reference Format:

Peter Wilcox and Heiner Litz. 2021. Design for Computational Storage Simulation Platform. In *Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '21)*, April 26, 2021, Online, United Kingdom. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3439839.3459085>

1 Introduction

In data-driven applications, the energy cost to move data to and from DRAM and persistent storage has come to dominate the energy cost of performing computation on that data. Compared to the cost of arithmetic operations, reading from DRAM and NVMe increases the energy cost of a compute task by as many as 6 orders of magnitude [14, 16, 17, 22, 23, 25, 31, 47, 48, 53]. Furthermore, applications with memory access patterns that lead to high cache miss rates can also spend an inordinate amount of execution time fetching data from DRAM, creating additional performance bottlenecks. These costs and slowdowns can be mitigated or eliminated by instead moving the compute task to the storage device. A variety of proposals have been made to reduce this overhead, such as the use of specialized domain-specific accelerators [40, 52], or performing online clustering of the data to reduce movement cost using specialized hardware [53]. Unfortunately, domain-specific devices have the drawback that they are generally not usable with other applications, requiring data center operators to invest in new hardware for each application.

These increasing overheads demand that we take advantage of inherent parallelism and data independence in the application by offloading compute tasks for concurrent execution on the storage device [14, 36, 45]. Offloading compute tasks to the storage layer will further allow us to leverage potential hardware benefits of computational storage, such as the significantly higher bandwidth connection existing on the device, and potentially integrating accelerator resources with existing tasks such as storage management. A broadly applicable device, however, must be able to provide generalized compute acceleration to the user application.

This paper presents a novel design for a computational storage simulation platform, allowing researchers to identify properties and characteristics of applications which make a

suitable fit for acceleration using computational storage. Researchers can further explore software interfaces, synchronization mechanisms, and program modifications in order to best utilize device capabilities. Additionally, hardware engineers can use our simulation platform to evaluate the impact that different device capabilities may have on application performance and overhead. Finally, developers using our platform can evaluate the methods by which new applications may best take advantage of computational storage devices, significantly reducing the engineering effort required to realize the benefits of computational storage devices.

2 Background and Related Work

Computational Storage. A computational storage device allows the host CPU to offload data-intensive compute tasks to the device where the data is stored. Historically these devices were implemented using HDDs (see [8, 30, 43]), and more recently this type of design has been implemented with SSDs (see [3, 14, 29, 48]). For simplicity, in the rest of this paper we use *smart SSD* to refer to computational storage SSDs.

Early work on smart SSDs proposed executing compute kernels using onboard compute resources already present on commercial SSDs [29, 51]. Later designs improved on this approach by adding additional compute capabilities (in the form of FPGA, ARM CPU, or GPU), however these designs still provided only a limited set of compute operations to the user application, such as sorting [35], database scans and joins [32], query processing [19], and similar [10, 13, 18, 27, 28, 41]. More recent smart SSD designs have implemented compute kernel offload using abstraction frameworks such as a filesystem [9, 20, 44–46]. To further improve the applicability of computational storage, remote storage disaggregation techniques have been proposed [21, 24, 34, 55]. On-chip computational capabilities have also been leveraged to improve the performance of storage devices, for instance, by executing machine learning-based prefetching mechanisms [12] on SSDs. Despite these improvements, these designs still have issues with programmability or generalization, because it requires significant engineering effort to port existing applications onto these platforms. Other recent designs have attempted to address this by adding general-purpose CPUs to host full Linux instances on the SSD [49, 50]. While this approach has resolved some of the programmability drawbacks, it comes at the cost of increased complexity in the hardware.

Our work differs from each of these systems in that, as a simulator platform, it is relatively easy for the end user to modify and extend the capabilities of the simulated device and its interface library in order to fit the needs of their application. The computational storage library included in our simulator platform enables the user to load custom compute kernels for execution on the simulated smart SSD (see § 3.3). This means that, in addition to block I/O and a filesystem abstraction, researchers or application developers can rapidly iterate

on existing software interfaces and compute kernel offload mechanisms. This significantly reduces the engineering effort required to realize the full potential of smart SSDs.

Compute Kernel. A *compute kernel* (not to be confused with an operating system kernel) is a small section of code which is both I/O bound and comprises a significant portion of the overall execution time of the application. We observe that a kernel should be small so that it can be executed efficiently on low-power compute resources available on the smart SSD. Furthermore, if it is not I/O bound then it stands to benefit little from offloading to the storage layer. Previous systems utilizing compute kernels include domain-specific accelerators [52], GPUs [4], and computational storage devices [15].

Our simulator platform differs from previous work in that we enable exploring the design space of compute kernels to include additional types of compute functions. For example, GPUs are specialized for highly parallel computation, and therefore the compute kernels offloaded to these devices are designed to be run many times in parallel by separate threads on the device [4]. Users are able to configure the kernel to be offloaded to our simulator, which greatly expands the set of space of applications which can be accelerated, for example compute kernels with data dependencies. Applications which previously required specialized hardware accelerators (such as Minimap2 [37], which is accelerated using Darwin [52]) can be accelerated using our design by carefully extracting the compute kernel from the algorithm. Additionally, users are able to modify how the application (running on the host CPU) interacts with the smart SSD interface using the CS library API (§ 3.3). This makes it possible to accelerate new types of applications which were not previously suitable for smart SSDs.

System Simulator. A *system simulator* is a platform which allows a researcher to model the behavior of computer systems in software. Examples of this type of tool include full system simulators which model entire processors and are typically able to support unmodified operating systems (such as MARSS [42] or gem5 [11]), as well as those which simulate specific components (such as Pantheon [54], Flashsim [33], or Vivado Simulator [6]). Previous works in this area have simulated storage devices [2, 33] or FPGA devices [6], however, to the best of the authors' knowledge, no prior work has provided a simulated smart SSD. In contrast to the previous work, our design allows the researcher to model the impact of different hardware properties in both the storage device as well as the attached accelerator.

3 Computational Storage Simulator

Here we present our design for a computational storage simulator platform, which simulates an SSD with onboard FPGA accelerator accessible using NVMe. We begin with an overview

Figure 1. Architecture of the computational storage simulator. The simulated device is made available to the guest VM using the virtual device interface.

of the system architecture and device emulation in our design (§3.1), then explore the storage stack (§3.2) and the user application interface (§3.3).

3.1 System Architecture

Our simulation platform is designed to be run on a VM host (which we call the *server* in this paper, as the term *host* is overloaded). The server runs an instance of the QEMU system emulator, which can emulate hardware devices and make them available to a guest VM [2]. As seen in Fig. 1, we emulate an NVMe controller, onboard accelerator, and onboard storage using QEMU running on the server. These components are exposed to the guest as a simulated smart SSD accessible via NVMe. As shown in Fig. 2, user applications running on the guest are able to interface with the simulated smart SSD using the provided CS library, which exposes a C API for block I/O and compute offload control. We also propose a POSIX-compliant file system using the FUSE kernel module which allows users to use traditional filesystem methods.

QEMU provides the capability to fully emulate NVMe hardware devices in software on the server by implementing an NVMe controller which receives and processes NVMe admin and I/O commands. QEMU assumes the existence of a flash translation layer (FTL) and refers to data using logical block addresses (LBAs) directly. NVMe submission queues are polled as part of QEMU’s main event loop, while NVMe I/O commands are scheduled using coroutines. NVMe command processing is detailed in Fig. 3.

We have modified the QEMU NVMe I/O stack by exposing additional command capabilities in the device. These capabilities are implemented using a *filter* block driver, which is inserted just above the storage device in the QEMU storage architecture. The QEMU I/O stack routes requests through

Figure 2. Architecture of the simulated platform, as seen by the guest VM.

this filter layer prior to delivering them to the drive, which allows our filter module to examine each request in order to determine if it should be handled as an accelerator command. This gives us a great deal of flexibility when implementing additional commands. These additional commands allow the user to specify compute kernels which can be executed by the simulated accelerator on the SSD. This modification is inserted in the coroutine function of the QEMU command processing algorithm Fig. 3. Additional kernels can be loaded using the CS library API (§3.3) or by implementing additional functions with corresponding opcodes in this section of the simulator code.

3.2 Storage Stack

Instead of using the traditional Linux storage stack, we implement the accelerator interface with a custom Computational Storage (CS) library (see Fig. 2). This library exposes a C API to the user (see §3.3), which wraps the NVMe block I/O interface and adds additional commands for handling compute kernel offload. Instead of the native Linux NVMe driver and interface, we use the xNVMe library, which provides a simple storage management API and is compatible with multiple storage backend drivers [7, 38, 39]. In our system we have implemented this with the SPDK driver, which binds the NVMe device entirely in user space and bypasses the kernel [1]. When the device is bound using this system, special regions of memory are allocated and configured for direct memory access (DMA) reads and writes from the device. Data structures needed to support NVMe I/O, such as submission and completion queues, are placed in this region. This allows the I/O pathway to bypass the kernel and provide a very high throughput connection to the device.

```

coroutine(cmd):
    if cmd->type == CS_CMD; then
        accel(cmd)
    else
        io(cmd)
    enqueue(cmd->cqe)

execute(cmd):
    if cmd->sq_id == 0; then
        admin(cmd)    // Handled synchronously
        enqueue(cmd->cqe)
    else
        cmd->status = NO_COMPLETE
        coroutine(cmd) // Add I/O to work queue
    return

// Main event loop
while sq not empty; do
    read(sq)          // Uses DMA to guest
    execute(sq->command)
foreach cqe; do
    write(cqe)        // Uses DMA to guest

```

Figure 3. NVMe I/O processing in the QEMU NVMe controller [26]. The main event loop polls the submission queue and examines each command as it is read from the guest. In the case of I/O commands, execution is scheduled to be performed using asynchronous coroutines, while admin commands are executed immediately. Upon command completion, the corresponding completion queue entries are posted and written to the guest using DMA.

3.3 Computational Storage Library

We implement a computational storage library which makes block I/O and compute kernel offload functions available to the user application in a simple C API, shown in Table 1. This reduces the barrier to entry for using our system, increasing programmability and making it easier for application developers to use it as a basis for modifying their applications to run on computational storage devices or to develop new ones. This API makes use of several opaque data structures, `cs_pointer`, `cs_device`, and `cs_request`. The internal fields of these data structures are used by the CS library functions to manage the NVMe interface to the device, and they allow the user application to reference devices, requests, and data on the storage device in a straightforward manner. In particular, we note that the pointer struct allows the user application to refer to data without needing to reference its corresponding LBA on the device, as this translation is performed internally by the library.

Table 1. The CS library exposes a simple API for user applications to control block I/O and compute kernel offloads.

Function	Description
<code>cs_open</code>	Open device, allocate memory for DMA.
<code>cs_close</code>	Close device, free allocated memory.
<code>cs_write</code>	Asynchronous block write.
<code>cs_read</code>	Asynchronous block read.
<code>cs_get_pointer</code>	Convert (LBA, offset) pairs into pointer of the specified data type.
<code>cs_get_lba</code>	Get (LBA, offset) pairs corresponding to a particular filename.
<code>cs_load_kernel</code>	Load a kernel function for execution on the simulated device.
<code>cs_exec</code>	Asynchronously execute a kernel.
<code>cs_status</code>	Check request status.

3.4 Computational Storage Filesystem

We propose utilizing the CS library to implement a POSIX-compliant filesystem layer using FUSE [5]. Users can mount this filesystem with the FUSE kernel module and make the storage device accessible to unmodified Linux tools and syscalls. Filesystem metadata (such as file and directory maps) are stored on the device in a region controlled by the CS library, to be accessed when performing operations based on the filesystem, such as translations between filesystem semantics and logical block addresses. Performing these metadata translations before issuing requests using the CS library cleanly separates filesystem operations used by the guest from block I/O operations used by the smart SSD. This allows us to simplify accelerator operations because it will not need to perform these translations on the device, but rather can communicate to the NVMe controller using LBA semantics.

Because file-based I/O to the device must pass through the FUSE kernel module in addition to the CS library interface, this access interface won't have the same performance characteristics as the block I/O and compute kernel offload interface provided by the library API. We note, however, that this design enables a user to write data to the device using traditional Linux tools, and therefore they need only modify the algorithms used in their application in order to take advantage of the higher throughput provided by the library API.

4 Compute Kernel Offload

Here we give an example of the compute kernel offload procedure using our simulator platform. We chose a simple string search procedure as the kernel to offload (see Fig. 5). Each of the numbered steps in the following description corresponds with the labels in Fig. 4. For brevity we assume that the appropriate data has already been stored to the device in a known location.

Figure 4. Compute kernel offload example, showing a substring search offload to the CS device.

1. The user application opens the CS device using the CS library.
2. The CS library sets up the NVMe data structures and two-way polling between the host and the device.
3. The user application issues a write request containing the match parameter to the drive, using the CS library API (see §3.3).
4. The write request passes through the storage stack, and when the request is complete, the completion queue entry is written back to the host (as described in §3.2).
5. With the previous write request complete, the guest issues a write request containing the control data structure to the corresponding LBA.
6. The write request passes through the storage stack to the emulated NVMe controller.
7. The controller examines the request and determines it is a write to the corresponding accelerator control address. Instead of performing a write, it initiates the specified compute kernel function (as described in Fig. 3).
8. The compute kernel function reads the input parameters from the control structure, then reads in the corresponding data from storage into buffers in memory. Once read, the kernel function executes the string search (as shown in Fig. 5).
9. Once the string search is complete, the kernel function writes the result to the specified result LBA.
10. With the result persisted to storage, the kernel function frees allocated memory and marks the initial write request as complete.
11. Once the write request completion queue entry has passed through the stack to the host, the host issues a read request to the specified results LBA.
12. The read request is handled at the device, and the results are written back to the client.

```
string_search(cmd):
    data = read(cmd->d_off, cmd->d_len)
    match = read(cmd->m_off, cmd->m_len)
    p, result = 0
    while data[p] is not null:
        while data[p] is not match[0]
          and data[p] is not null:
            p++
        if data[p] is null:
            break
        equal = true
        for i from 0 to cmd->m_len:
            if data[p] is not match[i]:
                equal = false
                break
        p++
        if equal is true:
            result++
    write(result, cmd->r_off)
    cmd->status = complete
```

Figure 5. A simple string search kernel which examines a plaintext string and returns the number of substrings which match the query string.

5 Evaluation

In this section we present some preliminary evaluations of our simulator design. Our evaluations were run on a single server/guest setup and compare host vs device kernel executions.

5.1 Experiment Setup

All experiments were performed on a test server equipped with an Intel Xeon 5218 CPU running at 2.30GHz, 376GB DRAM, and a 2TB Samsung 860 SSD. Our prototype simulator is based on QEMU version 5.2.0 and xNVMe version v0.0.22. The guest VM image is setup with host CPU mode, SMP set to 4, 64GB memory, intel-iommu and intremap set to on, and a 32GB qcow2-formatted SCSI OS drive. The guest and the server both run Ubuntu 20.04. Our prototype CS device is implemented as a filter node on top of an NVMe file-backed device node with 4KB logical/physical blocks.

5.2 Experiment Procedures

For these evaluations we ran two experiments in order to compare execution time of a simple kernel (see Fig. 4 and Fig. 5) when it is run on the host vs. when it is offloaded to the device. We started by preloading a 100MB plaintext dataset onto the device. We then repeated each test procedure over the range of dataset sizes (in terms of logical blocks, i.e. 4KB chunks) possible with this dataset.

Executing on host. In order to model the type of behavior exhibited by the applications we are targeting with our system,

we began the host procedure by reading the dataset into DRAM using userspace NVM commands similar to those in the library API. Each sequence of dataset reads was configured to use the minimum number of requests by reading the largest number of logical blocks possible (on our system, this was 128 blocks). Once the data was read into DRAM, we executed the compute kernel and recorded the results. In this procedure, timestamps were taken before and after the read step and the compute step, as well as before and after the total procedure.

Executing on device. In order to simulate the target application usage of this type of device, we began the procedure by writing the match string to the device, followed by the accelerator command struct. We then waited for the kernel to complete and recorded the results. In this procedure, timestamps were taken before and after each run.

5.3 Results

We executed the above procedure using a substring search kernel on a plaintext dataset with samples from 1 to 20,000 4KB blocks, which we chose because this is the smallest data size readable with this device configuration. The results are shown in Fig. 6. In particular we note that while I/O does take a proportion of time for this compute kernel, it does not dominate the execution time. This is evidenced by the difference between the lines signifying the host read time and host total execution time. While read time does increase as the dataset size increases, it is not as significant as the compute time needed on either the host or the device. The primary benefit that this kernel offload obtains, then, is in data movement reductions. We note that this kernel requires only two NVMe requests to offload to the device, the query string and the control structure itself. Assuming that each of these is at most one logical block in length (a reasonable assumption with this sort of query), we observe that the kernel offload method sends and receives only 3 blocks during its execution: the query string, the control structure, and the result. Compared to the host execution method this can be a significant reduction in data movement when considered at scale, as the host must read the entire dataset into DRAM in order to perform this kernel.

6 Conclusion

In this paper we have presented a novel design for a computational storage simulator platform. This platform provides researchers with the tools needed to explore the limits and scalability of compute kernel offload, by experimenting with different device interface methods, device capabilities, and mechanisms for synchronizing and consuming the results of kernel offload. Furthermore, our platform gives software developers an easy-to-use and accessible framework for modifying existing applications, or designing new ones, to be accelerated using computational storage devices. This framework will reduce the engineering effort required to produce these

Execution Time: Host vs. Device

Execution Time (s) vs Data Size (4KB blocks)

Figure 6. Device execution, host read, and host total execution times as a function of dataset size.

applications, making it easier to adopt these types of devices in production environments.

7 Acknowledgements

This work has been supported by Western Digital as well as by NSF grants CCF-1942754 and CNS-1841545.

References

- [1] 2019. Storage Performance Development Kit. <https://spdk.io/>
- [2] 2020. QEMU. <https://www.qemu.org/>
- [3] 2020. Samsung SmartSSD Computational Storage Drive. <https://samsungsemiconductor-us.com/smartssd/index.html>
- [4] 2021. CUDA C++ Programming Guide. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- [5] 2021. libfuse. <https://github.com/libfuse/libfuse>
- [6] 2021. Vivado Simulator. <https://www.xilinx.com/>
- [7] 2021. xnvme.io. <https://xnvm.io/>
- [8] Anurag Acharya, Mustafa Uysal, and Joel Saltz. 1998. Active disks: programming model, algorithms and evaluation. *ACM SIGOPS Operating Systems Review* 32, 5 (Oct. 1998), 81–91. <https://doi.org/10.1145/384265.291026>
- [9] Ian F. Adams, John Keys, and Michael P. Mesnier. 2019. Respecting the block interface - computational storage using virtual objects. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. USENIX Association, Renton, WA. <https://www.usenix.org/conference/hotstorage19/presentation/adams>
- [10] Duck-Ho Bae, Jin-Hyung Kim, Sang-Wook Kim, Hyunok Oh, and Chanik Park. 2013. Intelligent SSD: a turbo for big data mining. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management (CIKM '13)*. Association for Computing Machinery, New York, NY, USA, 1573–1576. <https://doi.org/10.1145/2505515.2507847>
- [11] 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. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. <https://doi.org/10.1145/2024716.2024718>
- [12] Chandranil Chakrabortii and Heiner Litz. 2020. Learning I/O Access Patterns to Improve Prefetching in SSDs. In *Proceedings of The European Conference on Machine Learning (ECML-PKDD)*.

- [13] Benjamin Y Cho, Won Seob Jeong, Doohwan Oh, and Won Woo Ro. 2013. XSD: Accelerating MapReduce by Harnessing the GPU inside an SSD. (2013), 6.
- [14] I. Stephen Choi and Yang-Suk Kee. 2015. Energy Efficient Scale-In Clusters with In-Storage Processing for Big-Data Analytics. In *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS '15)*. Association for Computing Machinery, Washington DC, DC, USA, 265–273. <https://doi.org/10.1145/2818950.2818983>
- [15] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. ACM, New York, NY, USA, 924–939. <https://doi.org/10.1145/3352460.3358276>
- [16] William J. Dally. 2019. The Future of Computing: Domain-Specific Accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Columbus, OH, USA. https://www.microarch.org/micro52/media/dally_keynote.pdf
- [17] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63, 7 (June 2020), 48–57. <https://doi.org/10.1145/3361682>
- [18] Arup De, Maya Gokhale, Rajesh Gupta, and Steven Swanson. 2013. Minerva: Accelerating Data Analysis in Next-Generation SSDs. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. 9–16. <https://doi.org/10.1109/FCCM.2013.46>
- [19] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query processing on smart SSDs: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1221–1230. <https://doi.org/10.1145/2463676.2465295>
- [20] 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. 2016. Biscuit: a framework for near-data processing of big data workloads. In *ACM SIGARCH Computer Architecture News*, Vol. 44. 153–165. <https://doi.org/10.1145/3007787.3001154>
- [21] Zvika Guz, Harry Li, Anahita Shayesteh, and Vijay Balakrishnan. 2017. NVMe-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference*. 1–9.
- [22] Bryan Harris and Nihat Altıparmak. 2020. Ultra-Low Latency SSDs' Impact on Overall Energy Efficiency. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association. <https://www.usenix.org/conference/hotstorage20/presentation/harris>
- [23] Mark Horowitz. 2014. Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 10–14. <https://doi.org/10.1109/ISSCC.2014.6757323> ISSN: 0193-6530, 2376-8606.
- [24] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. {TCP}{RDMA}: CPU-efficient Remote Storage Access with i10. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 127–140.
- [25] Siddhartha Jana, Oscar Hernandez, Stephen Poole, and Barbara Chapman. 2014. Power Consumption Due to Data Movement in Distributed Programming Models. In *Euro-Par 2014 Parallel Processing (Lecture Notes in Computer Science)*, Fernando Silva, Inês Dutra, and Vítor Santos Costa (Eds.). Springer International Publishing, Cham, 366–378. https://doi.org/10.1007/978-3-319-09873-9_31
- [26] Klaus Jensen. 2020. Reviving The QEMU NVMe Device (from Zero to ZNS). In *Storage Developers Conference 2020*. SNIA, Santa Clara CA USA. <https://www.snia.org/educational-library/reviving-qemu-nvme-device-zero-zns-2020>
- [27] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment* 9, 12 (Aug. 2016), 924–935. <https://doi.org/10.14778/2994509.2994512>
- [28] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An appliance for Big Data analytics. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 1–13. <https://doi.org/10.1145/2749469.2750412> ISSN: 1063-6897.
- [29] Yangwook Kang, Yang-suk Kee, Ethan L. Miller, and Chanik Park. 2013. Enabling cost-effective data processing with smart SSD. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–12. <https://doi.org/10.1109/MSST.2013.6558444> ISSN: 2160-1968.
- [30] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. 1998. A case for intelligent disks (IDISks). *ACM SIGMOD Record* 27, 3 (Sept. 1998), 42–52. <https://doi.org/10.1145/290593.290602>
- [31] Gokcen Kestor, Roberto Gioiosa, Darren J. Kerbyson, and Adolfo Hoisie. 2013. Quantifying the energy cost of data movement in scientific applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 56–65. <https://doi.org/10.1109/IISWC.2013.6704670>
- [32] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. 2016. In-storage processing of database scans and joins. *Information Sciences* 327 (Jan. 2016), 183–200. <https://doi.org/10.1016/j.ins.2015.07.056>
- [33] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar. 2009. FlashSim: A Simulator for NAND Flash-Based Solid-State Drives. In *2009 First International Conference on Advances in System Simulation*. 125–131. <https://doi.org/10.1109/SIMUL.2009.17>
- [34] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash == Local Flash. In *Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*.
- [35] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. 2020. SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD. *IEEE Computer Architecture Letters* 19, 2 (July 2020), 110–113. <https://doi.org/10.1109/LCA.2020.3009347> Conference Name: IEEE Computer Architecture Letters.
- [36] Young-Sik Lee, Luis Cavazos Quero, Youngjae Lee, Jin-Soo Kim, and Seungryoul Maeng. 2014. Accelerating External Sorting via On-the-fly Data Merge in Active SSDs. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*. USENIX Association, Philadelphia, PA. <https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/lee>
- [37] Heng Li. 2018. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics* 34, 18 (2018), 3094–3100. <https://doi.org/10.1093/bioinformatics/bty191>
- [38] Simon Lund and Klaus Jensen. 2020. Morning Tutorial 2: An introduction to NVMe Zoned Namespaces. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA.
- [39] Simon F. A. Lund. 2019. Programming Emerging Storage Interfaces. In *Storage Developers Conference 2019*. SNIA, Santa Clara, California. https://www.snia.org/sites/default/files/SDC/2019/presentations/NVMe/Lund_Simon_Programming_Emerging_Storage_Interfaces.pdf
- [40] Anirban Nag, C. N. Ramachandra, Rajeev Balasubramonian, Ryan Stutsman, Edouard Giacomin, Hari Kambalasubramanyam, and Pierre-Emmanuel Gaillardon. 2019. GenCache: Leveraging In-Cache Operators for Efficient Sequence Alignment. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. ACM, New York, NY, USA, 334–346. <https://doi.org/10.1145/3352460.3358308> event-place: Columbus, OH, USA.

- [41] Jian Ouyang, Shiding Lin, Zhenyu Hou, Peng Wang, Yong Wang, and Guangyu Sun. 2013. Active SSD design for energy-efficiency improvement of web-scale data analysis. In *International Symposium on Low Power Electronics and Design (ISLPED)*. 286–291. <https://doi.org/10.1109/ISLPED.2013.6629310>
- [42] A. Patel, F. Afram, S. Chen, and K. Ghose. 2011. MARSS: A full system simulator for multicore x86 CPUs. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1050–1055.
- [43] Erik Reidel, Garth Gibson, and Christos Faloutsos. 1999. Active Storage For Large-Scale Data Mining and Multimedia Applications. (Sept. 1999). <https://doi.org/10.1184/R1/6602978.v1>
- [44] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. 379–394. <https://www.usenix.org/conference/atc19/presentation/ruan>
- [45] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. 2020. Accessible near-storage computing with FPGAs. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. <https://doi.org/10.1145/3342195.3387557>
- [46] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 67–80. <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/seshadri>
- [47] Yanwei Song and Engin Ipek. 2015. More is less: improving the energy efficiency of data movement via opportunistic use of sparse codes. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 242–254. <https://doi.org/10.1145/2830772.2830806>
- [48] Devesh Tiwari, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Simona Boboila, and Peter J. Desnoyers. 2012. Reducing Data Movement Costs Using Energy-Efficient, Active Computation on SSD. In *2012 Workshop on Power-Aware Computing and Systems (HotPower 12)*. USENIX Association, Hollywood, CA. <https://www.usenix.org/conference/hotpower12/workshop-program/presentation/Tiwari>
- [49] Mahdi Torabzadehkashi, Siavash Rezaei, Vladimir Alves, and Nader Bagherzadeh. 2018. CompStor: An In-storage Computation Platform for Scalable Distributed Processing. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1260–1267. <https://doi.org/10.1109/IPDPSW.2018.00195>
- [50] Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. 2019. Catalina: In-Storage Processing Acceleration for Scalable Big Data Analytics. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 430–437. <https://doi.org/10.1109/EMPDP.2019.8671589> ISSN: 2377-5750.
- [51] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2016. Morpheus: creating application objects efficiently for heterogeneous computing. *ACM SIGARCH Computer Architecture News* 44, 3 (June 2016), 53–65. <https://doi.org/10.1145/3007787.3001143>
- [52] Yatish Turakhia, Gill Bejerano, and William J. Dally. 2018. Darwin: A Genomics Co-processor Provides up to 15,000X Acceleration on Long Read Assembly. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '18*. ACM Press, Williamsburg, VA, USA, 199–213. <https://doi.org/10.1145/3173162.3173193>
- [53] Shibo Wang and Engin Ipek. 2016. Reducing data movement energy via online data clustering and encoding. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. <https://doi.org/10.1109/MICRO.2016.7783735>
- [54] John Wilkes. 1995. The Pantheon storage-system simulator.
- [55] Minghao Xie and Chen Qian. 2020. ReFlex4ARM: Supporting 100GbE Flash Storage Disaggregation on ARM SoC. In *OCP Future Technology Symposium*.