I/O and Energy Efficient Large-scale Image Similarity Search Using Computational Storage Devices

Anonymous Author(s)*

ABSTRACT

Finding an optimum method for multimedia similarity search for high performance and low energy consumption is one of the critical issues in big-data processing. To address the problems of scalability and memory bottleneck, this paper proposes leveraging in-storage processing on computational storage devices (CSDs), allowing high, scalable data processing by meeting fast I/O and low storage cost requirements of data-intensive applications. As an in-storage processing SSD, Newport CSD is designed and implemented, providing high-level programmability with a general-purpose OS. To efficiently deal with a large dataset not fitting into volatile memory, a distributed batch similarity search technique is introduced where the dataset is divided into smaller batches, and multiple CSDs perform similarity search in parallel using an MPI framework. The effectiveness of the proposed approach was demonstrated with the one billion image dataset. Experimental results show an approximate improvement of 5.8X in query throughput and 2.6X in energy consumption.

1 INTRODUCTION

In the age of Big Data, the huge volume of data produced every day is posing a significant challenge of extracting useful information and knowledge from raw data in an efficient way. In fact, a study in 2012 revealed that only 0.5% of all data is ever analyzed due to a lack of effective methods to manage the data [1]. The problem is especially prominent in the handling of visual data such as images and videos, as companies such as Google and Facebook are experiencing unprecedented growths in the production of photos and videos [2, 3]. It is clear that efficiently indexing, retrieving, and interpreting visual contents are very important tasks, and developing more scalable and robust methods to deal with a large quantity of visual data is required.

In the analysis of large volume of visual data, given an image as a query, a crucial computation primitive is to search for images and video frames that are similar to the query. A common technique is to employ various machine learning techniques to transform images into high-dimensional, real-valued vectors so that similarity can be evaluated as the distance between the vectors. Such feature vectors, called embeddings are essentially dense representations of input images, which can be obtained from pre-trained image-classification networks such as DenseNet [4] as shown in Figure 1.a.

Because the networks produce similar vectors for similar images, the distances (e.g., Euclidean) between the vectors can be measured to quantify how semantically similar the corresponding images are. Thus, given an image, the problem of finding k most similar images can be solved by the K-Nearest Neighbors (KNN) algorithm in the vector space. Unlike computation-intensive operations such as feature extraction, calculating distances between feature vectors in KNN are relatively simple and can be efficiently performed without high-end CPUs or GPUs. On the other hand, moving many vectors from storage to processing elements is a significant bottleneck (Figure 1.b), as the bandwidth of conventional storage systems cannot keep up with the amount of many parallelizable distance calculations. To decrease the data movement cost, the vectors can be encoded (e.g., Product Quantization (PQ) [5]) into a smaller number of bits, but this data reduction comes with the cost of loss of accuracy.

KNN over a large number of vectors is an ideal candidate for in-storage processing (Figure 1.c), because it is memory-intensive and computationally simple. The amount of data transferred from storage to host is significantly reduced by returning only the result of a KNN query, which is a set of k identifiers of the nearest images thereby reducing the amount of I/O in the system. The reduced amount of I/O and the scalability of in-storage processing lead to the high performance of KNN, while utilizing the potentially dormant computation resources in SSDs. Also, the energy cost of moving the data across the memory hierarchy is known to be significant [6], and the in-storage processing on programmable SSDs improves the energy consumption of KNN by minimizing the movement of data.

Previous works on in-storage processing have addressed various problems and challenges to advance the state-of-the-art, but most of them lacked sufficiently dedicated in-SSD resources or the software framework 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 device limited the capability and flexibility to perform large-scale, complex applications in many of these systems.

This paper describes Newport, a computational storage device (**CSD**) [7] that provides high-level programmability through a complete standard network software and protocol stack. 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®), allowing general application developers to fully leverage existing tools and libraries to minimize the effort to create and maintain applications running in-storage. This device can be configured to be used as a simple solid state drive (*Newport SSD*) or a full blown computational storage device capable of processing data in-situ (*Newport CSD*).

Our contributions in this paper are summarized as follows:

- We discuss the design and implementation of Newport computational storage device,
- We introduce Batch Similarity Search to deal with large datasets that do not fit in memory,
- We demonstrate the improved performance and energy efficiency of in-storage processing for image similarity search with multiple computational storage devices.

The rest of this paper is organized as follows. In Section 2, we explain the prerequisite background of this paper. We discuss the

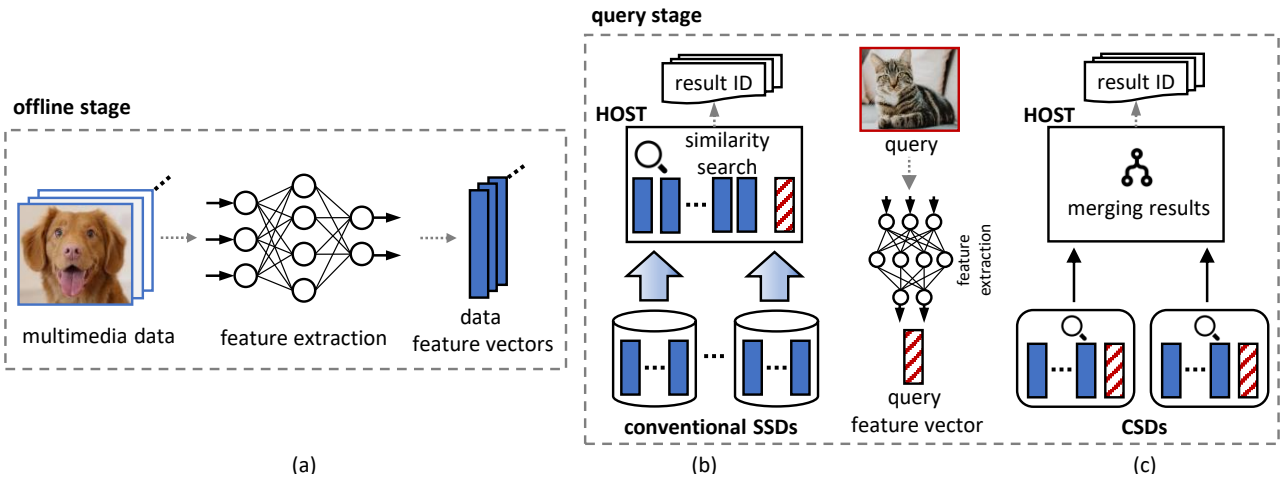

Figure 1: Similarity search pipeline: (a) extracting feature vectors from dataset is performed offline. At the query stage, the image search consisting of many highly parallelizable distance calculations can be done (b) on host after moving the vectors from conventional SSDs, or (c) inside computational storage devices.

design and implementation of Newport CSD in Section 3, and introduce Batch Similarity Search in Section 4. In Section 5, we performed experiments for large-scale image similarity search. Related works are discussed in Section 6, and conclusions are provided in Section 7.

2 BACKGROUND

2.1 Flash Solid State Drives (SSDs)

There are two main components in any solid state drive: the controller and the non-volatile storage media - most commonly NAND flash [8]. NAND flash must be erased before data is written and memory can endure only a limited number of erases before it can no longer be used. The controller implements various functions to effective manage NAND flash under these limitations, such as garbage collection that reclaims blocks containing invalid data and wear leveling that ensures that flash blocks are used evenly to prolong SSD life. A Flash Translation Layer (FTL) maps logical addresses to physical addresses so that 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 channel is a bus used for communication between the SSD controller and a subset (a column) of flash chips, and 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.

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 [9]. NVM Express (NVMe) is the most recent standard for 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 magnetic storage (i.e. hard disk drives), but 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 the application or use case - client, enterprise, cloud, mobile, etc.

2.2 Programmable SSDs

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 [10–16] 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 programmable 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.

2.3 Similarity Search

Despite the research on KNN for the last several decades, it is commonly believed that finding exact nearest neighbors in high-dimensional datasets is very difficult [17], and researchers have started focusing on approximate nearest neighbor (ANN) search where close neighbors are found without the strict guarantee of being closest [18–23].

FAISS [1] is an open source library from Facebook Research for efficient similarity search of vectors. It supports a wide range of methods for indexing and searching similar vectors, from brute-force exact distance calculations to aggressive ANNs that trades off accuracy for performance. We use the type of index from FAISS called IVF-Flat [18, 24] that combines an inverted file index (IVF) with exact Euclidean distance calculations. IVF-Flat utilizes a coarse quantizer that clusters vectors around smaller number of centroids. When a query vector is given to search for k nearest neighbors, the coarse quantizer is used to find the index of the closest Voronoi cell, and the residual vector of query is compared to all the vectors in the cell to find k closest distances. The inverted file indexing significantly improves the performance of KNN as the number of distance calculations are significantly reduced, but it loses accuracy when the closest vectors belong to neighboring cells. Increasing the number of cells or searching multiple cells during a query may improve accuracy at the cost of performance. The methods proposed in this paper are not restricted to IVF-Flat, and can be applied to any other types of index.

3 NEWPORT CSD

In this work a new type of programmable SSD is used in order to overcome the limitations of conventional SSDs with regard to in-storage processing. Newport CSD is a computational storage device (Figure 2) that provides dedicated computing resources and a complete software stack for near-data execution of user applications.

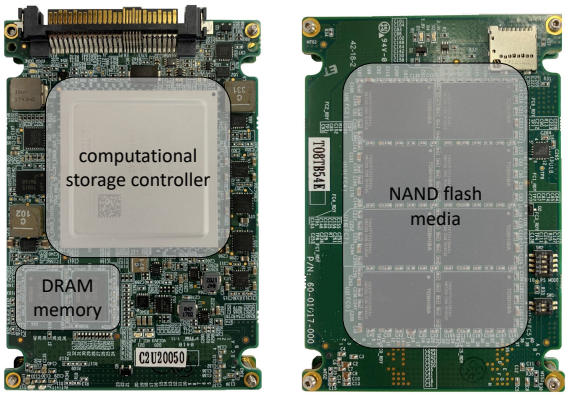

Figure 2: Newport CSD in 2.5-inch form factor.

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

3.1 Hardware Architecture

Fig. 3 shows the architecture of the Newport computational storage device. The Newport CSD is built around a custom ASIC designed to replace the previous prototype CSDs based on the Zynq UltraScale+® FPGA device from Xilinx, which is widely used in the design of wireless infrastructure, cloud computing, and defense applications [25]. The custom ASIC integrates a processing subsystem with a feature-rich 64-bit quad-core ARM® Cortex-A53 (with the companion ARM® Neon SIMD engines, 1MB L2 cache, and 8GB DDR4), running up to 1.3GHz that are dedicated to performing data processing. It also includes Cortex-M7 real-time embedded processors exclusively used to implement the SSD storage functionality. The ASIC also includes a large amount of logic used for implementing the custom logic, including the NAND Flash media controller with 16 flash channels, the error correction engine (ECC), and the NVMe protocol hardware acceleration engine among others. The host interface is a 4-lane PCIe Gen3 bus with an approximate raw bandwidth of 4GB/s.

In order to provide superior performance, the CSD controller includes embedded hardware accelerators as key components of the CSD to accelerate three critical functions: Search, Pattern Matching and Sorting. Thanks to dedicated hardware engines, these functions can be performed at an order of magnitude higher throughputs compared to a generic x86 CPU [20]. The regular expression acceleration engine can perform pattern matching on user data sitting in DRAM as well as data stored on Flash media in forms of LBA ranges using pipelined pattern matching algorithms. Streaming user data directly from flash media to pattern matching engine eliminates the software layers conventionally involved in software-based pattern matching. A common way of executing regular expression searches is to first compile the expression itself to a Deterministic Finite Automata (DFA). A possible prototype implementation could use RE2 (Cox), a C++ package that parses regular expressions and builds the correspondent DFA database. The hardware accelerator comes with a standard RegEx software API, which is very similar to conventional hardware-based RegEx API [21].

The proposed CSD architecture also includes an encryption engine supporting AES-XTS with 256-bit keys and 2- AES-GCM (using AES-CTR mode and GHASH) with 256-bit keys. The encryption engine is inserted in the data path of the CSD controller, and it is used for two main purposes in the Newport design as follows: a. Real-time encryption/decryption of data at rest as it is written/read to/from the Flash media (self-encryption drive) b. Real-time encryption and authentication of data as it is streamed out of the storage device (corresponding to a host CPU read operation) for secure data transmission from the source, i.e., the storage device containing the data.

From a system-level perspective, Newport CSD architecture can be shown as Fig. 4. The Newport CSD hardware components can be categorized into three subsystems, namely, SSD front-end, SSD back-end, memory interface controller, and the in-storage processing subsystem. The front-end subsystem includes the host PCIe/NVMe interface, and one of the M7 cores, while the back-end is composed of three M7 cores and the ECC unit (see Fig. 3). The front-end subsystem is responsible for communicating with the host to receive the I/O commands and check the integrity of the commands. On the

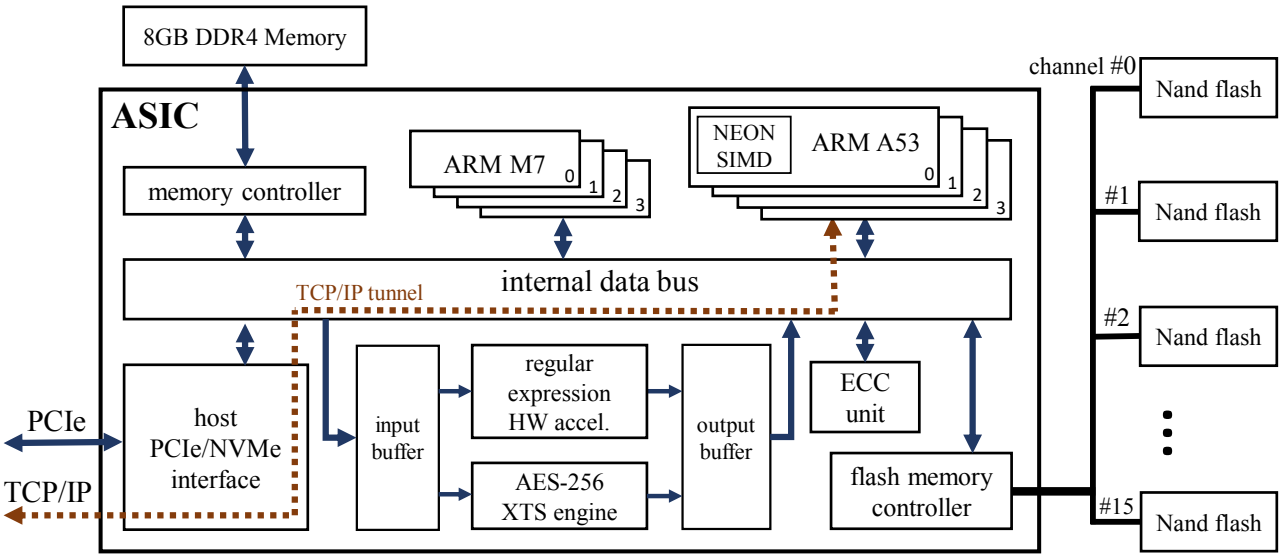

Figure 3: Hardware Architecture of Newport CSD

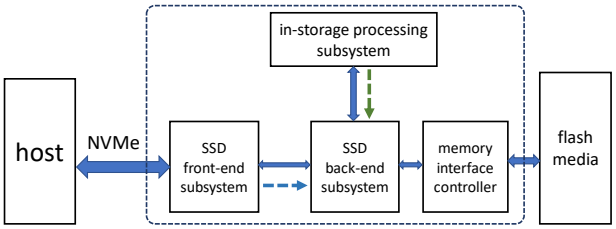

Figure 4: Newport Internal Dataflow.

other hand, the back-end subsystem manages the Flash memory modules and handles the low-level Flash I/O operations via the memory interface controller.

Besides these three subsystems, Newport CSD is equipped with an in-storage processing subsystem, which is composed of the quad-core ARM Cortex-A53 processor, Neon SIMD engines, regular expression HW accelerator, and AES encryption engine. This subsystem resides inside the CSD controller, and it has a high-speed and low-power access to the data stored in the Flash media. The back-end subsystem serves both the front-end and in-storage processing subsystems. In Fig. 4, the dataflow from the in-storage processing and front-end processing subsystems to the back-end subsystem are shown with two arrows.

3.2 Software Stack

A seamless programming model can be achieved using a complete software stack, starting with a 64-bit operating system running on top of the ARM® processing subsystem described in Section 3.1. Compared with the embedded software development process that is complex and makes programming and debugging very challenging, Newport CSD enables application developers to fully leverage existing tools, libraries, and expertise instead of low-level/bare-metal software running on embedded real-time processors. Currently, Newport CSDs run a variety of Linux operating systems including Ubuntu® 16.04.5 LTS. In this paper we use Newport CSD to investigate the significance of in-storage processing especially in the area of image similarity search, but it can be leveraged for other data-intensive applications as well.

Various software stack layers comparing with the standard Open Systems Interconnection (OSI) model are shown in Figure 5. For the sake of simplicity, the figure shows only one Newport CSD connected to a host machine. In order to support distributed and parallel image similarity search with multiple Newport CSDs, we built a session for the high-performance, distributed-memory communication with Message Passing Interface (MPI) [26].

As illustrated in Figure 5, both the host machine and Newport CSD can access data stored on NAND flash memory. From the host side, data access (Figure 5.a) 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. From the Newport CSD side, data access (Figure 5.b) is provided through a custom block device driver that is added as a module to the kernel of the operating system of the device. The block device driver communicates directly with the CSD'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.

A network-based connection is one of the main requirements of most of distributed processing frameworks like MPI and Hadoop.

For data communication between host and each Newport CSD to support the in-storage processing, we implemented a network-based connection using TCP/IP tunneling over NVMe/PCIe (Figure 5.c.) In this approach, the TCP/IP network packets are encapsulated into NVMe packets to be transferred to/from the device. One shared buffer is used for each data transfer direction. Finally, the software stack supports clustered and distributed file systems, e.g. OCFS2,

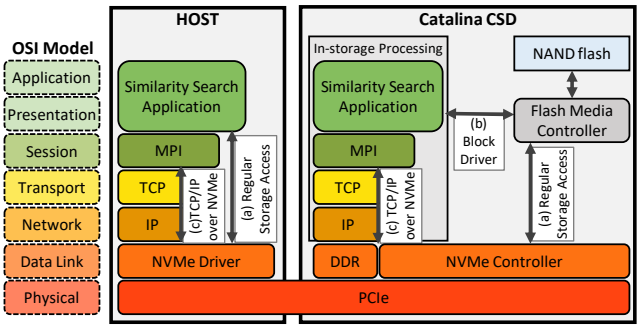

Figure 5: Software stacks on Host and Newport CSD.

[27] such that the Newport device can be used as a conventional SSD and an in-storage processing device at the same time, which is not possible with previous programmable SSDs.

4 SIMILARITY SEARCH ALGORITHM

Once vectors for all dataset images are obtained through the offline stage in the similarity search pipeline (Figure 1.a), the next step is to conduct the query stage to find k "closest" vectors[2] in a multidimensional vector space. As the simplest way of getting the top k closest vectors, we can use a brute-force approach that computes all the vector distances - exactly and exhaustively - between the query vector and each of the other vectors in the dataset. However, the difficulty of calculation of the vector distance, 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, as described in Section 2.3, multi-dimensional space is partitioned to multiple cells, and it is presented as an inverted file index (IVF) [24] 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 a 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 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:

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

This brute-force approach, however, could be very expensive due to a large numbers of distance calculations among vectors. In contrast, the IVF-Flat index considered in this paper 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.

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

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

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 $p/|C|$.

4.2 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 data sets 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 for more details).

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 that 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 Equation 2, 3 and 4 are re-written as:

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

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

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

At the query 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 data set is shown in Algorithm 1.

As shown in the algorithm, we load each IVF-Flat index, and search k nearest neighbors from the index (Equation 7) serially.

[2]There is a number of 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 paper, we focus on the Euclidean metric.

Algorithm 1: Batch Similarity Search

Input: (x, I=[I_1,...,I_b])
Output: $L_{IVF-FLAT}$
Function BBS(x, I)**:**
  $L_{IVF-FLAT} = \{\}$
  foreach $I_j \in [I_1, ..., I_b]$ **do**
    Load(I_j)
    L_j = **Search**(x, I_j)
    $L_{IVF-FLAT} = L_{IVF-FLAT} \cup L_j$;
  end
  return k-MinDistSort($L_{IVF-FLAT}$);
End Function

At the end, $b \times k$ vector indices are gathered in $L_{IVF-FLAT}$. Note that to return the final k nearest neighbors of the query vector, the k-MinDistSort function is used to sort vector indices based on the calculated distances between x and $y_i (i \in L_{IVF-FLAT})$.

4.3 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 peudo-code of Distributed BSS when there are s distributed nodes in the distributed environment.

Algorithm 2: Distributed Batch Similarity Search

Input: (x, I=[I_1,...,I_b])
Output: $L_{IVF-Flat}$
Function DistributedBBS(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

As shown in the algorithm, the Partition function returns a subset of IVF-Flat indexes that is processed by the m^{th} node (See Section 5.5 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.

5 EXPERIMENTS

In this section, we present experimental results to describe potential benefits of using computational storage devices for an image similarity search application.

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

5.1 Experimental Setup

Hardware: We conducted all the experiments to evaluate the performance and energy consumption of large-scale image similarity search on a high-capacity in-storage processing platform shown in Figure 6. The platform is equipped with an Intel Xeon Processor E5-2630 v4 running at 2.20GHz (supporting 20 threads with 10 cores), and 80GB of DRAM memory. This platform's PCIe bus supports up to 24 4-lane Newport CSDs, each of which with up to 20 TB flash capacity and connected with the host through the TCP/IP tunneling over NVMe/PCIe as described in Section 3. Each Newport CSD has four dedicated ARM® Cortex-A53 cores running at 1.5GHz used as compute resources. Both the host server and the computational storage devices run a 64-bit Ubuntu® 16.04 LTS operating system.

The power consumption of the platform is measured with the Wattman HPM-100A power meter [3] with an accuracy of 0.4%. The device's power measurement range is from 0.009W to 3,600W, and it logs data in one second intervals.

Software: To evaluate how computational storage devices augment the performance of image similarity search on large datasets, we chose 1.2.1 version of Faiss[4], a widely-used similarity search library of dense vectors. Faiss is developed for 64-bit x86 platforms, and has pure C++ code optimized for Intel SSE/AVX instructions. We made code changes to make Faiss portable to the ARM® architecture, and optimized its operations for the ARM® SIMD operations as well. For the MPI implementation mentioned in Section 3.2, we used Open-MPI[5], an open-source MPI library that is widely used in the academia and the industry.

We carried out all the experiments for performance and energy efficiency comparative analysis on a large-scale dataset: ANN_SIFT1B[6] that contains 1 billion SIFT [29] image vectors of 128 dimensions extracted from public images. We first used a subset of 50 million vectors of the base set to investigate various aspects of in-storage processing performance in Section 5.3 - 5.5, and then used the full base set of 1 billion vectors to demonstrate the impact on the large-scale search in Section 5.6.

In addition to the base set of 1 billion vectors, ANN_SIFT1B consists of a learning set of 100 million vectors, and a query set of 10 thousand vectors. During the offline stage of the similarity search pipeline (Figure 1.a), an inverted file (IVF) index is built with

[3] http://shop2.adpower21com.cafe24.com
[4] We note that our proposed approach explored in this paper is not restricted to a specific similarity search method. Other well-known similarity search libraries are found in [28].
[5] https://www.open-mpi.org/
[6] http://corpus-texmex.irisa.fr/

4096 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, we performed Batch Similarity Search (BSS) where the dataset is divided into smaller batches. With 6 GB of usable DRAM of each Newport CSD, the largest size of an IVF-Flat index that can fit into the memory is around 10 million vectors. Therefore, we created an IVF-Flat index per each batch, total 100 IVF-Flat indexes. In the experiments with 50 million vectors, however, we used a smaller batch size of around 3 million vectors, resulting in total 16 batches. Dividing into 16 batches allowed easier distribution of the workload across multiple processing elements, while maintaining sufficient workload for each batch. During the query stage (Figure 1.b), a query vector is compared to all the base-set vectors belonging to candidate cells (out of 4096 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 paper, 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)**

We measured the elapsed time taken to load IVF-Flat indexes and to search nearest neighbors of all query-set vectors to calculate query throughputs (i.e., queries/sec). For optimal performance, we parallelized search loops over the query-set vectors via the OpenMP [30] API that supports shared memory multiprocessing programming.

In the experiments described in this section, three main system configurations were used to analyze the query throughput and energy consumption as shown in Figure 7: (1) BSS is performed on the host platform with Newport SSDs where the in-storage processing feature is disabled (*"host-only"*), (2) on Newport CSDs where the in-storage processing feature is enabled (*"CSD-only"*), and (3) on both the host and Newport CSDs together (*"hybrid"*). Note that in all configurations, a single coordinator task is created to gather search results from many distributed worker tasks through MPI_Gather, one of the collective communication routines defined in the MPI standard specification[7].

5.2 Datapath vs. Control Path Performance

Newport CSD is a platform built around custom integrated circuit. As mentioned in 3.2 there are two communication mechanisms between the host and the computational storage devices: (1) the NVMe device interface specification for accessing non-volatile storage media attached via a PCI Express (PCIe) bus (i.e., the datapath) and (2) a network based connection based on tunneling of TCP/IP packets over the NVMe/PCIe bus (i.e., the control path).

The sequential read performance of the NVMe bus (data path) is approximately 1600MB/s while the sequential write performance reaches 1200MB/s. The TCP/IP tunnel connection (control path) runs at approximately 175MB/s. The performance is measured using widely used tools for network and storage I/O performance measurements: iPerf [8] for the TCP/IP connection and Fio [9] for the NVMe interface. The reduced performance of the TCP/IP connection compared to the NVMe path is explained by two factors: firstly the fact that there are three layers of packetization (PCIe, NVMe and TCP/IP) that add overhead and secondly that there are inefficiencies in copying data from the reserved memory spaces out of the Linux® space (as of shared device_to_host/host_to_device buffers) to the kernel space.

It is worth noting that in the later case we are currently working on driver optimization to eliminate that inefficiency and bring the TCP/IP connection bandwidth within less than 25% of the NVMe bandwidth. Even with these performance limitations we have been able to augment the performance of the similarity search application when in-storage processing is enabled as shown in the remainder of this section.

5.3 Performance of Host-only Processing

Figure 7.a shows the configuration we used to measure the performance of the host-only processing with *N* Newport SSDs (the in-storage processing feature disabled). *N* worker tasks and one coordinator task are created to execute similarity search, and to gather the output of each worker task for the final result. The MPI framework is used to distribute and gather the computations.

Figure 8.a shows the comparison of similarity search throughputs (measured in queries per second), over the subset of 50 million vectors on the host-only configuration using one Newport SSD (i.e., N=1). We examined two cases: (1) using virtual memory and (2) using BSS to handle a large dataset that does not fit into memory. As shown in the figure, with virtual memory the host operating system needs to constantly swap vector data back and forth between memory and the device, resulting in a significant performance drop. In contrast, BSS loads each batch of vectors only once, therefore, no additional I/Os are occurred to load the same vector data.

Figure 8.b shows the host-only processing performance in the BSS mode as varying the number of Newport SSDs. Since each Newport SSD is given its own set of four PCIe lanes, we observed that the I/O bandwidth scales linearly with the number of SSDs. However, the throughput performance does not. A detailed profiling of the application performance shows that while the loading time decreases more SSDs are used, the search time is bottlenecked by the host CPU and memory bandwidths, and as a result, in this specific configuration the performance improvement plateaus at eight Newport SSDs (i.e., N=8).

5.4 Scalable Performance of In-Storage Processing

In this section, we evaluated the scalable performance of the in-storage processing of Newport CSD with the CSD-only configuration illustrated in Figure 7.b. In this configuration, similarity search is performed with energy-efficient compute resources available inside Newport CSDs, and the host only acts as a coordinator that collects the results of each CSD.

As shown in Figure 8.c, we observed that the throughput scalability of the similarity search is almost close to linear as the number

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

[8] https://iperf.fr/

[9] https://github.com/axboe/fio

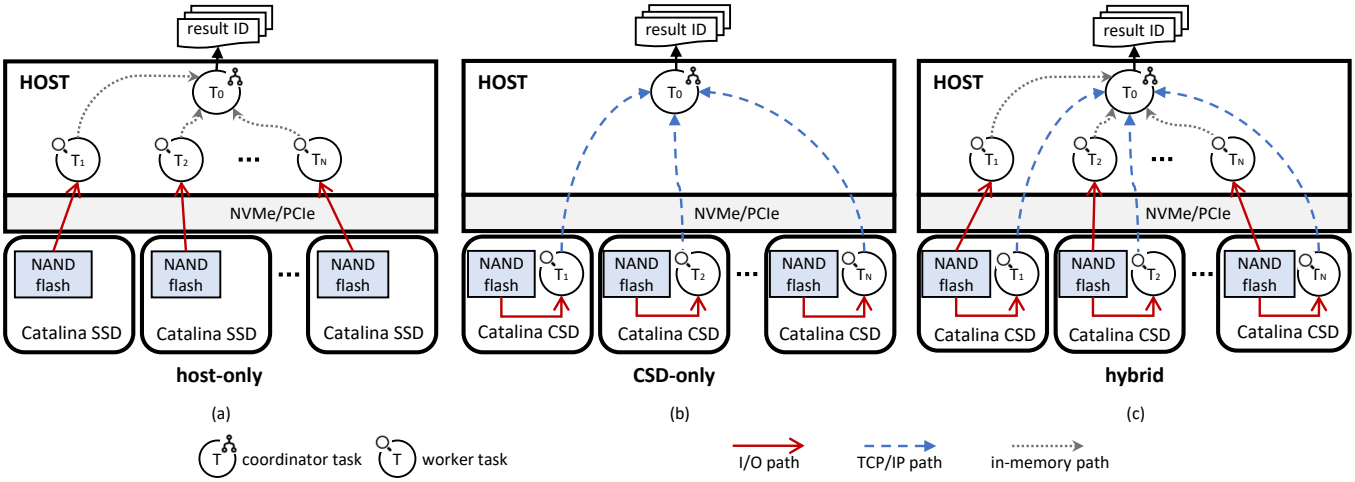

Figure 7: Experimental setup configurations: a) host-only, b) CSD-only and 3) hybrid.

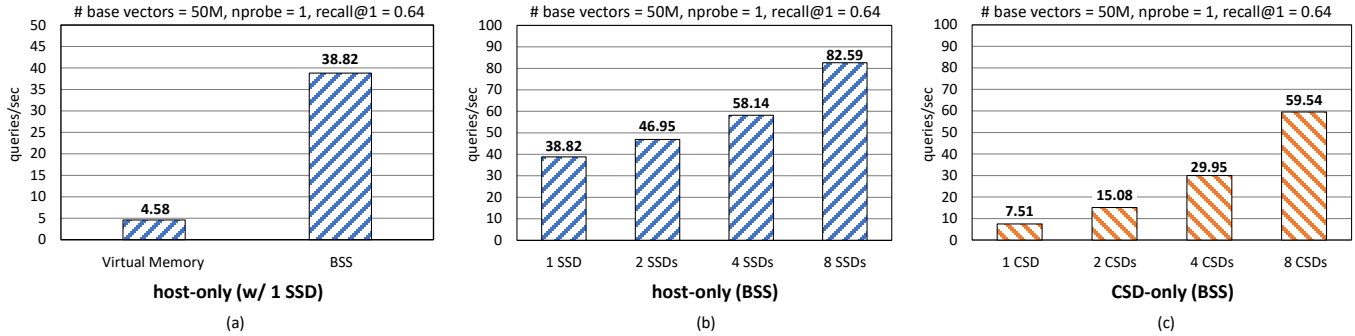

Figure 8: Experimental results: a) virtual memory vs. BSS with a single Newport SSD, b) host-only, and c) CSD-only BSS performance as varying the number of Newport SSDs and CSDs, respectively.

of Newport CSDs increases. The processing approach of Newport CSDs is basically a shared-nothing architecture in which each CSD can operate independently and separated from the other CSDs. Thus a cluster of Newport CSDs enables the processing power to be scaled nearly linearly by adding more devices into the cluster. In addition the scalable processing power, the aggregated data bandwidth in the CSD-only processing also can scale linearly because it is internal within the CSDs in contrast to the host-only processing that depends on physical PCIe lanes (, which reduces the I/O traffic on the PCIe bus and leads to lane sharing opportunities) as described in Section 5.3.

Compared to the host-only processing performance shown in Figure 8.b, it is clear that the combined processing power of multiple Newport CSDs is comparable to the host with a much more powerful processor. For the data-intensive tasks such as similarity search over large datasets, the in-storage processing may handle the workload while the host is freed to perform other computational tasks.

5.5 Distributing Computations between Host and Computational Storage Devices

For large-scale image similarity search, it is important to maximize the search performance by utilizing resources available in both host and CSDs. Figure 7.c shows the scenario where the host and Newport CSDs run similarity search in coordination. In this scenario, one important problem is to partition the search task over multiple powerful (but expensive) processors in the host and energy-efficient processors in Newport CSDs, respectively. A detailed analysis on different partitioning schemes is out of scope of paper. Instead, we chose a simple approach of dividing the base-set vectors in two groups - one is processed by the host and the other by the devices - so that the host and the CSDs perform their assigned search tasks, and complete them almost simultaneously. This approach can be written as an optimization problem 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 $d_{host} \approx 29M$ and $d_{csd} \approx 2.6M$.

Figure 9.a shows the comparison of the hybrid approach against the host-only and CSD-only processing. The results shown in the figure indicate that the in-storage processing on eight Newport CSDs improves the query throughput by 73% in comparison to the host attached to eight SSDs. We repeated the comparison after increasing the nprobe parameter (see Section 5.1) from one to four to search a bigger vector space, and results are shown in Figure 9.b.

As illustrated in the figure, searching more neighboring cells in the vector space led to higher accuracy (i.e., recall@1 = 0.96) while significantly decreased query throughput because more search computations are required to find the nearest neighbors of query vectors. Compared to the host-only processing attached to eight SSDs, the hybrid approach improved the query throughput by 88%, demonstrating that the performance improvement of the proposed approach scales well for increased amount of computations.

5.6 Similarity Search with One Billion Vectors

We performed the large-scale image similarity search with the entire dataset of one billion vectors using the hybrid approach (with 8 and 24 Newport CSDs), and compared the results against the host-only processing as shown in Figure 9.c. The hybrid approach of distributing the workload across the host and eight CSDs improves the query throughput by 71% compared to the host-only processing with eight SSDs. Each Newport CSD adds its own computational power and internal data bandwidth, and therefore the scalability of the proposed approach is not affected by the physical limitations of I/O. It is projected that a system with 24 CSDs performs 5.8 times better than the host with a single SSD. The performance improvement and scalability of in-storage processing may be applied to a wide range of applications with large-scale parallel workloads, especially when the application performance is limited by the I/O bandwidth of a system.

Figure 10 shows the throughput and energy consumption of the proposed configurations. It is clear from the figure that combining in-storage processing of multiple CSDs outperforms the host-only processing, and hence leads to high energy efficiency. The hybrid approach with 8 CSDs outperforms the host-only processing with 8 SSDs, and the energy consumption was reduced by 27% (22.4 Kilojoules). The system with 24 CSDs would consume 35 Kilojoules for the similarity search on one billion vectors, which is only 38% of the energy consumed by the host-only processing with a single SSD. The results show that the high performance as well as the reduced movement of data in the memory hierarchy improve the energy efficiency of the proposed approach.

5.7 Cost Analysis

Compared with a normal SSD based on a conventional, embedded controller, a CSD should contain a more powerful processor to efficiently run a variety of software applications. Interestingly, according to the SSD bill of material (BOM) analysis [31, 32], 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. For example, an enterprise-level 8TB NVMe SSD is priced starting at around $1,600 in the open marketplace, and the spot price of NAND flash TLC 512Gb chips is $0.15 per GB (source: *https://www.dramexchange.com*) at the time of writing this paper[10]. Thus, the cost of flash memory chips in this case is $1200, which is about 75% of the SSD price. With other miscellaneous costs (such as DRAM, miscellaneous components and manufacturing costs) that would account for 20-25% of the SSD price, the SSD controller would account for at most 5% of the SSD price.

6 RELATED WORKS

6.1 Programmable SSD

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. [10] 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. [13] extended the MapReduce framework to process a sequence of operations on distributed data sets. 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. Firstly, 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, Newport 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 [33]. In some of the previous works, storage devices are supplemented with FPGAs for the sake of in-storage processing [34]. Although FPGAs are very power efficient and potentially provide a considerable performance improvement through process parallelism, implementing an FPGA accelerator requires a deep understanding of hardware design and the design process

[10]While the prices of an SSD and a flash chip have wide variability at any time and more over time, the analysis provides a good big picture of the economic feasibility of CSDs.

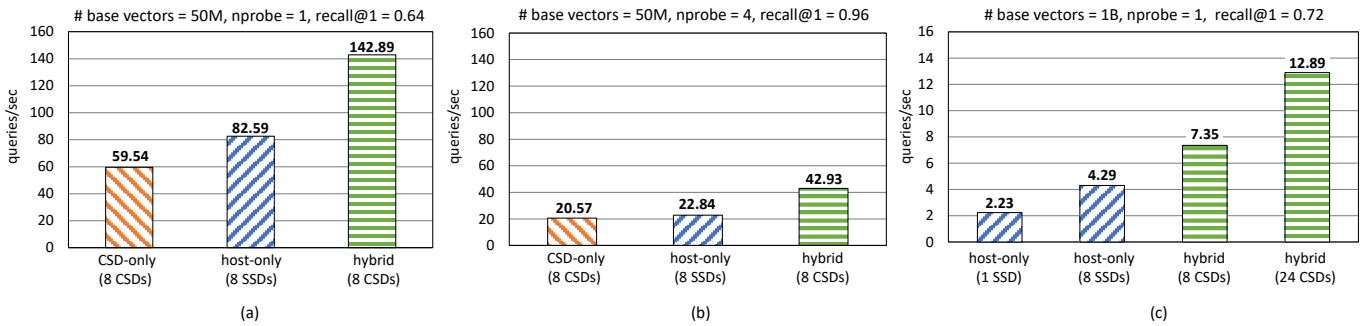

Figure 9: Experimental results: BSS performance comparison with different configurations. a) 50M base vectors (nprobe = 1), b) 50M base vectors (nprobe = 4), and c) 1B base vectors (nprobe = 1).

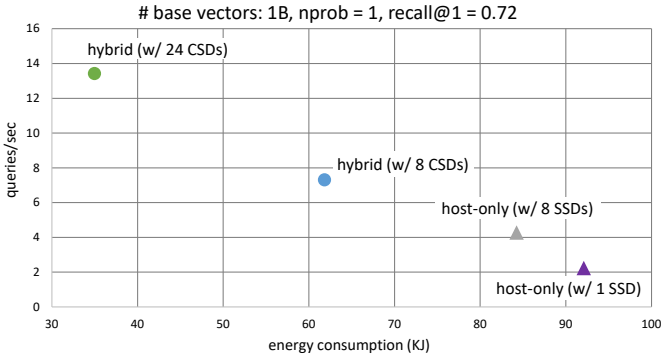

Figure 10: Energy consumption and performance with 1B base vectors.

is very time consuming. On the other hand, 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 [35].

Recently, several studies have studied better programming models for computational storage. In [14], 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. [11] 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 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.

6.2 Large-scale Similarity Search

The proposed approach in this paper is unique that it performs distributed in-storage processing of large-scale similarity search. Large-scale similarity search is an important topic of research with large number of related works, and the most related works are as follows.

Johnson et al. [18] and Feng et al. [36] both have focused on efficient processing of large-scale similarity search on GPU devices. While GPU devices provide a lot of processing power for similarity search, they are ultimately limited by I/O and do not provide the energy efficiency of in-storage processing.

Many research works have proposed efficient methods of performing distributed processing of large-scale similarity search, and MapReduce [37] has been the popular framework for many such works [38–42]. Liu et al. [41] and Chatzimilioudis et al. [42] among them have also used the batch-based approach of handling large data. None of these works have been experimented under the context of in-storage processing, and this paper has a clear contribution compared to these works.

While this paper has used one type of indexing algorithm for KNN, many different algorithms have been proposed such as [18–23, 43–46]. These works complement the proposed approach and investigating them for in-storage processing is a promising direction of research.

7 CONCLUSIONS

This paper proposes improving the performance and energy consumption of large-scale similarity search by performing in-storage processing on CSDs. The proposed approach of in-storage processing is ideal for the large-scale similarity search primarily limited by I/O, because each CSD added to the system has its own internal data bandwidth to support its processing power. The experimental results show that adding more CSDs to a system improves the performance while reducing the energy consumption of the task, and the system with 24 CSDs is projected to have 5.8 times better performance and 2.6 times better energy consumption compared to the conventional host-only processing with a single SSD. It is also found that the BSS method can address the problem of the massive memory consumption of large-scale similarity search, as it improves the query throughput by more than 8 times compared to the method using virtual memory. This work shows that CSDs can be used in large-scale distributed systems as an effective way of augmenting the performance of data-intensive applications, providing excellent scalability and reducing the energy consumption by significantly reducing data movement.

The Newport CSD platform described in this paper is planned to be replaced soon by an ASIC-based platform with significantly

lower power consumption and improved performance. The future work includes experiments using an ASIC based CSD to assess performance impact and use of even larger datasets to test the limits of the scalability of the proposed approach.

REFERENCES

[1] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," *IDC iView: IDC Analyze the future*, vol. 2007, no. 2012, pp. 1–16, 2012.

[2] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan, "Youtube-8m: A large-scale video classification benchmark," *arXiv preprint arXiv:1609.08675*, 2016.

[3] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, *et al.*, "Finding a needle in haystack: Facebook's photo storage.," in *OSDI*, vol. 10, pp. 1–8, 2010.

[4] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[5] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2011.

[6] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *2013 IEEE international symposium on workload characterization (IISWC)*, pp. 56–65, IEEE, 2013.

[7] SNIA, "Computational storage technical working group," *https://www.snia.org/computational(date of access: 05.03.2019)*, 2019.

[8] M. Cornwell, "Anatomy of a solid-state drive.," *Commun. ACM*, vol. 55, no. 12, pp. 59–63, 2012.

[9] K. Eshghi and R. Micheloni, "Ssd architecture and pci express interface," in *Inside solid state drives (SSDs)*, pp. 19–45, Springer, 2013.

[10] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: opportunities and challenges," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 1221–1230, ACM, 2013.

[11] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, *et al.*, "Biscuit: A framework for near-data processing of big data workloads," in *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 153–165, IEEE Press, 2016.

[12] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong, "Yoursql: a high-performance database system leveraging in-storage computing," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 924–935, 2016.

[13] Y. Kang, Y.-s. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart ssd," in *2013 IEEE 29th symposium on mass storage systems and technologies (MSST)*, pp. 1–12, IEEE, 2013.

[14] S. Seshadri, M. Gahagan, M. S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A user-programmable ssd.," in *OSDI*, pp. 67–80, 2014.

[15] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, "Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines.," in *FAST*, pp. 119–132, 2013.

[16] G. Koo, K. K. Matam, H. Narra, J. Li, H.-W. Tseng, S. Swanson, M. Annavaram, *et al.*, "Summarizer: trading communication with computing near storage," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 219–231, ACM, 2017.

[17] R. Marimont and M. Shapiro, "Nearest neighbour searches and the curse of dimensionality," *IMA Journal of Applied Mathematics*, vol. 24, no. 1, pp. 59–70, 1979.

[18] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *arXiv preprint arXiv:1702.08734*, 2017.

[19] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng, "Query-aware locality-sensitive hashing for approximate nearest neighbor search," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 1–12, 2015.

[20] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin, "Srs: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index," *Proceedings of the VLDB Endowment*, vol. 8, no. 1, pp. 1–12, 2014.

[21] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, 2014.

[22] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[23] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.

[24] J. Sivic and A. Zisserman, "Video google: A text retrieval approach to object matching in videos," in *null*, p. 1470, IEEE, 2003.

[25] V. Boppana, S. Ahmad, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig, "Ultrascale+ mpsoc and fpga families," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, pp. 1–37, IEEE, 2015.

[26] D. W. Walker and J. J. Dongarra, "Mpi: A standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[27] M. Fasheh, "Ocfs2: The oracle clustered file system, version 2," in *Proceedings of the 2006 Linux Symposium*, vol. 1, pp. 289–302, 2006.

[28] M. Aumüller, E. Bernhardsson, and A. Faithfull, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *Information Systems*, 2019.

[29] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[30] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *Computing in Science & Engineering*, no. 1, pp. 46–55, 1998.

[31] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger, "Active disk meets flash: A case for intelligent ssds," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pp. 91–102, ACM, 2013.

[32] S. Kim, H. Oh, C. Park, S. Cho, S.-W. Lee, and B. Moon, "In-storage processing of database scans and joins," *Information Sciences*, vol. 327, pp. 183–200, 2016.

[33] S. Rezaei, K. Kim, and E. Bozorgzadeh, "Scalable multi-queue data transfer scheme for fpga-based multi-accelerators," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pp. 374–380, Oct 2018.

[34] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "Bluedbm: An appliance for big data analytics," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 1–13, ACM, 2015.

[35] S. Rezaei, C. Hernandez-Calderon, S. Mirzamohammadi, E. Bozorgzadeh, A. Veidenbaum, A. Nicolau, and M. J. Prather, "Data-rate-aware fpga-based acceleration framework for streaming applications," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–6, Nov 2016.

[36] Y. Feng, J. Tang, M. Liu, C. Wang, and J. Xie, "Fast document cosine similarity self-join on gpus," in *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 205–212, IEEE, 2018.

[37] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[38] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 1016–1027, 2012.

[39] J. Maillo, I. Triguero, and F. Herrera, "A mapreduce-based k-nearest neighbor approach for big data classification," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 2, pp. 167–172, IEEE, 2015.

[40] G. Song, J. Rochas, L. El Beze, F. Huet, and F. Magoules, "K nearest neighbour joins for big data on mapreduce: a theoretical and experimental analysis," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 9, pp. 2376–2392, 2016.

[41] T. Liu, C. Rosenberg, and H. A. Rowley, "Clustering billions of images with large scale nearest neighbor search," in *2007 IEEE Workshop on Applications of Computer Vision (WACV'07)*, pp. 28–28, IEEE, 2007.

[42] G. Chatzimilioudis, C. Costa, D. Zeinalipour-Yazti, W.-C. Lee, and E. Pitoura, "Distributed in-memory processing of all k nearest neighbor queries," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 4, pp. 925–938, 2016.

[43] N. El Maliki, H. Silkan, and M. El Maghri, "Efficient indexing and similarity search using the geometric near-neighbor access tree (gnat) for face-images data," *Procedia computer science*, vol. 148, pp. 600–609, 2019.

[44] J.-Y. Li and J.-H. Li, "Approximately nearest neighborhood image search using unsupervised hashing via homogeneous kernels," *Complexity*, vol. 2018, 2018.

[45] A. J. Muller-Molina and T. Shinohara, "Efficient similarity search by reducing i/o with compressed sketches," in *2009 Second International Workshop on Similarity Search and Applications*, pp. 30–38, IEEE, 2009.

[46] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Liu, P. Sadowski, E. Racah, S. Byna, C. Tull, W. Bhimji, P. Dubey, *et al.*, "Panda: Extreme scale parallel k-nearest neighbor on distributed architectures," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 494–503, IEEE, 2016.