Cost-effective, Energy-efficient, and Scalable Storage Computing for Large-scale AI Applications

JAEYOUNG DO, Microsoft Research, USA
VICTOR C. FERREIRA, Federal University of Rio de Janeiro, Brazil
HOSSEIN BOBARSHAD and MAHDI TORABZADEHKASHI, NGD Systems, USA
SIAVASH REZAEI and ALI HEYDARIGORJI, University of California, Irvine, USA
DIEGO SOUZA, Wespa Intelligent Systems
BRUNNO F. GOLDSTEIN and LEANDRO SANTIAGO, Federal University of Rio de Janeiro
MIN SOO KIM, University of California, Irvine, USA
PRISCILA M. V. LIMA and FELIPE M. G. FRANÇA, Federal University of Rio de Janeiro, Brazil
VLADIMIR ALVES, NGD Systems, USA

The growing volume of data produced continuously in the Cloud and at the Edge poses significant challenges for large-scale AI applications to extract and learn useful information from the data in a timely and efficient way. The goal of this article is to explore the use of computational storage to address such challenges by distributed near-data processing. We describe Newport, a high-performance and energy-efficient computational storage developed for realizing the full potential of in-storage processing. To the best of our knowledge, Newport is the first commodity SSD that can be configured to run a server-like operating system, greatly minimizing the effort for creating and maintaining applications running inside the storage. We analyze the benefits of using Newport by running complex AI applications such as image similarity search and object tracking on a large visual dataset. The results demonstrate that data-intensive AI workloads can be efficiently parallelized and offloaded, even to a small set of Newport drives with significant performance gains and energy savings. In addition, we introduce a comprehensive taxonomy of existing computational storage solutions together with a realistic cost analysis for high-volume production, giving a good big picture of the economic feasibility of the computational storage technology.

CCS Concepts: • **Information systems** → **Storage architectures**; • **Computer systems organization** → *Distributed architectures*; • **Computing methodologies** → *Artificial intelligence*;

Additional Key Words and Phrases: Computational storage, in-storage processing, solid-state drive, similarity search, neural network, object tracking

This work is supported in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. Also, this material is based upon work supported by the National Science Foundation under Grant No. 1660071.

Authors' addresses: J. Do (Corresponding author), Microsoft Research, Redmond, Washington; email: jaedo@microsoft.com; V. C. Ferreira, B. F. Goldstein, L. Santiago, P. M. V. Lima, and F. M. G. França, Federal University of Rio de Janeiro, Rio de Janeiro, Brazil; H. Bobarshad and M. Torabzadehkashi, NGD Systems, Irvine, California; S. Rezaei, A. HeydariGorji, and M. S. Kim, University of California, Irvine, Irvine, California; D. Souza, Wespa Intelligent Systems, Rio de Janeiro, Brazil; V. Alves, NGD Systems, Irvine, California.

1553-3077/2020/10-ART21 $15.00
https://doi.org/10.1145/3415580

ACM Reference format:
Jaeyoung Do, Victor C. Ferreira, Hossein Bobarshad, Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Diego Souza, Brunno F. Goldstein, Leandro Santiago, Min Soo Kim, Priscila M. V. Lima, Felipe M. G. França, and Vladimir Alves. 2020. Cost-effective, Energy-efficient, and Scalable Storage Computing for Large-scale AI Applications. *ACM Trans. Storage* 16, 4, Article 21 (October 2020), 37 pages.
https://doi.org/10.1145/3415580

1 INTRODUCTION

In the age of Big Data, the massive volume of data produced every day is posing a significant challenge of extracting useful information and knowledge from the data in a timely and efficient way. A study in 2012 revealed that only 0.5% of all data are ever analyzed due to a lack of effective methods to manage it [24]. The problem is especially prominent in the handling of visual data as companies such as Google and Facebook are experiencing unprecedented growth in the production of photos and videos [1, 5]. Thus, a development of scalable and robust methods is vital to efficiently search a large volume of visual content by indexing and retrieving it. In addition, recent developments in neural networks and deep learning approaches have greatly advanced a way of interpreting the visual data by understanding their context with a high degree of accuracy. This enables, for example, tracking multiple objects of interest in a given scene, which is crucial for autonomous systems such as self-driving vehicles [6, 21, 31, 33, 34, 46, 65, 85].

Similarity Search: Similarity Search is an effective way to analyze a large set of visual data where the basic computation primitive is to search for similar images and video frames when an image is given as a query. A common mechanism for the search starts with transforming images into high-dimensional, real-valued vectors by employing a set of machine learning techniques. Such feature vectors, called *embeddings*, are typically dense representations of input images obtained from pre-trained image-classification networks such as DenseNet [35] (Figure 1(a)). Because the networks produce similar vectors for similar images, distances (e.g., Euclidean) among the vectors can be measured to quantify how semantically similar the corresponding images are. Thus, given an image query, the problem of finding k most similar images is usually solved with a K-Nearest Neighbors (KNN) algorithm [14, 18, 30, 84] in the vector space.

Object Tracking: Object Tracking is an efficient method for tracking one or more objects through a large set of video streams (Figure 2(a)). The general process of tracking the objects consists of selecting *regions of interest* (ROI) that are tracked over the video frames, which can be performed by several machine learning techniques such as Correlation Filters [6, 32], Deep Neural Networks [31, 58], or Weightless Neural Networks [17, 23, 46]. Compared with Object Detection [7] whose goal is to find specified objects over the frames, object tracking can sustain a high throughput (in terms of frames per second) by referencing object information from previous frames, which is not possible in object detection. However, developing a fast, accurate, and general-purpose tracking system is still challenging due to the presence of noise, obstruction, and other errors.

1.1 Problem and Opportunity

AI applications require the computation of an enormous amount of (structured or unstructured) raw data to "learn," which makes the computing capability easily become a bottleneck to support the AI revolution. One approach that has been traditionally used to solve this issue is to use several nodes for more aggregated computing resources (such as high-end CPUs, GPUs, or FPGAs). Unfortunately, not only is adding more server nodes costly from both purchase and operating cost perspectives, but more importantly, this scale-out approach could increase interconnect length,

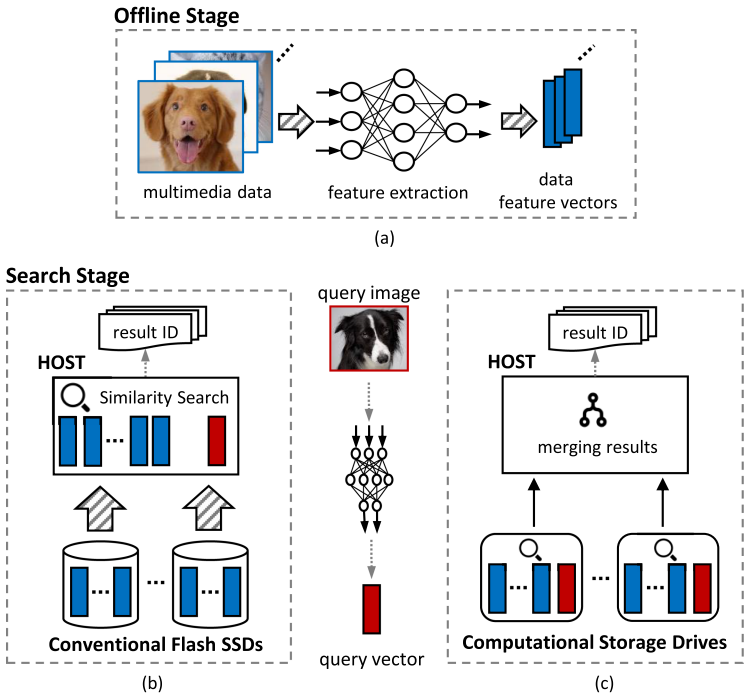

Fig. 1. Similarity search pipeline: (a) extracting feature vectors from dataset is performed offline. At the search stage, after extracting a feature vector from the given query image, many highly parallelizable distance calculations are performed (b) on host after moving the vectors from conventional flash SSDs or (c) inside CSDs.

which also increases the time and energy required for moving a huge volume of data from storage to compute planes. Given mind-blowing amounts of the data continue to be generated at an unprecedented rate (according to the IDC [60], the sum of the world's data will exceed 175 zettabytes by 2025), such storage I/O bandwidth limitation could create a big problem for the AI applications that need to interact continuously with storage systems. Furthermore, this problem gets accentuated with the astonishing growth of IoT and Edge deployments [75].

For example, the storage bandwidth limitation could be a significant bottleneck when performing a large-scale similarity search. As shown in Figure 1(b), at the search stage, feature vectors already created in the previous stage and stored on storage must be transferred to host memory to be processed. However, due to a massive amount of the vectors, it is likely that the bandwidth of conventional storage systems cannot keep up with the amount of many parallelizable distance calculations required for the search.[1] If the distance calculations, which are memory-intensive and computationally simple, could be performed near the vectors in storage, then the amount of data transferred from the storage to the host would be significantly reduced by returning only search results, as illustrated in Figure 1(c). Reducing the data movement could also improve the performance of object tracking applications. Considering the trend toward larger-sizes video frames with 4 K or 8 K resolutions, the entire set of frames may not fit into the host memory (Figure 2(b)). Therefore, executing object tracking algorithms on storage devices where the video frames are stored

[1]To decrease the data movement cost, the vectors can be encoded (e.g., Product Quantization (PQ) [37]) into a smaller number of bits, but this data reduction comes with the cost of loss of accuracy.

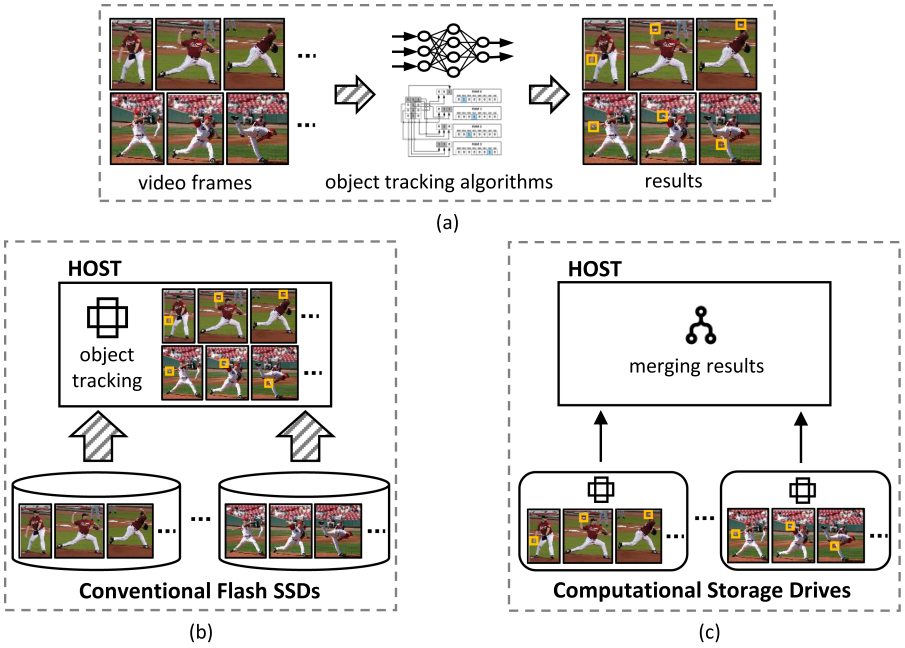

Fig. 2. Object tracking pipeline: (a) When a video stream is given, object tracking algorithms track desired objects with bounding boxes over frames belonging to the stream. (b) With conventional SSDs, a potentially very large set of video frames needs to be transferred to host first before computing the bounding boxes. (c) This computation, however, can be offloaded to multiple CSDs and be performed in parallel.

not only decreases the cost of moving data, but also frees up host resources that can be used for other activities running on the host (Figure 2(c)).

In the traditional Von Neumann architecture that has been in use for the past 70 years, storage devices are simply used to store data. The data have to be read first and then transferred to host before being processed. Fortunately, Computational Storage [16, 70], an emerging technology allows placing computing capabilities directly on storage (i.e., *in-storage processing*) to enable the data to be processed in the place where they reside. By bringing intelligence to the storage media itself, data reduction and qualification could occur before the data are sent to the host, resulting in the system's overall throughput being increased. In addition to facilitating faster processing, this also enables the host resources to be better utilized and to scale across a larger number of workloads—thus cutting costs for the enterprise. Cost efficiencies are further increased through a reduced power envelope, as well as through reduced demand for server memory and network bandwidth resources.

Modern flash SSDs combine processing and storage components to carry out routine functions required for managing the SSDs. These computing resources present interesting opportunities to run general user-defined programs, which has been studied in many works [15, 28, 39, 41, 43, 67, 76, 77, 78, 79, 80] in the past couple of years. While previous approaches have addressed various problems and challenges to advance the state-of-the-art, most of them lacked sufficiently dedicated in-SSD resources or adequate software frameworks necessary to execute various applications in a scalable manner to demonstrate the full potential of in-storage processing. Particularly, the lack of operating system support for the in-storage processing limited the capability and flexibility to perform large-scale, complex applications in many of these systems. Also, there is a risk that

existing large applications might need significant redesigns to take advantage of in-storage processing, requiring much time and effort.

1.2 Our Contributions

In this article, we describe *Newport*, a high-performance and energy-efficient computational storage drive (*CSD*) developed for realizing the full potential of in-storage processing. Newport is equipped with general-purpose, multi-core processors and multiple GBs of DRAM. To the best of our knowledge, Newport is the first commodity SSD that can be configured to run a server-like operating system (e.g., Linux). This feature enables general application developers to fully leverage high-level programming languages, existing tools, and libraries to minimize the effort for creating and maintaining applications running inside the storage. This also allows application developers to easily port large applications already running on host operating systems to the device with minimal or no code changes (by using, for example, the container technology such as Docker [50]). Newport can be configured to be used as either a general solid state drive (*Newport SSD*) or a full blown computational storage drive capable of processing data *in situ* (*Newport CSD*).

Designing and implementing a general-purpose CSD is our overarching contribution in this work. We describe aspects of our effort in the remainder of the article. We provide a background about SSDs and CSDs in Section 2. In Section 3, we present the hardware architecture and software framework of Newport. Section 3 also discusses the design and implementation aspects of Newport in detail. Then, we explore how two major types of AI applications that extensively use a large set of vision data can be efficiently executed at the storage-device level in Section 4. Another contribution is our evaluation of the resulting performance with respect to throughput and energy efficiency in Section 5. The results demonstrate the effectiveness of our effort. In Section 6, we analyze the cost of the most common and prevalent types of commercial CSDs, and we describe related works and a taxonomy of research-prototype and commercial CSDs in Section 7. We end with a short conclusion in Section 8.

2 BACKGROUND

2.1 Flash Solid State Drives (SSDs)

Solid-state drives (SSDs) are widely used in computer and storage systems today as a primary method of data storage. SSDs deliver significantly higher sequential (random) read (write) performance than hard disk drives (HDDs). In addition, SSDs can be in smaller form factors with much higher capacity consuming less static power, and they have better reliability and performance with regards to shock and vibration. The cost per bit of SSDs is still higher than that of HDDs but the gap is quickly closing especially with the advent of Quadruple-Level Cell (QLC) flash. QLC flash cells store a 4-bit value per cell. Encoding more bits per cell increases the capacity of a NAND flash die without increasing the chip size. The negative side effect is that it reduces reliability by bit error rate. These extraordinary technology advancements have led to the sweeping adoption of the SSD storage technology in private and public clouds, enterprise data centers, and even in consumer devices [36, 52, 73].

Any modern SSD is composed of two main components: SSD controller and non-volatile storage media—most commonly NAND flash [12]. NAND flash must be erased before data are written and can endure only a limited number of erases before it can no longer be used. The controller runs firmware to implement various functions to effectively manage NAND flash under these limitations. For example, garbage collection reclaims blocks containing invalid data and wear leveling that ensures that flash blocks are used evenly to prolong the SSD life. A Flash Translation Layer

(FTL) maps logical addresses to physical addresses so the logical view of storage is separated from the inner management of the physical memory.

The storage media is usually organized as a two-dimensional array of NAND flash dies grouped in multiple columns or channels. A flash channel is a bus used for communication between the SSD controller and a subset (a column) of flash chips. A chip consists of multiple blocks, each of which holds multiple pages. The unit of erasure is a block, while the read and write operations are done at the granularity of pages. To obtain higher I/O performance from the storage media, channel and chip-level interleaving techniques are usually employed. When the SSD controller handles an I/O request initiated by the host, it manipulates user data and the associated metadata, e.g., scrambling and unscrambling data to improve bit error rates, implementing error correcting codes such as LDPC encoding/decoding, and adding end-to-end data path protection.

There are different types of host interfaces that have been used over the years for SSDs, starting with legacy interfaces such as SAS and SATA [20]. NVM Express (NVMe) is the most recent standard host interface protocol that has been widely adopted across the industry. NVMe is a protocol for the transport of data over different media and for optimized storage in NAND flash. PCI Express (PCIe) is currently the most-used transport medium for NVMe while other media, like NVMe over Fabrics (e.g., Ethernet), are currently being developed and standardized. The NVMe protocol provides a high-bandwidth and low-latency framework to the storage protocol—in contrast with traditional HDDs, with flash-specific improvements. Several criteria must be used in selecting the appropriate SSD technology, such as price, capacity, performance, power efficiency, data integrity, reliability, and form factor. These criteria will vary in importance depending on target applications and use cases—client, enterprise, cloud, mobile, and so on.

Currently, there are many challenges that the SSD industry is facing. As an instance, the SSDs' execution model often mimics the legacy one based on HDDs, resulting in adding a latency penalty to compute and storage systems. In addition to the macroscopic forces of technology moving to new interface standards and storage media, the line separating computation and storage is getting blurred, which calls for the development of new generation of controllers. Some of the key factors driving controller technology changes are:

- NAND flash cell density that has been rapidly increased for the past 10 years from single-level cell (SLC), two-level cell (MLC), three-level cell (TLC), and more recently four-level cell (QLC) [36, 55]
- Flash technology going from planar to 3D manufacturing technology for increased die density to reach 512 GB and recently 1TB dies
- Ever higher NAND flash interface speeds ONFI3/Toggle2 (400 MT/s) to ONFI4/Toggle3 (800 MT/s)
- Higher data rates (e.g., PCIe Gen4) and new protocols (e.g., Ethernet)
- Higher defect densities and lower UBER (Uncorrectable Bit Error Rate) requiring increasingly powerful ECC engines, sophisticated RAID schemes, and end-to-end data path protection

2.2 Computational Storage Drives (CSDs)

Computational Storage is defined as an architecture that enables data to be processed within the storage device in lieu of being transported to the host server's central compute elements. Computational storage reduces the input and output transaction load through mitigating the volume of data that must be transferred between the storage and compute planes. As a result, it stands to better serve modern workloads such as high-volume big data analytics and AI tasks with faster performance and better data-center infrastructure utilization.

A primary benefit of computational storage is more energy-efficient data processing with higher system-wide throughput. Computational storage offloads work from the host compute elements, in favor of spreading that work across storage drives. Without computational storage, for example in the data analytics context, a request made by the host requires that all data from the storage device be transferred to it. The host must then thin down the data prior to performing its designated pass. In a computational storage approach, the storage media takes the initial step of qualifying data for their relevance to the host request before moving that data to the main compute tier to be processed. To further accelerate data processing speeds, CSDs could include energy-efficient multi-core processors, which enable multithreading, or FPGA/ASIC hardware accelerators. The reduced host compute instructions per workload means that the main server has more processing power available to support other workloads.

Another benefit of computational storage is that it makes a shared storage environment more beneficial to the most performance-hungry workloads. Typically, a direct-attached storage approach is used to serve these workloads to avoid storage network latency and to increase throughput by spreading the data across many devices. However, this often results in resource underutilization and also introduces further delay by needing to search more devices for the relevant data. In contrast, computational storage allows applications to be ported to each drive at the same time. This provides a level of parallel processing to enable a microservices-like approach to running those applications across all the individual drives. This ability to process the data simultaneously greatly reduces the time to locate the data and provide the host the results needed.

Computational storage can also help leverage the existing network infrastructure for much longer, as well as to truly scale next-generation networks. Because computational capabilities enable the storage to work on a larger dataset first, it leverages higher I/O capabilities of the storage device and avoids performance being restricted by a network. As a result, the network interconnect is less critical with computational storage. Thus, computational storage stands to add value by enabling multiple applications' performance to be accelerated on the same infrastructure, while at the same time optimizing utilization of infrastructure resources across the stack.

Modern SSDs have dedicated processing and memory elements (such as embedded processors, DRAM, and flash memory) to execute read, write, and erase commands on user data as well as the aforementioned flash management functions. With these computing resources, several projects [15, 28, 39, 41, 43, 67, 76] started to explore opportunities to run user-defined and data-intensive compute tasks such as database operations inside the storage device itself. While some performance and energy savings were observed, a few challenges prevented the broad usage and adoption of computational SSDs. First, the processing capabilities available are limited by design. The low-performance 32-bit real-time embedded processor and the high latency to the in-storage DRAM require extra careful programming to run user code to avoid performance limitations. Moreover, a flexible and general programming model is needed to easily execute in-place user code written in a high-level programming language (such as C/C++). The programming model also needs to support the concurrent execution of multiple in-storage applications with multiple threads to make it an efficient platform for complex user applications.

3 NEWPORT CSD

In this section, we introduce a new type of computational SSD that is carefully designed and developed to overcome the limitations of conventional SSDs with regard to in-storage processing. Newport, shown in Figure 3, provides dedicated computing resources and a complete software stack for near-data execution of user applications.

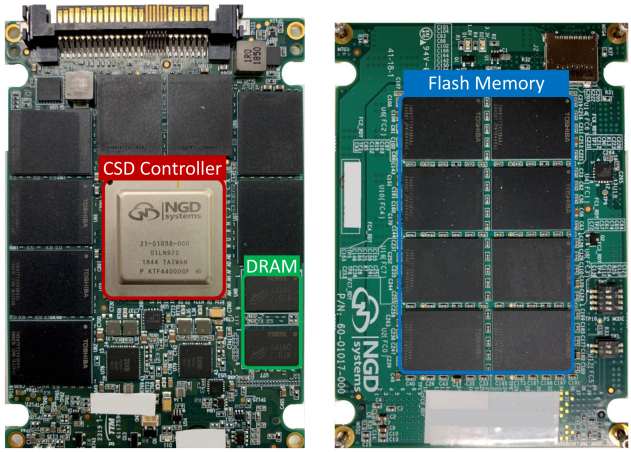

Fig. 3. The Newport CSD hardware in the 2.5-inch form factor.

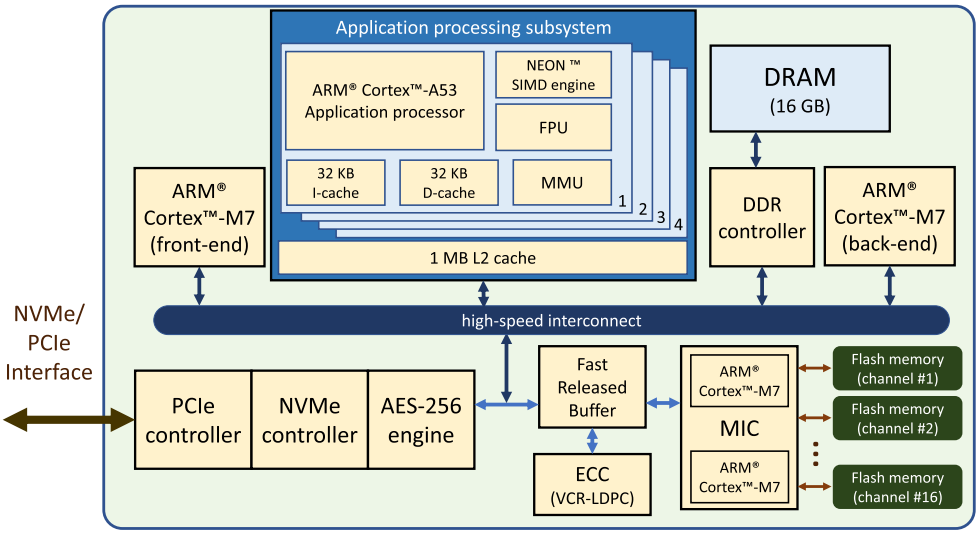

Fig. 4. Hardware architecture of Newport CSD.

3.1 Hardware Architecture

The in-storage processing technology aims at augmenting storage devices with efficient processing engines, enabling them to process data locally without transferring them to a host system. Despite the simplicity of the concept, many previously proposed CSD architectures fall short of providing compelling flexibility and efficiency. In contrast, Newport, an ASIC-based CSD architecture, provides a flexible environment for executing user applications with little or no modification. Newport enables *in situ* processing natively, which means its controller chip includes a dedicated 64-bit application processor subsystem in addition to other components necessary to realize the functionality of an enterprise-grade SSD—all integrated in a single ASIC chip. Figure 4 describes a high-level block diagram depicting Newport CSD controller's hardware architecture. The internal logic subsystems can be grouped into three sets:

- **Front-end Subsystem (FE):** A host server communicates with an SSD via a high-speed serial host interface (e.g., SATA, SAS, PCIe/NVMe). Newport's FE subsystem includes a Gen3 4-lane PCIe PHY and controller that feeds an NVMe protocol engine. Further down the data path an AES-256 XTS encryption/decryption engine ensures full-disk encryption functionality. The FE subsystem uses one of the four 32-bit embedded real-time ARM® M7 processors to execute the FE firmware. The NVMe controller under the control of the FE firmware receives, unpacks, checks the integrity, and executes the NVMe commands sent by the host. Read and write commands are then translated to requests to the back-end subsystem.
- **Back-end Subsystem (BE):** Its central piece is a 32-bit embedded real-time ARM® M7 processor running the BE firmware that includes mainly the flash memory logical-to-physical address translation layer (FTL), wear leveling, garbage collection, and flash reliability management [9, 11, 29, 86].

  A hardware-automated fast release buffer (FRB) collects and organizes host write data to optimize write latency. The same buffer is used for collecting and organizing data read from the flash media as the result of a host read command. The FRB architecture ensures full-duplex functionality to avoid any contention and to guarantee QoS by keeping read and write latencies constant. The ECC engine is used to detect and correct the errors that occur within flash memory or during data transfer between the flash die to the SSD controller to meet data storage reliability requirements. The FE firmware is responsible for splitting host data into codewords, which will include also a correction code or parity. The strength of protection offered by ECC is determined by the coding rate, which in turn translates into the size of the parity code. A lower code rate provides stronger protection but consumes more storage space. This is an important reliability vs. capacity tradeoff for the end customer. ECC algorithms typically used are variants of LDPC [48, 87] including variable code rate LDPC (VCR-LDPC).

  The Memory Interface Controller (MIC) is a module that establishes the connection to the flash memory. The storage media are organized as an array of flash memory chips each one containing a number of devices—typically 4 or 8, but as many as 16 for the highest density products. In this array, each row is connected to a flash memory channel. The connection for each channel is typically an 8-bit wide with some implementations opting for 16-bit wide bus between the controller and one of the flash memory chips. Both data and flash commands are sent over the channel. Each flash memory die connected to a flash channel has its dedicated chip enable (CE) signal. In each channel, the bus drives data, address, and commands to all dies tied to the channel.

  The MIC logic is quite complex and includes two ARM® M7 processors each dedicated to 8 flash channels for a total of 16 flash channels. The typical tradeoff in deciding the number of channels determines the maximum drive capacity vs. the controller chip footprint. The firmware running in this pair of embedded 32-bit real-time processors translates read and write commands issued by the BE firmware and generates low-level commands directed to a programmable finite state machine that synthesizes the correct control signals at the flash interface compatible with TOGGLE or ONFI standards [56].
- **Computing Subsystem (CS):** The computational storage processor subsystem is based on a feature-rich 64-bit quad-core ARM® Cortex-A53 (with its companion ARM® quad Neon SIMD engines), 1 MB L2 cache, and supporting up to 16 GB DDR4,[2] running up to 1.5 GHz.

[2]Note that Newport CSDs used for running experiments in Section 5 are equipped with 8 GB of DDR memory of which 6 GB is solely dedicated for running user applications.

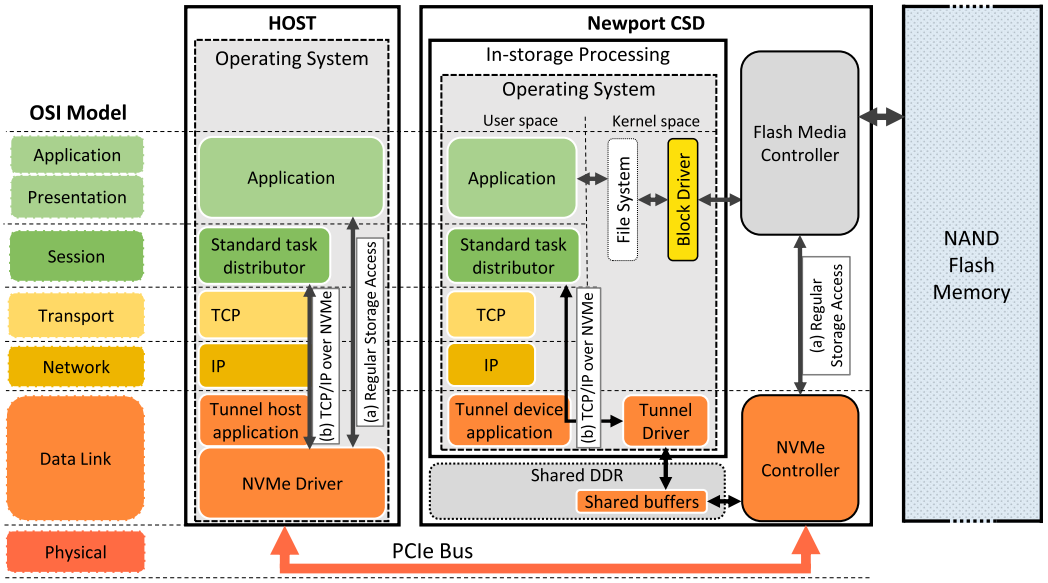

Fig. 5. Software architecture of Newport CSD: (a) The conventional data path through an NVMe driver for accessing non-volatile storage media via a PCIe bus. (b) The TCP/IP tunneling over the PCIe bus—mostly used as a control path.

All the subsystems in the Newport architecture are connected by a high-speed interconnect bus, as shown in Figure 4. Newport CSD is an enterprise-grade storage device that adheres to the power limitations dictated by industry standards relative to each form factor. *In situ* computing resources power requirements are included in the power envelope specified by the form factor storage standard. The same ASIC controller was used to develop CSDs compliant to different form factors such as M.2 22110, U.2 and EDSSF E1.S—each with its own power envelope. For example, the M.2 2210 standard [57] calls for a maximum power consumption of 8.25 W while the U.2 form factor accommodates for several different power envelopes (12 W, 15 W, and 25 W) [27] .

3.2 Software Stack

The main purpose of the software stack depicted in Figure 5 is to achieve a seamless programming model starting with a 64-bit operating system (OS) running on top of an ARM® processing subsystem. Compared with the embedded software development process that is complex and makes programming and debugging very challenging and time-consuming, Newport CSD enables application developers to fully leverage existing tools, libraries, and expertise instead. Newport SSDs/CSDs are 100% compatible with both PCIe and NVMe standards and show up as a standard PCIe device (endpoint) and then an NVMe device as the host boot process takes place. The configuration of the Linux Kernel and root filesystem determine the OS feature set as in any conventional Linux-based compute system—no modifications to the BIOS are required.

The software stack layers follow the recommended format of the standard Open Systems Interconnection (OSI) model [8]. For the sake of simplicity, we are showing only one Newport CSD connected to a host CPU.

On a Newport CSD, the boot procedure starts by bringing up the firmware that enables the basic SSD functionality. This is done by loading the appropriate code in the memory space of the few embedded 32-bit real-time processors throughout the controller. The purpose of these

embedded controllers (as depicted in Figure 4) is to implement the basic functionality of the solid state drive, e.g., read and write commands, garbage collection, wear-leveling, among many other critical functions—no user application runs on these processors. Then, part of the firmware initial setup includes loading a Uboot module and a Linux OS image in the application processor's memory space and executing the Uboot code. The Uboot code boots Linux OS with the desired configuration, including a block device driver module for accessing flash and a TCP/IP Tunnel device driver module to provide network connection. These modules will be registered at the boot time of the Linux OS. In Newport CSD, we have developed a TCP/IP Tunnel over NVMe/PCI protocol that will be established when the CSD's Linux is up and running. More detailed explanation of the TCP/IP Tunnel connection is provided later in this section.

As Figure 5 illustrates, both the host and Newport CSD can access data stored on NAND flash memory. From the host side, data access happens through the host's NVMe device driver, the CSD's front-end (PCIe phy and NVMe controller), and the CSD's back-end components, i.e., the NAND Flash media controller and interface. This path is shown in Figure 5(a). From the Newport CSD side, data access is provided through a custom block device driver that is added as a module to the kernel of the Newport's OS. The block device driver communicates directly with the Newport's back-end through two mailboxes. These mailboxes are communication links for the block device driver to send flash read and write commands to the SSD controller back-end, and also for the back-end to send back completion commands. To reach the maximum data rate, our block device driver enables scatter gather–based data transfer from back-end to the shared DDR memory.

As mentioned above in the description of the boot process, a critical layer of the CSD software stack is proprietary block device driver (see Figure 5) that interfaces to the controller's internal hardware accessing the Flash media and allows basic read and write operations originated by the OS or user applications. Deploying a conventional block device driver allows the OS running on Newport to access stored data through standard file systems. Thus, applications could process data in the concept of files, which is the most convenient and common way of programming. Depending on the target applications and the use cases, two approaches can be used for accessing storage from both host and Newport CSD. One simple approach is to divide the storage system into two (or more) separate partitions, one being available to the host and one to the Newport's OS and file system. When there is no need for sharing or concurrently accessing data between the host and the Newport through the storage, this approach can be used effectively. As an alternative, clustered and distributed file systems, such as GFS2 [72] and OCFS2 [22], can be used, allowing the host and the Newport to share the same storage partitions and to access them simultaneously, thanks to the locking mechanisms provided by the file system.

To synchronize the host and Newport file systems, distributed file systems need a network-based communication link to exchange control messages. Besides the synchronization requirement between the host and Newport filesystems, the proposed TCP/IP tunnel provides flexible and efficient means to use standard task distribution frameworks such as MPI [26], Hadoop [68], and OpenMPI [25]. The TCP/IP network connection also enables users to directly access the Linux operating system of the Newport CSD through protocols like "ssh." It provides a convenient mechanism to invoke and run applications on a Newport CSD both manually or using scripts. The TCP/IP Tunnel changes the essence of a Newport CSD to an independent processing node with a specific IP address that can be accessed remotely (it can be the host or even other computers in a network). Consequently, one can send a cross-compiled binary file to the Newport CSD (or even compile a source code inside the Newport CSD) and then run the binary file. A Newport CSD has the capability of being connected to the internet through the TCP/IP Tunnel. This allows users to download or install any application using standard Linux command lines. The TCP/IP link also enables automatic OS update mechanisms such as dpkg.packaging.system or apt-get.

To summarize, there are two communication mechanisms between a host and a Newport CSD: (1) the NVMe device interface for accessing non-volatile storage media attached via a PCIe bus—only used as a data path—and (2) the network-oriented connection based on the TCP/IP tunneling over the PCIe bus—mostly used as a control path. To better understand the efficiency of each communication mechanism, we measured their read and write performance with widely used tools for network and storage I/O performance measurements: FIO[3] for the NVMe interface and iPerf[4] for the TCP/IP connection. The Newport CSD tuned for this measurement provides a peak raw read/write performance on the NVMe/PCIe bus of approximately 4,000 MB/s and average sequential read/write performance of approximately 1,600 MB/s. Currently, the TCP/IP tunnel connection (control path) runs at approximately 210 MB/s. The difference in performance for the TCP/IP connection compared to the NVMe path can be explained by two factors: (1) there are three layers of packetization (PCIe, NVMe, and TCP/IP) that collectively add overhead, and (2) there is an inefficiency in copying data from the shared buffers located in the Newport DDR memory. It is worth noting that we are currently working on optimizing various software stack components to mitigate such overhead and inefficiency. Furthermore, the flexibility of the CSD platform opens the door for the exploration of other communication protocols such as content-centric networking protocols. However, despite such inefficiencies this approach to computational storage provides a scalable solution that augments system performance and energy efficiency, as presented in Section 5.

3.2.1 Running an Application Step-by-step. In this section, we provide an overview of the steps that a developer would take to execute a generic application on a Newport CSD, as well as some insight on the underlying communication happening, detailed above. When the host machine is turned on, Newport CSD starts to boot its own Linux operating system. After the boot-up, the tunnel application is brought up to allow communication between the host and the *in situ* Linux OS. This allows for the user to communicate and send data via TCP/IP (if necessary). At this point, the user has many options on how to execute the application. It is important to keep in mind that for an *in situ* execution, one must compile the application on the device or use some type of cross-compilation technique if the host architecture is not compatible with the CSD's. The application has many ways of accessing host and *in situ* files, as explicitly stated in this section. If required, application communication can take advantage of sockets, MPI, and other well-known libraries. When it comes to executing the application itself, the CSD is similar to an independent computer on a heterogeneous cluster.

4 APPLICATIONS

4.1 Similarity Search

Search for similar visual content can be a hard task, especially with the surge of data. Similarity Search is a two-step algorithm capable of analyzing these large sets of data to retrieve similar contents, given an input query. It leverages the power of deep neural networks to extract features from the data and store them into vectors. Instead of finding similar contents from the raw data, it applies a K-Nearest Neighbors (KNN) algorithm in the feature vector space.

The first part of the algorithm (Figure 1(a)), entitled the offline stage, is mainly responsible for extracting essential features from a set of images. These features are usually presented as vectors collected from the last hidden layer of a deep neural network. Once the vectors for all dataset images are obtained through the offline stage in the similarity search pipeline (Figure 1(a)), the

[3] https://github.com/axboe/fio.
[4] https://iperf.fr/.

next step is to conduct the search stage to find k "closest" vectors in a multi-dimensional vector space. As the most straightforward way of getting the top k closest vectors, we can use a brute-force approach that computes all the vector distances—precisely and exhaustively—between the query vector and each of the other vectors in the dataset. However, the difficulty of calculation of the vector distance,[5] growing proportionally to the size and dimension of the data, makes the brute-force search far more expensive.

As a way of avoiding the necessity to calculate distances to all of the vectors at query time, multi-dimensional space is partitioned to multiple cells, and it is presented as an inverted file index (IVF) [69] where data vectors belonging to the same cell are stored as a flat array, called an inverted list. This inverted index structure (called IVF-Flat) holds pointers to all inverted lists, providing fast access to any of them. Once a query is given, the most relevant inverted lists to the query vector are scanned for finding its nearest neighbor vectors.

4.1.1 Problem Statement. Given a d-dimensional query vector $x \in \mathbb{R}^d$ and a collection of base set vectors, $Y = [y_i]_{i=1:n_d}$, ($y_i \in \mathbb{R}^d$), Flat search method computes the full pair-wise L2 distances between the query vector and the base-set vectors without encoding them; i.e., we can search the k nearest neighbors of x as:

$$L_{Flat} = k\text{-}argmin_{i=1:n_d} \|x - y_i\|_2. \tag{1}$$

This brute-force approach, however, could be very expensive due to a large number of distance calculations among vectors. In contrast, the IVF-Flat index considered in this article is not exhaustive, as we compute distances only between the query and a subset of base-set vectors based on a coarse quantizer, f, which is defined as:

$$f : \mathbb{R}^d \rightarrow C \subset \mathbb{R}^d, \tag{2}$$

where C is a set of coarse centroids obtained at the offline stage by clustering base-set vectors. In other words, the d-dimensional vector space is partitioned into $|C|$ cells, and therefore base-set vectors, Y, are grouped into $|C|$ inverted lists in the IVF-Flat index.

$$L_{IVF} = p\text{-}argmin_{c \in C} \|x - c\|_2 \tag{3}$$

$$L_{IVF-Flat} = k\text{-}argmin_{i=1:n_d \ s.t. \ f(y_i) \in L_{IVF}} \|x - y_i\|_2 \tag{4}$$

The probe parameter, p, is the number of coarse-level centroids we consider during the search. The similarity search with IVF-Flat is performed in two steps—selecting relevant cells first (Equation (3)) and then doing a flat search by linearly scanning inverted lists belonging to the selected cells (Equation (4)). Therefore, the number of pair-wise distance calculations performed during the search is reduced by a fraction of $p/|C|$.

4.1.2 Batch Similarity Search. In practice at query time, it is common for a similarity search application to maintain data vectors stored in respective inverted lists in memory. However, as the size of the datasets grow, it becomes extremely difficult to keep all relevant data in memory at once. One common solution to the problem is using virtual memory on flash SSDs and paging to expand the amount of visible memory. While being economically attractive, this solution could result in a poor search throughput performance if the same inverted lists need to be loaded to the memory multiple times for different query vectors (see Section 5.3.1 for more details).

[5]There are many different ways of calculating vector distance. The canonical distance metric is the Euclidean distance (a.k.a., L2 distance), with other popular ones such as Cosine, Manhattan, Jaccard, and Chi-squared metrics. In this article, we focus on the Euclidean metric.

A more efficient approach is dividing the dataset into smaller batches and gathering the batch results to produce the final result. This approach is hereafter referred to as *Batch Similarity Search (BSS)*. It begins with splitting the whole dataset into smaller batches and then builds an inverted index structure over a batch (so each could fit in memory) during the offline stage. If we assume that the base set, Y, is divided into $[Y_j]_{j=1:b}$ batches, then Equations (2), (3), and (4) are rewritten as:

$$f_j : \mathbb{R}^d \rightarrow C_j \subset \mathbb{R}^d, \tag{5}$$

$$L_{IVF_j} = p\text{-}argmin_{c \in C_j} \|x - c\|_2, \tag{6}$$

$$L_{IVF-Flat_j} = k\text{-}argmin_{i=1:n_d \ s.t. \ f(y_i) \in L_{IVF_j}} \|x - y_i\|_2. \tag{7}$$

At the search stage, once a query vector is given, similarity search is performed against the first IVF-Flat. After finding the k closest neighbors from the first batch, the next IVF-Flat is loaded into memory, and the search is performed to find another set of k closest neighbors against the second batch. In other words, the BSS approach involves updating k closest neighbors after performing similarity search on each batch, and at the end, we have k closest neighbors from the whole dataset. The pseudo-code of the BSS approach with a query vector, x, and a set of IVF-Flat indexes, I, over b batches of the entire dataset is shown in Algorithm 1.

ALGORITHM 1: Batch Similarity Search

```
Input: (x, I=[I_1, ..., I_b])
Output: L_IVF-FLAT
Function BBS(x, I):
    L_IVF-FLAT = {}
    foreach I_j ∈ [I_1, ..., I_b] do
        Load(I_j)
        L_j = Search(x, I_j)
        L_IVF-FLAT = L_IVF-FLAT ∪ L_j;
    end
    return k − MinDistSort(L_IVF-FLAT);
End Function
```

As shown in the algorithm, we first load each IVF-Flat index and search k nearest neighbors from the index (Equation (7)) serially. At the end, all collected nearest vector indices are sorted based on the calculated distances between x and $y_i (i \in L_{IVF-FLAT})$ by the **k-MinDistSort** function.

4.1.3 Distributed Batch Similarity Search. In a distributed environment, we further enhance the search performance with Distributed BSS where distributed nodes process dedicated portions of IVF-Flat indexes in parallel. Algorithm 2 shows the pseudo-code of Distributed BSS when there are s distributed nodes in the distributed environment.

As shown in Algorithm 2, the **Partition** function returns a subset of IVF-Flat indexes that is processed by the mth node (see Section 5.3.3 for the partition scheme we used in experiments), and the BSS function (i.e., Algorithm 1) is called to get the search results. After gathering all vector indices computed by distributed nodes in parallel, the **k-MinDistSort** function is used to return the final k nearest neighbors.

It is worth to note that BSS with a single query can be easily parallelized to deal with a set of query vectors simultaneously, which increases the efficiency of the search with multiple CPU threads.

ALGORITHM 2: Distributed Batch Similarity Search

Input: (x, I=[$I_1, \ldots, I_b$])
Output: $L_{IVF-Flat}$
Function `DistributedBSS`(x, I)**:**
  $L_{IVF-FLAT} = \{\}$
  Parallel for $m = 1;\ m \leq s;\ m = m + 1$ **do**
    I_m = **Partition**(I)
    L_m = **BSS**(x, I_m)
    $L_{IVF-FLAT} = L_{IVF-FLAT} \cup L_m$;
  end
  return k − **MinDistSort**($L_{IVF-FLAT}$);
End Function

4.2 Object Tracking

This section gives an introduction to selected object tracking algorithms based on Deep Neural Networks (DNNs), Correlation Filters (CFs), and Weightless Neural Networks (WNNs). It is important to highlight here that due to the Newport CSD being developed for general application execution, no changes were made on the codes for executing the algorithms on different CPU environments.

DNNs powered by recent breakthroughs in training and convolution have been applied on computer vision problems to great success. Unfortunately, there is a deficiency for visual object tracking training datasets, which forces the use of pre-trained convolution networks on big classification datasets such as ImageNet. Although these networks do provide unprecedented accuracy and robustness, they also tend to be computationally hungry, requiring powerful devices such as GPUs for real-time tracking. Other techniques, such as CFs and WNNs, follow different approaches that can keep a high frames per second (FPS) rate with minimum resources requirements and training time. They can enable features like online training, which uses the first frame to learn about the object and infer the position on the following frames. As the object changes they can also provide training updates as an adaptation mechanism. However, there is a tradeoff in accuracy and robustness to achieve such high FPS rates with few requirements.

4.2.1 Deep Neural Network Trackers. Deep Neural Networks (DNNs) have gained popularity with breakthroughs in a wide range of tasks, such as computer vision and natural language processing. The ability to handle large sets of data to approximate complex functions has established DNNs as the state-of-the-art technique on classification and regression problems. Convolutional Neural Network (CNN) is a popular DNN architecture composed of a stack of computational layers responsible for extracting low, middle, and high-level features from the input data. It operates in two phases: training and inference. During the inference phase, the data flows through the layers, extracting the features, and outputting the probabilities for each predicted class. The training phase consists of an inference phase followed by a loss function that calculates how far the prediction was from the expected output. This loss is backpropagated through the network so each layer can calibrate its parameters (weights) based on the previous prediction [34].

GOTURN [31] is a single target object-tracking algorithm based on CNNs that can achieve high FPS by applying an offline training scheme. It leverages the potential of CaffeNet [38], an eight-layer deep network pre-trained with labeled images and video frames to track novel objects during the test mode. The model requires a cropped version of the target object as input to be tracked on the remaining frames. It tracks the object movement by using a search region based on the

previous location. Although the model achieves high fps on a GPU device, which is crucial for real-time applications, it can only handle single target.

Real-time object-detection is an alternative approach to track objects in a frame-by-frame fashion. Instead of predicting the movement of the target object, it keeps detecting it in every frame. YOLO [58] is a state-of-art technique for multi-object detection in real-time systems that can detect up to 9K targets in a single frame. Differently from most detection algorithms, YOLO requires just a single step over the image/frame to detect several objects. It splits the input into grid cells that are fed into a 53-layer deep CNN, called Darknet, which predicts a set of bounding boxes. Each predicted bounding box has four coordinates (location and dimension) and a confidence score that tells how close that prediction was from the ground truth. Darknet also classifies each object inside a predicted bounding box by leveraging a set of pre-trained layers from a classification task.

The third version of YOLO, known as YOLOv3 [59], outperforms a set of other methods in terms of inference time and presents competitive accuracy results. However, the technique is not optimized for non-GPU systems, making the FPS rate to plunge in CPU-only platforms. To overcome this fact, YOLO-Lite [58], a lightweight version of YOLO, was created. It targets real-time systems with a limited amount of computational resources, such as autonomous vehicles and smart devices. YOLO-Lite shrinks the network to eight-layer deep and sacrifices a good portion of accuracy to achieve a 10× better FPS compared with the traditional GPU version.

4.2.2 Correlation Filters. Correlation Filters are an online training algorithm that use image template as filters to discriminate through images [81]. By using correlation operations on the Fourier domain, they are able to speed up computation and to provide online updates for the filters as frames are processed. This approach allows for learning on the fly and makes object tracking more robust.

The Minimum Output Sum of Squared Error (MOSSE) Filter [6] is an object tracker based on adaptive correlation filters. It uses the first frame ground truth position to create and train the first filter. By using random affine transformations the tracker creates a small training set that initializes the filter. To find the best-suited filter on training it looks for the minimum sum of the squared error between the output of the convolution and the desired output. Tracking is done by correlating the filter with a search window on the subsequent frames. Both the frame and filter are converted into the Fourier domain and apply a Fast Fourier Transform Convolution operation. This process is required, because convolution operations are an element-wise multiplication on the Fourier domain, decreasing overall computational complexity. MOSSE is able to perform online training updates on the filter, as well as detecting when the object has left the frame, resuming tracking in case it returns by using Peak to Sidelobe Ratio measurements.

Kernelized Correlation Filters (KCF) [32] apply a Gaussian Kernel on Histogram of Oriented Gradients features instead of raw pixels values to perform object tracking. It uses the properties of circular matrices with the Discrete Fourier Transform and a so-called "kernel trick" to reduce computational complexity. KCF uses the first frame to train the model for the chosen target, but it uses an area value larger then the ground truth. On the subsequent frames it updates the position based on the image patch with maximum value. At this point a new model is trained for the new object position. Unlike MOSSE, KCF does not provide any type of failure mechanism, being unable to detect if the object has left the frame.

4.2.3 Weightless Neural Network Tracker. Weightless Neural Networks (WNNs) [2] have been explored on object tracking tasks in several previous works [17, 23, 46]. WNNs are Artificial Neural Networks where each neuron is represented as a Random Access Memory (RAM) node. By modelling the neurons in a binary format using existing memory resources in computer systems,

Fig. 6. High capacity in-storage processing platform with Newport CSDs.

these networks offer an attractive solution to pattern recognition and even artificial consciousness applications, obtaining competitive performance in comparison to the state-of-the-art techniques.

Differently from other trackers explored in this work, the WNN object tracker was built from the ground up. The implementation was inspired and based on Reference [17], which proposes a tracker based on short- and medium-term memories to store the WiSARD Discriminators [3]. Such implementation does not require any previous training and supports online training updates. In the first stage of the algorithm the ground truth regions of interest (ROI) of the first frame are used as the training base for the first discriminator. For each next frame, a search is performed around the last object position by using a slide window technique. All windows are sent to all Discriminators from a queue to classify the corresponding object.

5 EXPERIMENTS

This section presents experimental results from the execution of three applications running on our target server configured with CSDs. It is worth to mention that the selection of the applications tested in this article is not motivated by a preconceived perception of possible advantages with regards to the near-data processing coupled with distributed processing. Rather, our motivation is to address computation challenges currently relevant in the industry that potentially involve the storage and analytics of large amounts of data, and to explore how computational storage can be effectively integrated with conventional setup for improving the overall system gains.

The first application, text compression, demonstrates the computational scalability and energy efficiency of our host server combined with multiple Newport CSDs. Next, two AI applications are explored considering the above-mentioned metrics and different computational task load mappings: similarity search and object tracking. These two AI applications are thought to be natural candidates to run over big data stored in CSDs. Similarity search was mapped into and executed in parallel by the host server and CSDs; computational scalability and different host-CSD balances were tested together with energy efficiency. With respect to object tracking, five different object trackers were easily ported and tested on CSDs.

5.1 Hardware Setup

To explore the performance and energy-consumption benefits of using computational storage drives, we conducted experiments on a high-capacity in-storage processing platform shown in Figure 6. The platform supports dual Intel® Xeon® Silver 4108 processors, each of which contains 8 cores (16 threads) running at 1.8 GHz, and 64 GB of DRAM. The server supports up to 24 NVMe U.2 SSD bays to provide a highly dense pure flash array solution. Each Newport CSD provides up to 32 TB flash memory capacity and is equipped with four ARM® Cortex-A53 cores running at

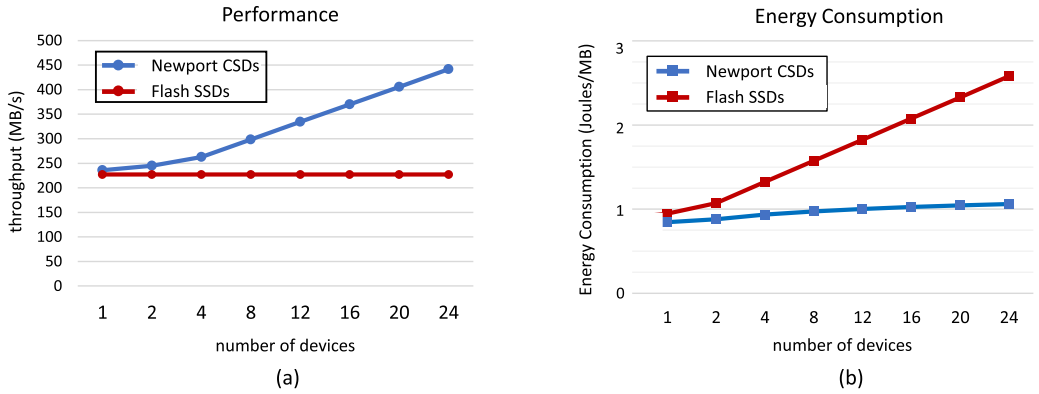

Fig. 7. Compressing a large set of text data on a system with CSDs or SSDs: (a) compression throughput and (b) energy consumption per data unit.

1.2 GHz, which are solely dedicated to user applications, not for regular SSD management tasks. Thus, the server could support up to 768 TB storage capacity and add 96 ARM cores available for processing data *in situ* with 24 Newport CSDs. For both the server and Newport CSDs, 64-bit Ubuntu® 16.04 LTS is used. The power consumption of the platform is measured with a Wattman HPM-100A power meter[6] with the accuracy of 0.4% and a logging interval of 1 second.

5.2 Text Compression Benchmark

In this section, before evaluating the system with Newport CSDs with the AI applications introduced in Section 1.1, we create a baseline for performance and energy consumption using a simple compression scheme. (De)Compression is a common functionality that is used in conjunction with storage and networking, mainly to save data movement and storage costs. However, these cost savings come from the use of expensive host computing resources. The goal of this experiment is to explore how computational storage can augment system performance and energy efficiency compared with a configuration using conventional SSDs where the host is solely responsible for compressing the data but also assessing the impact of other variables such as dataset size, CPU performance, among others.

For the compression scheme, we used gzip,[7] which is one of the most popular algorithms used in many applications mainly due to its superior compression ratio (in some cases, gzip can achieve up to 70% compression ratio [71]). With the dataset consisting of about about 1.7 TB text files extracted from the Gutenberg online library,[8] we measured the throughput and energy consumption required for compressing the raw text files stored in storage for the following two system configurations: (a) a system with *Newport CSDs* where the text data are locally compressed inside the storage devices in addition to the host CPU; (b) a system with enterprise-grade *flash SSDs*[9] where the text data are first loaded from the storage devices and then compressed on the host.

5.2.1 Performance. Figure 7(a) shows the throughput performance with respect to the amount of data that are compressed in a second. As can be seen in the figure, we achieved almost linear

[6] http://shop2.adpower21com.cafe24.com.
[7] https://www.gzip.org/.
[8] https://www.gutenberg.org/.
[9] The conventional SSD used for the text compression experiment could provide up to 3.5 GB read/write performance for sequential data.

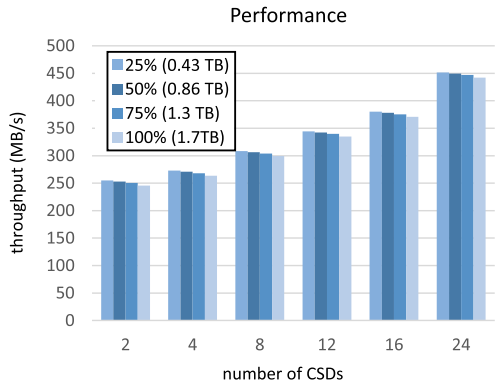

Fig. 8. Compressing subsets of the dataset on a system with CSDs.

performance gains by adding Newport CSDs that could additionally perform the compression task in parallel. Interestingly, regardless of the number of SSDs, we observed that host CPU becomes a system bottleneck due to the host processing power stuck at around 225 MB/s. The compression speed of the host CPU could not keep up with the speed of transferring large amounts of the text data from the SSDs. As another configuration, we also conducted the same experiment with a single processor. As expected, the host processing power stuck at a lower speed (i.e., around 140 MB/s) while we achieved 2.42× higher throughput with 24 CSDs.

Figure 8 shows the results of compressing subsets of the dataset, each of which contains the first 25%, 50%, and 75% of the text data, respectively. As shown in the figure, the performance scalability remains mostly unaffected by the dataset size (within less than 3% performance variation).

5.2.2 Energy Consumption. The amount of energy consumption per data unit, required for compressing the text data, is presented in Figure 7(b). For each test, the following three steps were followed to calculate the energy consumption: (1) we first measured the power drawn from the idle-state server (Figure 6) with no drives, resulting in 78.5 W; (2) for each test, over the elapsed time, the difference between time-discrete real power values of the system with drives and the base idle-state system power (78.5 W) was summed; (3) then the summed energy consumption was divided by the total text data size in MB.

As shown in the figure, we observed a dramatic increase in energy consumption, adding more SSDs to the system. For example, with the SSDs, the energy consumption required to compress 1 MB data is 2.57 J per MB. In contrast, the processors packaged inside the CSDs performed the same compression task in a much energy-efficient way (i.e., 1.06 J/MB), resulting in 2.4× energy consumption gain.

5.3 Similarity Search

To explore the effectiveness of processing a large-scale similarity search in storage, we used Faiss (version 1.2.1),[10] a library released by Facebook for efficient similarity search and clustering of dense vectors.[11] Faiss is developed for 64-bit x86 platforms and has pure C++ code optimized for Intel SSE/AVX instructions. We compiled Faiss to the ARM® architecture and optimized its operations for the ARM® SIMD operations as well. In addition, for a task distribution purpose between

[10] https://github.com/facebookresearch/faiss/.

[11] We note that our proposed approach explored in this article is not restricted to a specific similarity search method. Other well-known similarity search libraries are found in Reference [4].

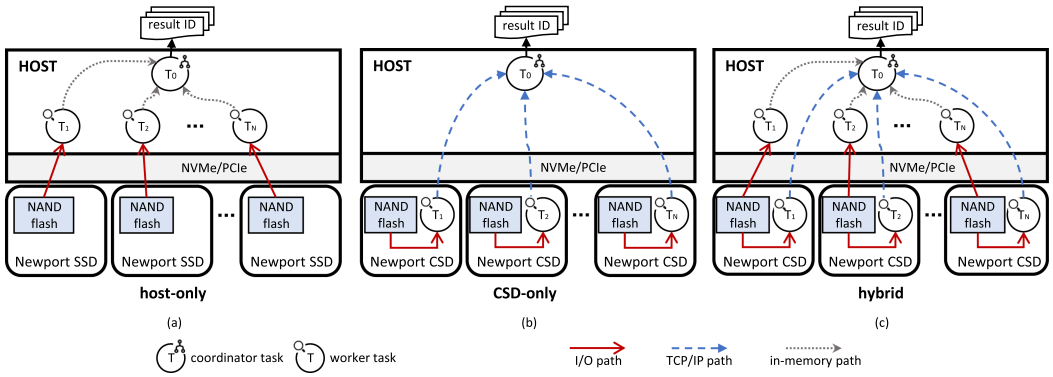

Fig. 9. Server configurations for image similarity search: (a) host-only: the search is solely performed on the host processing unit; (b) CSD-only: the search is performed inside storage devices in parallel; (c) hybrid: the search is performed with all compute resources available in the host and the CSDs.

host and CSDs (for more details, see "Standard Task Distributor," described in Section 3.2), we used Open-MPI[12]—an open-source MPI library that is widely used in both academia and industry settings.

We carried out all the experiments for the comparative analysis with respect to performance and energy efficiency on a large dataset, ANN_SIFT1B,[13] which contains 1B SIFT [47] feature vectors of 128 dimensions extracted from public images. We first used a subset of 50M vectors of the base set to investigate various aspects of the in-storage processing performance in Sections 5.3.1–5.3.3 and then used the full base set of 1B vectors to demonstrate the impact on the large-scale search in Section 5.3.4. In addition to the base set of 1B vectors, ANN_SIFT1B consists of a learning set of 100M vectors, and a query set of 10K vectors.

During the offline stage of the similarity search pipeline (Figure 1(a)), an inverted file (IVF) index is built with 4,096 cell centroids that are formed by clustering the learning-set vectors with a flat coarse quantizer. After that, an IVF-Flat index is created by inserting each base-set vector to the closest cell of the IVF index. As discussed in Section 4.1.2, we performed Batch Similarity Search (BSS) where the dataset is divided into small batches, and an IVF-Flat index is created per each batch. During the search stage (Figure 1(b) or 1(c)), a query vector is compared to all the base-set vectors belonging to candidate cells (out of 4,096 cells) in the IVF-Flat indexes. The number of the candidate cells searched for the query is hereafter referred to as "nprobe," and it is given as a search parameter. The accuracy of the search is given by the recall factor. Recall at k is the proportion of relevant items found in the top k recommendations. In this article, *recall@k* is defined as follows:

***recall@k* = (# of queries for which k nearest neighbors are returned in the first k results) / (total # of queries)**

In the experiments described in this section, the BSS is performed on the system (Figure 6) with Newport CSDs in the following three different setups, as shown in Figure 9: (a) *host-only:* The Newport CSDs are configured to be used as general SSDs (i.e., Newport SSDs) where the in-storage processing feature is disabled. In this setup, the entire BSS is conducted only by the host processing unit. (b) *CSD-only:* The Newport CSDs are configured to be used as computational storage (i.e., Newport CSDs) where the in-storage processing feature is enabled. In this setup, the

[12] https://www.open-mpi.org/.
[13] http://corpus-texmex.irisa.fr/.

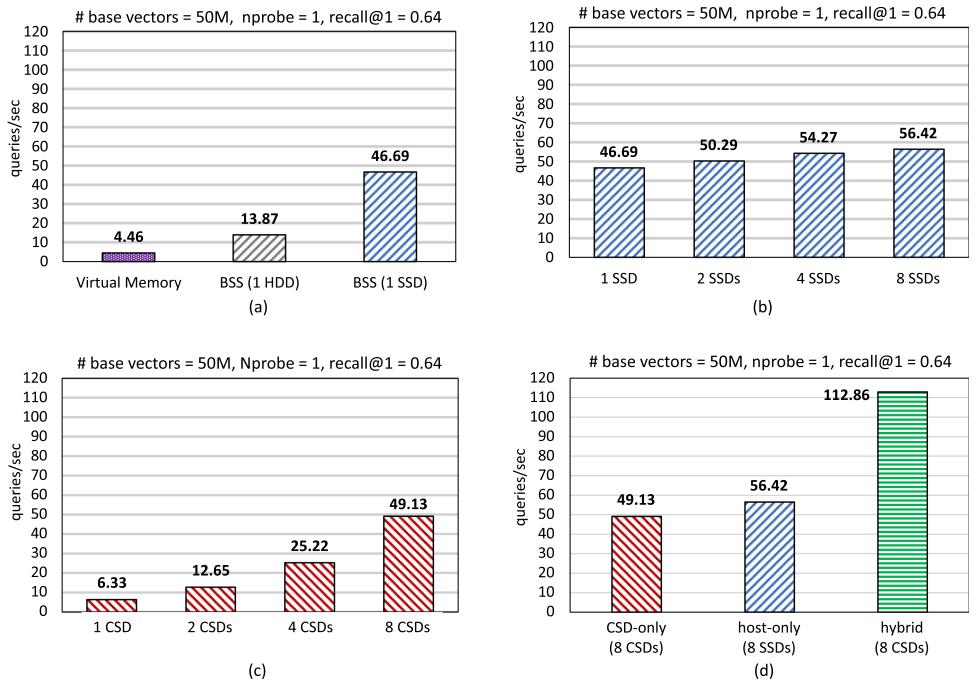

Fig. 10. Similarity search throughput: (a) virtual memory vs. single SSD (or HDD) performance; (b) the host-only processing with SSDs; (c) the CSD-only processing with CSDs; (d) the hybrid approach compared with the host-only or the CSD-only approaches.

entire BSS is performed inside the storage devices in parallel. (c) *hybrid :* the Newport CSDs are used to increase the aggregated processing capability of the system. In this setup, the BSS tasks are distributed to the host and the CSDs so the search can be done by using all processing resources available in the system.

We measured the elapsed time taken to load the IVF-Flat indexes and to search nearest neighbors of all query-set vectors to calculate query throughput (i.e., queries/sec). For optimal performance, we parallelized search loops over the query-set vectors via OpenMP [13], which supports shared memory multiprocessing programming. Note that in all configurations illustrated in Figure 9, a single coordinator task is created and run on the host to gather search results from many distributed worker tasks through MPI_Gather, one of the collective communication routines defined in the MPI standard specification.[14]

5.3.1 Host-only Processing. When a dataset does not fit into memory, one easy but naive solution for dealing with this situation is to use virtual memory, as mentioned in Section 4.1.2. This solution is attractive, as no application or system configuration changes are needed, but could result in poor search performance. We first examined the effectiveness of performing the BSS on a Newport SSD against the virtual memory solution over the subset of 50M vectors on the host-only configuration (illustrated in Figure 9(a)). The results are shown in Figure 10(a). With the virtual memory the host OS needs to constantly swap the vector data back-and-forth between the host memory and the storage device, resulting in a significant performance drop. In contrast, the BSS on the SSD allows to load each batch of vectors only once, and therefore no additional I/Os are

[14] https://www.mpi-forum.org/docs.

occurred to load the same vector data, contributing to around 10× performance gain. This figure also shows the performance benefit of using an SSD, comparing with a HDD.

Figure 10(b) shows the host-only processing performance of executing the BSS as increasing the number of Newport SSDs. The figure indicates that the host processing capability does not keep up with the speed of transferring vectors from the SSDs. While we observed a slightly increased throughput by adding more SSDs into the platform, the performance improvement was not linear, because the processing power of the host CPU caused bottlenecks in the entire search time.

5.3.2 Scalable Performance of In-Storage Processing. Next, we investigated the scalable performance of in-storage processing on the CSD-only configuration illustrated in Figure 9(b). In this configuration, the BSS is performed by energy-efficient compute resources available inside Newport CSDs. The host only acts as a coordinator that collects the results returned from the CSDs. As shown in Figure 10(c), we observed that the throughput scalability of the similarity search is almost close to linear as the number of CSDs increases. The processing approach leveraged in the CSD-only configuration mimics a shared-nothing architecture in which each CSD can operate independently from the other CSDs. Thus, a cluster of the CSDs enabled the processing power to be scaled almost linearly by adding more CSDs into the cluster. In addition to the scalable processing power, the aggregated I/O bandwidth available inside the CSDs also can scale linearly. Compared to this, in the host-only configuration, each SSD has its own set of physical PCIe lanes, which might reduce the available bandwidth on the PCIe bus due to the increased change of sharing the lanes, as described in Section 5.3.1.

5.3.3 Hybrid Computation with Host and CSDs. Figure 9(d) shows a hybrid scenario where the host and Newport CSDs perform similarity search in coordination to maximize the performance by using all resources available in the system. In this scenario, one important problem is to optimally partition the search task over multiple powerful (but expensive) processors in the host and energy-efficient processors in Newport CSDs. A detailed analysis on different partitioning schemes is out of the scope of the article. Instead, we used a simple approach of dividing the base-set vectors in two groups—one is processed by the host and the other by the CSDs—so the host and the CSDs perform their assigned search tasks and complete them almost simultaneously. This approach can be written as an optimization problem, as described in Equation (8):

$$\begin{aligned} \underset{d_{host}}{\text{minimize}} \quad & T_{host}(d_{host}) \\ \text{subject to} \quad & T(d) = T_{loading}(d) + T_{searching}(d) \\ & D = |d_{host}| + N \cdot |d_{csd}| \\ & T_{host}(d_{host}) = T_{csd}(d_{csd}), \end{aligned} \tag{8}$$

where the function T represents the total time taken to load and to search a set of vectors, d. T_{host} and T_{csd} denote the elapsed times taken by the host and one Newport CSD, respectively. d_{host} and d_{cost} are subsets of the full base vector set, D, and N is the number of Newport CSDs. For example, when $nprobe = 1$, $N = 8$, and $D = 50M$, our numerical analysis of the optimization problem results in $d_{\text{host}} \approx 29M$ and $d_{\text{csd}} \approx 2.6M$.

Figure 10(d) shows the comparison of the hybrid approach against the host-only and the CSD-only processing. As shown in the figure, processing similarity search on the system with eight Newport CSDs augmenting the host CPU improves the query throughput about 2× compared to the host-only processing with eight SSDs. We repeated the same experiment with a higher nprobe parameter (see Section 5.1) from one to four to search a bigger vector space. The results shown in Figure 11(a) indicate that searching more neighboring cells in the vector space could provide a higher accuracy (i.e., recall@1 = 0.96) with a significant reduction in query throughput.

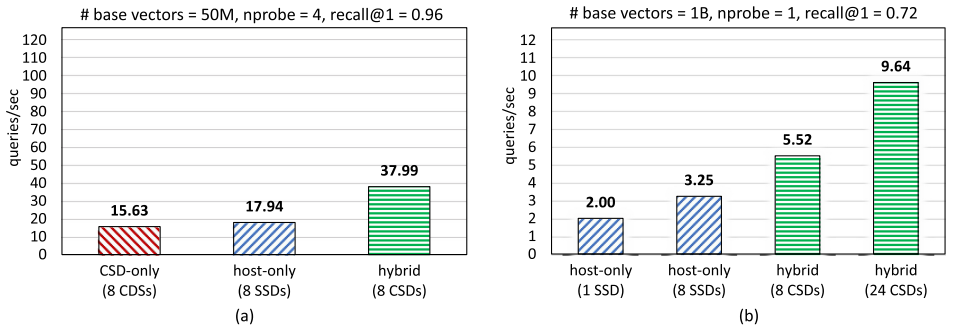

Fig. 11. BSS performance throughput achieved with the host-only, the CSD-only, and the hybrid configurations: (a) 50M vectors with nprobe=4; (b) 1B vectors with nprobe = 1.

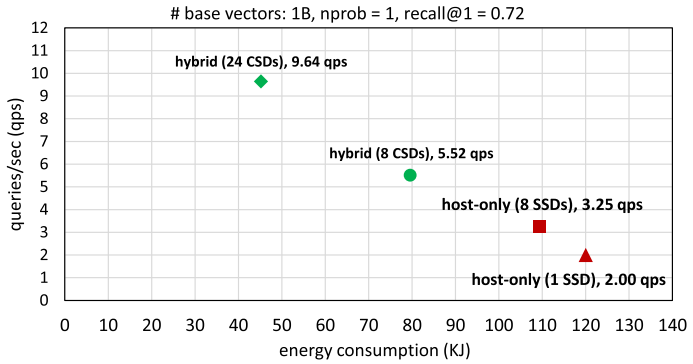

Fig. 12. Performance and energy consumption of BSS with 1B vectors.

It is because more search computations are required to find nearest neighbors of the given query vectors. Compared with the host-only processing with eight SSDs, the hybrid approach improved the query throughput (2.1× speedup), demonstrating that the performance improvement of the proposed hybrid approach is appropriate for the increased computation.

5.3.4 A Large-scale Search with 1B Vectors. We performed a large-scale image similarity search with the entire dataset of 1B vectors using the hybrid approach and compared it with the host-only processing. The results are shown in Figure 11(b). The hybrid approach of distributing the workload across the host and eight CSDs improved the query throughput by 41% compared to the host-only processing with eight SSDs. The system with 24 CSDs performed about 4.8× better than the system with a single SSD. Each Newport CSD adds its own computational power and internal data bandwidth without interfering each other. Thus, the performance of a wide range of applications processing large-scale parallel workloads would be improved by efficiently leveraging in-storage processing.

Figure 12 shows the throughput and the energy consumption when performing BSS on the host-only and the hybrid configurations with 1B vectors, respectively. It is clear from the figure that not only the hybrid approach outperforms the host-only processing, but also is much more energy-efficient. For example, the hybrid approach with 8 CSDs consumed about 27.3% (= 29.9 KJ) less energy than the host-only processing with 8 SSDs. Also the system with 24 CSDs consumed 45 KJ, which is only 37.6% of the energy consumed by the host-only processing with a single SSD. The

Table 1. Selected Object Trackers with Computing Resources They Have Been Tested on

Computing Resource	YOLO	GOTURN	KCF	MOSSE	WiSARD
GPU	✓	✓	–	–	–
CPU	✓	✓	✓	✓	✓
CSD	✓	✓	✓	✓	✓

results in this figure indicate that the way in which we build a high-performance, energy-efficient system might change by effectively leveraging computational storage.

5.4 Object Tracking

In this section, we measured the performance of executing multiple well-known object tracking algorithms.[15] Table 1 shows the selected object trackers and the compute resource that they have been tested on. Among the object trackers, it is worth to note that only YOLO[16] and GOTURN are implemented to be executed on GPU. In addition, YOLO is the only one that supports the multi-thread CPU execution. We first expected to have GOTURN using multi-cores as well, since it is compiled with the CaffeNet framework where the multi-thread execution is enabled, but that was not the case. So, we used a multi-process approach—we populate multiple tracker processes, each of which is assigned to a single core to perform an object tracking algorithm on a video stream. In this way, we can keep all the cores busy while working on different video loads. We used the same approach for other trackers that support the CPU execution.

For the visual tracking dataset, we used OTB-100 [85] as a test workload for all experiments conducted in this section. Many tracking applications [45, 51, 54] have evaluated their inference performance with OTB-100. This dataset consists of 100 videos of different resolutions and number of frames. Each video is tagged with a series of 11 categories indicating their most relevant characteristics, such as illumination variation, partial occlusion, and motion blur. Each frame of a video has a ground-truth, bounding-box annotation that marks the correct position of a target. For each object tracker, we used a pre-trained model provided by its respective authors.

We used NVIDA GeForce RTX 2080 Ti for the GPU execution. Since other trackers do not support, we only ran YOLO and GOTURN on the GPU. The CSD execution is solely performed by total 12 CSDs without augmenting the CPU.

5.4.1 Throughput. Table 2 shows the throughput performance in terms of frames per second (FPS) when running object trackers with different computing environments. To calculate the overall throughput, we measured the total elapsed time to complete tracking on all the test videos. For the CSD execution, it is important to balance CSD workloads so all CSDs would be fully utilized at the same time in parallel. For this purpose, we implemented a simple greedy load balancing algorithm to reduce the discrepancy between execution times in the CSDs. In addition to the overall throughput, we measured the average throughput of tracking each video ("Avg. Per-Video Throughput" in Table 2). As mentioned in Section 5.4, we had to use the multi-process (not multi-thread) approach to make all the cores busy due to the limitations of each tracker's

[15] Each object tracking algorithm belongs to three different types: Deep Neural Network (DNN), Correlation Filters (CF), and Weightless Neural Networks (WNN), across three different computing resources—GPU, CPU, and CSD, respectively. For more details, see Section 4.2.

[16] Note that we used YOLOv3 for the GPU execution, and YOLO-Lite for the CPU/CSD execution.

Table 2. Throughput Performance and Energy Consumption of Each Object Tracker Running on Different Computing Resources

Computing Resource	Object Tracker	Overall Throughput (FPS)	Avg. Per-Video Throughput (FPS)	Energy Consumption (J/Frame)
GPU	YOLOv3	34.79	34.66	6.03
	GOTURN	194.83	199.26	2.13×10^{-1}
CPU	KCF	1,145.85	220.69	4.38×10^{-2}
	MOSSE	3,310.22	493.70	1.48×10^{-2}
	WiSARD	1,568.00	72.59	3.95×10^{-2}
	YOLO-Lite	2.72	2.76	15.26
	GOTURN	21.24	2.03	2.42
CSD	KCF	186.78	32.09	1.51×10^{-2}
	MOSSE	1,295.30	143.87	5.54×10^{-3}
	WiSARD	199.28	11.96	1.35×10^{-2}
	YOLO-Lite	2.36	0.23	2.00
	GOTURN	1.83	0.12	6.26×10^{-1}

implementation. Thus, while this parallel approach improved the overall throughput, it did not affect the per-video throughput, simply because a video is processed by a single core.

As shown in the table, for most object tracker applications, the overall and the per-video throughput was reduced as the device computational power decreased, as expected. However, by parallelizing the workload process with the multiple CSDs, we observed that the CSDs processed object tracking for all videos within a reasonable amount time. If you used CSDs to augment the host CPU like the hybrid approach we did in similarity search (See Section 5.3), we would expect higher throughput.

Among the object trackers we tested, the MOSSE filter had the fastest time as well as the highest per-video throughput, but it did not provide enough robustness for the whole dataset (we explain this in Section 5.4.3). The WNN approach with WiSARD showed great promise with good overall throughput time and also high per-video throughput. For the DNN trackers (i.e., YOLO and GOTURN), both had an orders of magnitude throughput reduction as the computing resource gets weaker (i.e., from the GPU to the CPU, then the CSDs). GOTURN had the lowest per-video throughput compared to other trackers running with the same type of computing resource (CPU or CSD). We believe its implementation is heavily optimized for the GPU execution. (Again, even though the CaffeNet framework has a native support for making use of multi-thread execution, GOTURN is still executed sequentially.) Interestingly, YOLO achieved very low throughput on both the CPU and the CSDs while it had multi-thread execution enabled. Since YOLO is originally built for object detection, which does not have dependencies from the previous frames, it treats each frame as a still image; therefore, it requires heavier computation if compared to the other object tracking algorithms. By having the CSDs working on the same video, but on different set of frames, we would improve the in-video throughput further.

Recently ARM has made available a new Neural Network SDK[17] that optimizes ARM instructions and functions on well-known neural network frameworks such as Tensorflow. Given the potential huge performance gains by taking advantage of this new technology (e.g., more than

[17] https://www.arm.com/products/silicon-ip-cpu/machine-learning/arm-nn.

Table 3. Average Accuracy (IoU) of Each Object Tracker

	MOSSE	KCF	YOLO-Lite	YOLOv3	GOTURN	WiSARD
Average IoU	0.1985	0.1832	0.1673	0.3588	0.4140	0.3543
Standard Deviation	0.2555	0.2183	0.1978	0.2980	0.2414	0.2439

10× in some cases [64]), it is worth optimizing the DNN-based object tracking for the ARM environment to get closer to a real-time performance, which is left as future work.

5.4.2 Energy Consumption. For the energy consumption measurement, we first collected the machine power output when it is in the idle state (i.e., only the OS is executing), which is our baseline. Then, we executed each object tracking method as measuring the active power in Watts during the execution. Based on the differential power consumption between these measurements (and the total elapsed execution time), we derived the energy consumption in Joules per Frame (J/Frame) for each tracker on a computing resource.

The Energy Consumption column in Table 2 presents the results. Looking at the overall results, trackers had better energy efficiency when being executed on the CSDs (compared to the CPU or GPU executions). Complex trackers such as YOLO and GOTURN required much more time to complete their executions compared with other trackers, resulting in around an order of magnitude higher energy consumption on the CPU. By offloading such trackers to the CSDs, we consumed much less energy, thanks to energy-efficient processors available inside the CSDs. Interestingly enough, the GPU execution of GOTURN consumed less energy than the CSD execution. We believe this is because GOTURN is highly optimized for GPU, and therefore it required a huge amount of computations only during a relatively short period of its execution time. YOLO's energy consumption was always bigger compared with other trackers on the same computing resource, which is expected, given that YOLO is more computationally intensive, since it is implemented based on object detection. Other CF-based and WNN-based trackers (MOSSE, KCF, and WiSARD) consumed much less energy than the DNN-based approaches, mainly due to a smaller amount of computation. As we describe in Section 5.4.3, however, such energy savings came with the cost of loss accuracy in the CF-based methods (but not as much in WiSARD).

5.4.3 Accuracy. In this section, we measured accuracy in object tracking methods. For the accuracy metric, we used Intersect over Union (IoU), which is one of the most popular metrics when evaluating the accuracy of object tracking systems [44]. IoU can be computed by dividing the intersection area by the union area of two ground-truth and predicted bounding boxes. The IoU output is in the range of zero and one, where one represents a 100% match. An IoU value above 0.5 is considered a very good match.

Table 3 shows the average IoU of all the tested videos on each object tracker. (For the accuracy numbers of each individual video, see Appendix A.) As shown in the table, more complex trackers such as YOLOv3 and GOTURN achieved better overall accuracy. YOLO-Lite, which is designed to provide a real-time object detection model on portable devices, is less complex and detects less objects than YOLOv3 by nature, and its accuracy was not as high as YOLOv3. Note that other trackers from the filters (i.e., MOSSE and KCF) did not provide enough robustness for the whole dataset, but still had some samples with better accuracy than the DNN-based approaches (see Appendix A). The WNN-based WiSARD method, however, achieved a good accuracy, which is comparable to YOLOv3. Table 3 also shows the standard deviation of accuracy. One of the main reasons of such high standard deviations is because the OTB-100 dataset is composed of a diverse pool of videos with different types of noises, as well as different target sizes. Therefore, unless a

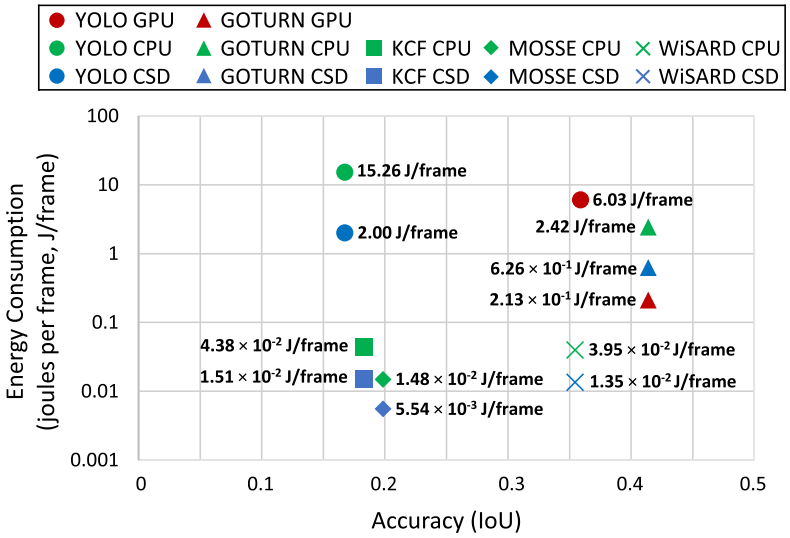

Fig. 13. Energy consumption and accuracy of executing object trackers on different computing resources.

tracker is robust enough to support all those varieties, getting relatively low accuracy in some sets of videos is difficult to avoid.

Figure 13 shows a tradeoff between energy consumption and accuracy of each tracker running on different computing resources. The results are presented in a log scale on the y-axis, since some energy-consumption values are too close to be well displayed. GOTURN's highest overall IoU coupled with its low energy consumption makes it the clear that it is best targeted to be optimally executed on a GPU device. The WiSARD tracker also had a great IoU coupled with being orders of magnitude faster than the DNN-based trackers on both the CPU and the CSDs as providing great energy efficiency, especially on the CSD execution. While the CF-based trackers also provided good energy efficiency, the accuracy was relatively low.

6 COST ANALYSIS

Compared with an enterprise-grade SSD based on a conventional controller, it is expected that a CSD should contain an additional processing resource such as a more powerful processor to efficiently run a variety of software applications. (Some CSD types make use of FPGA based hardware accelerators. For more details, see Section 7.) It is also fair to assume that CSDs would require larger DDR memory (for the application computation) in contrast to the conventional SSD.

Interestingly, the SSD bill of material (BOM) analysis [10, 42] shows that the main cost difference between an SSD and a CSD would be marginal, given that the current SSD price is largely dominated by the cost of NAND flash chips. The exception being for FPGA-based CSD implementations due to the comparatively very high cost of FPGA devices. For example, in the first quarter of 2020 an enterprise-level 8 TB NVMe SSD is priced starting at around $1,600 in the open marketplace, and the spot price of NAND flash TLC 512 GB chips is approximately $0.15 per GB (source: *https://www.dramexchange.com*) at the time of writing this article.[18] Other miscellaneous

[18]While the prices of an SSD and NAND flash devices have wide variability at any time and more over time, the analysis provides a good big picture of the economic feasibility of CSDs. Even though NAND flash price per GB has been continuously dropping, both NAND flash density and average SSD capacity have been increasing steadily, therefore compensating for it when considering the cost for a CSD unit.

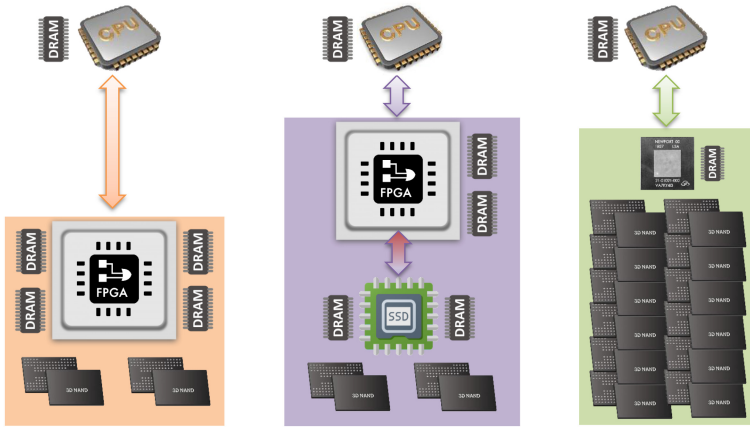

Fig. 14. Commercial CSDs implementation approaches.

Table 4. 8 TB Commercial CSDs Cost Analysis—High Volume

Components	FPGA as Controller	Drive + FPGA	ASIC-based
FPGA	$800	$800	–
8 TB NAND	$816	$816	$816
DRAM	$128	$192	$128
Passive Components	$75	$75	$75
PCB	$50	$50	$50
CSD ASIC Controller	–	–	$20
SSD Controller	–	$15	–
Power Loss Capacitors	$5	$5	$5
TOTAL	$1,874	$1,953	$1,094

costs (such as DRAM, miscellaneous components, and manufacturing costs) can account for 10% to 25% of the CSD price. The amount of additional memory will vary for different designs but again it is not expected to be a significant factor of total cost.

In this section, we analyze the cost of the most common and prevalent types of commercial CSDs, as shown in Figure 14. These types of CSD designs have been actively discussed at a technical working group of Storage Networking Industry Association (SNIA) [70], which was set up in the year of 2019 to standardize device interoperability, management, and security among SSD vendors developing a number of different technologies and approaches to CSDs. Figure 14(a) depicts the FPGA-based controller approach [66]. This design provides significant flexibility, thanks to the re-programmable logic at the expense of a convoluted programming model and higher cost. The second variant, labelled *FPGA + SSD* [19, 49, 66] is depicted in Figure 14(b). Reference [41] presents very similar drawbacks from the cost point of view. Finally, Figure 14(c) depicts the ASIC-based architecture represented by the Newport CSD [74].

Table 4 shows the estimated cost for low and high volumes of a mid-range capacity 8 TB NVMe CSD. The costs of the main components (DRAM, FPGA, off-the-shelf SSD controller, and NAND flash) are obtained from References [82, 83]. When one excludes NAND flash from the BOM cost, as shown in Figure 15, the ASIC-based approach turns out to be approximately 4× less costly.

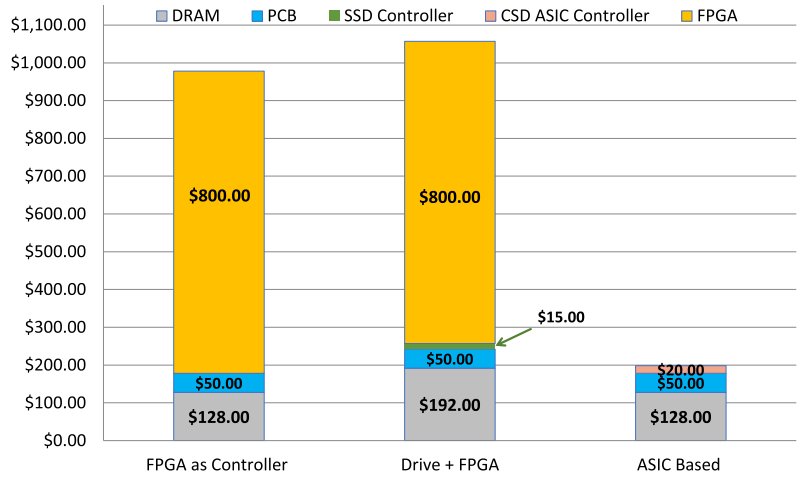

Fig. 15. 8 TB Commercial CSDs relative cost—NAND flash excluded.

7 RELATED WORK

A modern flash SSD contains computing components, such as an embedded processor and DRAM memory, to perform various tasks, as described in Section 2.1. This characteristic of the SSD provides interesting opportunities to run user-defined programs inside the SSD.

Do et al. [15] were the first to explore such opportunities in the context of database query processing. They modified a commercial database system to push down selection and aggregation operators into a SAS flash SSD. With the same type of SSD, Kang et al. [41] extended the MapReduce framework to process a sequence of operations on distributed datasets. They pioneered the creative use of flash SSDs to open up cost-effective ways of processing data, but their approaches were primarily limited by hardware and software aspects of the SSD. First, the embedded processors in the prototype SSDs were clocked at a few hundred MHz and were not powerful enough to run various user-defined programs. More importantly, the embedded software development environments made the development and analysis very challenging, preventing thorough exploration of in-storage processing opportunities. In contrast to these prototypes, Catalina SSD provides powerful processing capabilities with abundant in-SSD computing resources, and the flexible development environment with a general-purpose operating system allows easy programming of the CSD.

It has been shown that deploying hardware accelerators, like FPGAs, is a very promising solution to gain both performance and energy efficiency [61, 63]. In some of the previous works, storage devices are supplemented with FPGAs for the sake of in-storage processing [40]. Although FPGAs are very power-efficient and potentially provide a considerable performance improvement through process parallelism, implementing a FPGA accelerator requires a deep understanding of hardware design and the design process is very time-consuming. However, for many applications with sequential behaviors, it is not possible to beat the performance of general-purpose processors due to the low working frequency of FPGAs [62].

Recently, several studies have studied better programming models for computational storage. In Reference [67], Seshadri et al. proposed Willow—a PCIe-based generic RPC mechanism—allowing developers to easily augment and extend the SSD semantics with application-specific functions. Gu et al. [28] explored a flow-based programming model where an in-SSD application can be dynamically constructed of tasks and data pipes connecting the tasks. These programming models

Table 5. Taxonomy of Published Research-prototype and Commercial CSDs

	Processing Resources	Programming Model	Media Access	Communication to Host
Research Prototypes				
• Do et al. [15]	• 32-bit embedded processor (< 200 MHz)	• Firmware (C code)	• Raw access	• SAS
• Kang et al. [41]	• 32-bit embedded processor (< 200 MHz) (shared with firmware)	• Firmware (C code)	• Raw access	• SATA
• Seshadri et al. [67]	• 32-bit embedded processor (< 200 MHz)	• Primitive OS (C code)	• Raw access (DRAM emulator)	• PCIe (RPC protocol)
• Jun et al. [40]	• FPGA	• Bluespec [53]	• Raw access	• PCIe (RPC protocol)
• Gu et al. [28]	• 32-bit processor (< 800 MHz) • Pattern matcher HW	• Flow-based model (C/C++)	• Raw access	• PCIe
Commercial Products				
• Samsung [49]	• FPGA	• Bare metal	• Raw access	• PCIe
• ScaleFlux [66]	• FPGA	• Bare metal	• Raw access	• PCIe
• Eideticom [19]	• FPGA	• Bare metal	• Raw access	• PCIe
• NGD [74]	• 64-bit processor (> 1.5 GHz) • FPGA	• Full-fledged OS (Linux)	• Clustered file system • Disk file system • Raw access	• PCIe (RPC protocol)

offer great flexibility of programmability, but are still far from being truly general-purpose. There is a risk that existing large applications might need to be heavily redesigned based on models' capabilities.

Table 5 provides a taxonomy of existing research-oriented and commercial CSDs to offer a common terminology for easy understanding and comparison. In the literature, Reference [28] describes a brief summary of research prototypes, which compare key properties of programming environment and system realization. In this table, we propose a new and complete taxonomy (by extending the previous summary) based on the major properties observed in currently existing CSD solutions.

8 CONCLUSIONS

This article explored Newport, a storage drive that can simultaneously work as an enterprise-grade solid state drive and as a computational storage unit capable of *in situ* processing of its data. Two data-intensive AI applications, image similarity search and object tracking, were chosen with the purpose of demonstrating how computational storage can enhance computational performance and reduce energy consumption at the system level. Experimental results over workload-dependent applications have shown that it is possible to achieve significant improvements with respect to performance gains and energy reduction by taking advantage of parallelism and near-data processing in large-scale distributed systems.

One of the key characteristics of the Newport computational storage platform is its simple and accessible programming model, which makes possible to execute in-storage such large-scale applications. Such approach allows for application developers to fully leverage existing high-level languages, tools, and libraries to minimize the effort to create and maintain applications running in-storage. This suggests how computational storage can be used to implement a scalable architecture that addresses the growth of datasets, whether centralized/localized or distributed, such as in edge computing use cases. In the experiments presented, we observed that heavy workloads can be efficiently parallelized, even by a small set of Newport drives. Last a realistic cost analysis for high-volume production of existing computational storage commercial solutions was provided, which gives a good big picture of the economic feasibility of computational storage drives.

A ACCURACY MEASUREMENT PER VIDEO

Table 6. Tracker Accuracy in IoU per Video with the OTB-100 Dataset

Video Names	MOSSE	KCF	YOLO-Lite	YOLOv3	GOTURN	WiSARD
Basketball	0,0000	0,0037	0,2219	0,4074	0,3516	0,1705
Biker	0,0000	0,1080	0,0383	0,0421	0,6772	0,3379
Bird1	0,0000	0,0031	0,0081	0,0517	0,0557	0,0443
Bird2	0,6878	0,0394	0,0031	0,1004	0,4741	0,3373
BlurBody	0,0000	0,0678	0,7434	0,8234	0,3126	0,4842
BlurCar1	0,1078	0,2455	0,5126	0,7249	0,0882	0,1037
BlurCar2	0,6256	0,2405	0,7525	0,8003	0,5117	0,0903
BlurCar3	0,1504	0,1846	0,4774	0,7897	0,0866	0,0713
BlurCar4	0,6652	0,7362	0,4550	0,6735	0,2039	0,5616
BlurFace	0,3126	0,5442	0,0652	0,1275	0,5703	0,1857
BlurOwl	0,0473	0,0887	0,0041	0,0089	0,1181	0,1366
Board	0,6678	0,6161	0,0001	0,0010	0,1060	0,1743

(Continued)

Table 6. Continued

Video Names	MOSSE	KCF	YOLO-Lite	YOLOv3	GOTURN	WiSARD
Bolt	0,0000	0,0043	0,0264	0,5655	0,0034	0,0132
Bolt2	0,0000	0,0052	0,1342	0,6108	0,3756	0,0080
Box	0,2520	0,0696	0,0011	0,0019	0,1239	0,6068
Boy	0,0000	0,1345	0,0731	0,0466	0,8045	0,6262
Car1	0,0830	0,0782	0,1333	0,6992	0,1569	0,7257
Car2	0,6588	0,6144	0,6944	0,7559	0,7598	0,8408
Car24	0,1138	0,4091	0,4195	0,8312	0,7831	0,6589
Car4	0,4686	0,2091	0,7240	0,8203	0,7857	0,8296
CarDark	0,6226	0,5400	0,0052	0,0812	0,0331	0,5365
CarScale	0,4080	0,3953	0,3696	0,8698	0,7258	0,3587
ClifBar	0,1172	0,0624	0,0260	0,1197	0,1791	0,5874
Coke	0,1550	0,0542	0,0000	0,0384	0,5014	0,1780
Couple	0,0000	0,0179	0,4489	0,6681	0,6013	0,0499
Coupon	0,8841	0,4614	0,0000	0,0758	0,7899	0,8629
Crossing	0,0000	0,0602	0,0253	0,7245	0,4860	0,6424
Crowds	0,0000	0,0076	0,0011	0,5773	0,4981	0,4114
Dancer	0,3982	0,1112	0,3818	0,4848	0,4825	0,7507
Dancer2	0,6988	0,2493	0,4862	0,6312	0,6434	0,7345
David	0,1026	0,0219	0,0533	0,0732	0,2303	0,1744
David2	0,0319	0,1648	0,0427	0,0493	0,4423	0,6589
David3	0,0000	0,0732	0,2168	0,6134	0,1985	0,0844
Deer	0,0776	0,0082	0,0350	0,0582	0,2179	0,0344
Diving	0,2131	0,1098	0,0763	0,4208	0,5057	0,1858
Dog	0,0119	0,0185	0,1963	0,2574	0,2666	0,4779
Dog1	0,4431	0,4235	0,2041	0,4095	0,3957	0,7738
Doll	0,0000	0,3067	0,0111	0,0615	0,1154	0,7439
DragonBaby	0,0400	0,0206	0,0855	0,1084	0,4578	0,1809
Dudek	0,7077	0,7364	0,1460	0,1779	0,8048	0,2661
FaceOcc1	0,0000	0,7586	0,0874	0,4264	0,5528	0,7565
FaceOcc2	0,6263	0,7272	0,0491	0,1696	0,6031	0,5535
Fish	0,7186	0,0544	0,2484	0,1198	0,7362	0,6923
FleetFace	0,4577	0,5760	0,1599	0,1922	0,7127	0,5292
Football	0,4329	0,2707	0,0339	0,0540	0,3587	0,5212
Football1	0,0000	0,0266	0,0106	0,0643	0,4873	0,4614
Freeman1	0,0000	0,0926	0,0717	0,0768	0,6677	0,3107
Freeman3	0,0000	0,0565	0,0744	0,0918	0,7699	0,0009
Freeman4	0,0000	0,0435	0,0308	0,0886	0,2063	0,1366
Girl	0,0212	0,0212	0,0855	0,2390	0,5086	0,2697
Girl2	0,3130	0,6674	0,3131	0,6815	0,1072	0,1178

(Continued)

Table 6. Continued

Video Names	MOSSE	KCF	YOLO-Lite	YOLOv3	GOTURN	WiSARD
Gym	0,0000	0,0121	0,2790	0,6519	0,4437	0,2652
Human2	0,0000	0,4411	0,5783	0,6972	0,7062	0,4520
Human3	0,1239	0,0026	0,2111	0,7604	0,0262	0,0058
Human4	0,1114	0,0565	0,0433	0,5848	0,0523	0,0959
Human5	0,0000	0,1058	0,1486	0,7995	0,6212	0,4174
Human6	0,0000	0,0063	0,2959	0,6643	0,6954	0,1883
Human7	0,3103	0,1702	0,3475	0,8425	0,4773	0,2236
Human8	0,4641	0,0215	0,2581	0,8553	0,2002	0,0951
Human9	0,0000	0,0166	0,2558	0,8289	0,5012	0,0738
Ironman	0,0000	0,0155	0,0307	0,0509	0,2039	0,0592
Jogging	0,0000	0,0002	0,0322	0,0367	0,0122	0,1261
Jump	0,0000	0,0747	0,1243	0,3910	0,1254	0,0559
Jumping	0,0474	0,0382	0,0609	0,0633	0,5235	0,1427
KiteSurf	0,0370	0,1491	0,0905	0,0790	0,3706	0,4545
Lemming	0,1253	0,0204	0,0063	0,0123	0,3620	0,3131
Liquor	0,3077	0,6085	0,4136	0,7496	0,3139	0,3114
Man	0,1434	0,1781	0,0737	0,0745	0,8030	0,7461
Matrix	0,0000	0,0052	0,0176	0,0374	0,0669	0,1346
Mhyang	0,7498	0,6930	0,0914	0,1333	0,6844	0,8329
MotorRolling	0,0000	0,0764	0,0482	0,3474	0,5296	0,0872
MountainBike	0,7633	0,0639	0,0172	0,5409	0,7076	0,2473
Panda	0,0018	0,0262	0,0033	0,2765	0,4680	0,4260
RedTeam	0,0667	0,4761	0,0010	0,0281	0,3004	0,4942
Rubik	0,2122	0,0966	0,0221	0,3202	0,2372	0,1958
Shaking	0,0000	0,0048	0,0439	0,0983	0,7889	0,1279
Singer1	0,0000	0,3503	0,1800	0,6312	0,5727	0,4683
Singer2	0,6609	0,0422	0,1563	0,3939	0,0418	0,5953
Skater	0,4669	0,0441	0,6152	0,6016	0,6933	0,4064
Skater2	0,0503	0,1782	0,5903	0,6469	0,6694	0,3916
Skating1	0,1608	0,0410	0,1644	0,7110	0,5442	0,2738
Skating2	0,0000	0,0003	0,0705	0,0705	0,0751	0,0644
Skiing	0,0000	0,0085	0,0043	0,1661	0,5413	0,0765
Soccer	0,0918	0,0484	0,0059	0,0278	0,1162	0,1291
Subway	0,0000	0,0489	0,0953	0,6163	0,0711	0,1518
Surfer	0,0000	0,0142	0,0823	0,0730	0,5146	0,2772
Suv	0,4820	0,3803	0,4664	0,5836	0,3190	0,3024
Sylvester	0,4055	0,3579	0,0034	0,1148	0,3214	0,4965
Tiger1	0,0716	0,0350	0,0062	0,0053	0,4779	0,5244
Tiger2	0,0880	0,0092	0,0003	0,0047	0,2675	0,2706

(Continued)

Table 6. Continued

Video Names	MOSSE	KCF	YOLO-Lite	YOLOv3	GOTURN	WiSARD
Toy	0,0862	0,0379	0,0131	0,1023	0,4740	0,4921
Trans	0,5447	0,5044	0,0337	0,2262	0,4764	0,3980
Trellis	0,5586	0,3120	0,1017	0,1248	0,7051	0,5751
Twinnings	0,0000	0,2101	0,0290	0,5320	0,1962	0,4912
Vase	0,0000	0,0978	0,0342	0,1082	0,5702	0,5653
Walking	0,0017	0,0236	0,1269	0,7262	0,6816	0,6760
Walking2	0,0000	0,2712	0,3695	0,6445	0,2825	0,3758
Woman	0,0000	0,0412	0,2835	0,5382	0,1013	0,1265

REFERENCES

[1] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. 2016. Youtube-8m: A large-scale video classification benchmark. *Arxiv Preprint Arxiv:1609.08675* (2016).

[2] I. Aleksander, M. De Gregorio, F. Maia Galvão França, P. Machado Vieira Lima, and H. Morton. 2009. A brief introduction to weightless neural systems. In *Proceedings of the 17th European Symposium on Artificial Neural Networks*. Retrieved from https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2009-6.pdf.

[3] I. Aleksander, W. V. Thomas, and P. A. Bowden. 1984. WISARD·a radical step forward in image recognition. *Sensor Rev.* 4, 3 (1984), 120–124. DOI: https://doi.org/10.1108/eb007637

[4] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.* 87 (2020), 101374.

[5] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, Peter Vajgel et al. 2010. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, Vol. 10. 1–8.

[6] D. S. Bolme. 2010. Visual object tracking using adaptive correlation filters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[7] Ali Borji, Ming-Ming Cheng, Qibin Hou, Huaizu Jiang, and Jia Li. 2019. Salient object detection: A survey. *Comput. Vis. Media* (2019), 1–34.

[8] Neil Briscoe. 2000. Understanding the OSI 7-layer model. *PC Netw. Advis.* 120, 2 (2000).

[9] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. 2004. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.* 3, 4 (2004), 837–863.

[10] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. 2013. Active disk meets flash: A case for intelligent SSDs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ACM, 91–102.

[11] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. 2009. A survey of flash translation layer. *J. Syst. Archit.* 55, 5–6 (2009), 332–343.

[12] Michael Cornwell. 2012. Anatomy of a solid-state drive.*Commun. ACM* 55, 12 (2012), 59–63.

[13] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry-standard API for shared-memory programming. *Comput. Sci. Eng.*1 (1998), 46–55.

[14] Trevor Darrell, Piotr Indyk, and Gregory Shakhnarovich. 2005. *Nearest-neighbor Methods in Learning and Vision: Theory and Practice*. The MIT Press.

[15] 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 ACM SIGMOD International Conference on Management of Data*. ACM, 1221–1230.

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

[17] Daniel N. Do Nascimiento, Rafael Lima De Carvalho, Felix Mora-Camino, Priscila V. M. Lima, and Felipe M. G. Franca. 2015. A WiSARD-based multi-term memory framework for online tracking of objects. In *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*. 978–287587014–8. DOI: https://doi.org/10.13140/RG.2.1.3387.5687

[18] Carlotta Domeniconi, Jing Peng, and Dimitrios Gunopulos. 2002. Locally adaptive metric nearest-neighbor classification. *IEEE Trans. Pattern Anal. Mach. Intell.* 24, 9 (2002), 1281–1285.

[19] Eideticom. 2020. Retrieved from https://www.eideticom.com/.
[20] K. Eshghi and Rino Micheloni. 2013. SSD architecture and PCI express interface. In *Inside Solid State Drives (SSDs)*. Springer, 19–45.
[21] A. E. Eshratifar, M. S. Abrishami, and M. Pedram. 2019. JointDNN: An efficient training and inference engine for intelligent mobile cloud computing services. *IEEE Trans. Mob. Comput.* (2019), 1–1.
[22] Mark Fasheh. 2006. OCFS2: The Oracle Clustered File System, version 2. In *Proceedings of the Linux Symposium*, Vol. 1. 289–302.
[23] Horacio L. França, João Carlos P. da Silva, Omar Lengerke, Max Suell Dutra, Massimo De Gregorio, and Felipe Maia Galvão França. 2010. Movement persuit control of an offshore automated platform via a RAM-based neural network. In *Proceedings of the 11th International Conference on Control Automation Robotics & Vision*. 2437–2441.
[24] John Gantz and David Reinsel. 2012. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the Far East. *IDC iView: IDC Analyze the future* 2007, 2012 (2012), 1–16.
[25] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. 2006. Open MPI: A flexible high performance MPI. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Waśniewski (Eds.). Springer Berlin, 228–239.
[26] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.* 22, 6 (1996), 789–828.
[27] SSD Form Factor Working Group. 2011. Retrieved from http://www.ssdformfactor.org/docs/SSD_Form_Factor_Version1_00.pdf.
[28] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho et al. 2016. Biscuit: A framework for near-data processing of big data workloads. In *ACM SIGARCH Comput. Archit. News*, Vol. 44. IEEE Press, 153–165.
[29] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. *DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings*. Vol. 44. ACM.
[30] Trevor Hastie and Robert Tibshirani. 1996. Discriminant adaptive nearest neighbor classification and regression. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 409–415.
[31] David Held, Sebastian Thrun, and Silvio Savarese. 2016. Learning to track at 100 fps with deep regression networks. In *Proceedings of the European Conference on Computer Vision*. Springer, 749–765.
[32] João F. Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. 2014. High-speed tracking with kernelized correlation filters. *IEEE Trans. Pattern Anal. Mach. Intell.* 37, 3 (2014), 583–596.
[33] Ali HeydariGorji, Siavash Rezaei, Mahdi Torabzadehkashi, Hossein Bobarshad, Vladimir Alves, and Pai H. Chou. 2020. HyperTune: Dynamic hyperparameter tuning for efficient distribution of DNN training over heterogeneous systems. *Arxiv Preprint Arxiv:2007.08077* (2020).
[34] Ali HeydariGorji, Mahdi Torabzadehkashi, Siavash Rezaei, Hossein Bobarshad, Vladimir Alves, and Pai H. Chou. 2020. STANNIS: Low-Power Acceleration of Deep Neural Network Training Using Computational Storage Devices. Retrieved from arxiv:cs.DC/2002.07215.
[35] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q . Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4700–4708.
[36] Jeff Janukowicz. October 2018. *How New QLC SSDs Will Change the Storage Landscape*. Technical Report. Micron. Retrieved from https://www.micron.com/-/media/client/global/documents/products/white-paper/how_new_qlc_ssds_will_change_the_storage_landscape.pdf?la=en.
[37] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2011. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128.
[38] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. *Arxiv Preprint Arxiv:1408.5093* (2014).
[39] 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. *Proc. VLDB Endow.* 9, 12 (2016), 924–935.
[40] 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 *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, 1–13. DOI: https://doi.org/10.1145/2749469.2750412
[41] Yangwook Kang, Yang-suk Kee, Ethan L. Miller, and Chanik Park. 2013. Enabling cost-effective data processing with smart SSD. In *Proceedings of the IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST'13)*. IEEE, 1–12.
[42] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. 2016. In-storage processing of database scans and joins. *Inf. Sci.* 327 (2016), 183–200.

[43] Gunjae Koo, Kiran Kumar Matam, H. V. Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, Murali Annavaram et al. 2017. Summarizer: Trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 219–231.

[44] Matej Kristan, Jiri Matas, Aleš Leonardis, Tomas Vojir, Roman Pflugfelder, Gustavo Fernandez, Georg Nebehay, Fatih Porikli, and Luka Čehovin. 2016. A novel performance evaluation methodology for single-target trackers. *IEEE Trans. Pattern Anal. Mach. Intell.* 38, 11 (Nov. 2016), 2137–2155. DOI: https://doi.org/10.1109/TPAMI.2016.2516982

[45] Bo Li, Junjie Yan, Wei Wu, Zheng Zhu, and Xiaolin Hu. 2018. High performance visual tracking with siamese region proposal network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 8971–8980.

[46] Rafael Lima De Carvalho, Danilo S. C. Carvalho, Felix Mora-Camino, Priscila V. M. Lima, and Felipe M. G. França. 2014. Online tracking of multiple objects using WiSARD. In *Proceedings of the 22nd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*. 541–546. Retrieved from https://hal-enac.archives-ouvertes.fr/hal-01059678.

[47] David G. Lowe. 2004. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vis.* 60, 2 (2004), 91–110.

[48] David J. C. MacKay and Radford M. Neal. 1996. Near Shannon limit performance of low density parity check codes. *Electron. Lett.* 32, 18 (1996), 1645–1646.

[49] Pankaj Mehra. 2019. Samsung smartSSD: Accelerating data-rich applications. In *Proceedings of the Flash Memory Summit*.

[50] Dirk Merkel. 2014. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J.* 2014, 239 (2014), 2.

[51] Hyeonseob Nam and Bohyung Han. 2016. Learning multi-domain convolutional neural networks for visual tracking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4293–4302.

[52] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. 2016. SSD failures in datacenters: What? when? and why?. In *Proceedings of the 9th ACM International on Systems and Storage Conference*. ACM.

[53] Rishiyur S. Nikhil. 2009. What is bluespec? *ACM SIGDA Newslett.* 39, 1 (2009), 1–1.

[54] Guanghan Ning, Zhi Zhang, Chen Huang, Xiaobo Ren, Haohong Wang, Canhui Cai, and Zhihai He. 2017. Spatially supervised recurrent convolutional neural networks for visual object tracking. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'17)*. IEEE, 1–4.

[55] Shigeo Ohshima and Yoichiro Tanaka. 2016. New 3D flash technologies offer both low cost and low power solutions. In *Proceedings of the Flash Memory Summit*.

[56] ONFI online. 2017. Open NAND Flash interface specification. Retrieved from http://www.onfi.org/specifications.

[57] PCI-SIG. 2020. Retrieved from https://pcisig.com/specifications/pciexpress/M.2_Specification/.

[58] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 779–788.

[59] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *Arxiv Preprint Arxiv:1804.02767* (2018).

[60] IDC Report. 2018. The digitization of the world from edge to core. Retrieved from https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

[61] Siavash Rezaei, Eli Bozorgzadeh, and Kanghee Kim. 2019. UltraShare: FPGA-based dynamic accelerator sharing and allocation. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig'19)*. IEEE, 1–5.

[62] S. Rezaei, C. Hernandez-Calderon, S. Mirzamohammadi, E. Bozorgzadeh, A. Veidenbaum, A. Nicolau, and M. J. Prather. 2016. Data-rate-aware FPGA-based acceleration framework for streaming applications. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig'16)*. 1–6. DOI: https://doi.org/10.1109/ReConFig.2016.7857162

[63] Siavash Rezaei, Kanghee Kim, and Eli Bozorgzadeh. 2018. Scalable multi-queue data transfer scheme for FPGA-based multi-accelerators. In *Proceedings of the IEEE 36th International Conference on Computer Design (ICCD'18)*. 374–380.

[64] Steve Roddy. 2019. Arm NN: the Easy Way to Deploy Edge ML. Retrieved from https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/arm-nn-the-easy-way-to-deploy-edge-ml?_ga=2.9822706.6940669.1579038789-277442185.1570226249.

[65] Mohammad Samragh, Mojan Javaheripi, and Farinaz Koushanfar. 2019. CodeX: Bit-flexible encoding for streaming-based FPGA acceleration of DNNs. *CoRR* abs/1901.05582 (2019).

[66] Scaleflux. 2020. Retrieved from http://scaleflux.com/index.html.

[67] Sudharsan Seshadri, Mark Gahagan, Meenakshi Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A user-programmable SSD. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 67–80.

[68] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop distributed file system. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. IEEE, 1–10.

[69] Josef Sivic and Andrew Zisserman. 2003. Video Google: A text retrieval approach to object matching in videos. In *Proceedings of the International Conference on Computer Vision*, Vol. 2. IEEE, 1470–1477.
[70] SNIA. 2019. Computational Storage Technical Working Group. Retrieved from https://www.snia.org/computational.
[71] SolarWinds. 2018. Can gzip Compression Really Improve Web Performance? Retrieved from https://royal.pingdom.com/can-gzip-compression-really-improve-web-performance/.
[72] Steven R. Soltis, G. M. Erickson, Kenneth W. Preslan, Matthew T. O'Keefe, and Thomas M. Ruwart. 1997. The global file system: A file system for shared disk storage. *IEEE Transactions on Parallel and Distributed Systems* 1 (1997), 1.
[73] Mohan Srinivasan. 2014. Flashcache. Retrieved from https://github.com/facebookarchive/flashcache.
[74] NGD Systems. 2020. Retrieved from https://www.ngdsystems.com.
[75] Ted Friedman, Thomas Bittman, Neil MacDonald. 2019. How to Overcome Four Major Challenges in Edge Computing. Retrieved from https://www.gartner.com/doc/reprints?id=1-1XWDQ2PW&ct=191210&st=sb.
[76] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. 2013. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 119–132.
[77] Mahdi Torabzadehkashi, Ali Heydarigorji, Siavash Rezaei, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. 2019. Accelerating HPC applications using computational storage devices. In *Proceedings of the IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS'19)*. IEEE, 1878–1885.
[78] Mahdi Torabzadehkashi, Siavash Rezaei, Vladimir Alves, and Nader Bagherzadeh. 2018. CompStor: An in-storage computation platform for scalable distributed processing. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'18)*. IEEE, 1260–1267.
[79] 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 *Proceedings of the 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'19)*. IEEE, 430–437.
[80] Mahdi Torabzadehkashi, Siavash Rezaei, Ali HeydariGorji, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. 2019. Computational storage: An efficient and scalable platform for big data and HPC applications. *J. Big Data* 6, 1 (2019), 100.
[81] Jack Valmadre, Luca Bertinetto, João Henriques, Andrea Vedaldi, and Philip H. S. Torr. 2017. End-to-end representation learning for correlation filter based tracking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2805–2813.
[82] DRAM Exchange Website. 2020. DRAM Exchange. Retrieved from https://www.dramexchange.com.
[83] Part Stock Website. 2020. Part Stock Specs for Xilinx XCZU19EG-2FFVC1760E. Retrieved from http://www.part-stock.com/product-part/xilinx__XCZU19EG-2FFVC1760E.html.
[84] Kilian Q. Weinberger and Lawrence K. Saul. 2009. Distance metric learning for large margin nearest neighbor classification. *J. Mach. Learn. Res.* 10(Feb. 2009), 207–244.
[85] Yi Wu, Jongwoo Lim, and Ming-Hsuan Yang. 2015. Object tracking benchmark. *IEEE Trans. Pattern Anal. Mach. Intell.* 37, 9 (2015), 1834–1848.
[86] Ming-Chang Yang, Yu-Ming Chang, Che-Wei Tsao, Po-Chun Huang, Yuan-Hao Chang, and Tei-Wei Kuo. 2014. Garbage collection and wear leveling for flash memory: Past and future. In *Proceedings of the International Conference on Smart Computing*. IEEE, 66–73.
[87] Kai Zhao, Wenzhe Zhao, Hongbin Sun, Xiaodong Zhang, Nanning Zheng, and Tong Zhang. 2013. LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. 243–256.

Received February 2020; revised July 2020; accepted August 2020