DeepStore: In-Storage Acceleration for Intelligent Queries

Vikram Sharma Mailthody*
UIUC

Zaid Qureshi*
UIUC

Weixin Liang†
Stanford University

Ziyan Feng
UIUC

Simon Garcia de Gonzalo
UIUC

Youjie Li
UIUC

Hubertus Franke
IBM Research

Jinjun Xiong
IBM Research

Jian Huang
UIUC

Wen-mei Hwu
UIUC

ABSTRACT

Recent advancements in deep learning techniques facilitate intelligent-query support in diverse applications, such as content-based image retrieval and audio texturing. Unlike conventional key-based queries, these intelligent queries lack efficient indexing and require complex compute operations for feature matching. To achieve high-performance intelligent querying against massive datasets, modern computing systems employ GPUs in-conjunction with solid-state drives (SSDs) for fast data access and parallel data processing. However, our characterization with various intelligent-query workloads developed with deep neural networks (DNNs), shows that the storage I/O bandwidth is still the major bottleneck that contributes 56%–90% of the query execution time.

To this end, we present DeepStore, an in-storage accelerator architecture for intelligent queries. It consists of (1) energy-efficient in-storage accelerators designed specifically for supporting DNN-based intelligent queries, under the resource constraints in modern SSD controllers; (2) a similarity-based in-storage query cache to exploit the temporal locality of user queries for further performance improvement; and (3) a lightweight in-storage runtime system working as the query engine, which provides a simple software abstraction to support different types of intelligent queries. DeepStore exploits SSD parallelisms with design space exploration for achieving the maximal energy efficiency for in-storage accelerators. We validate DeepStore design with an SSD simulator, and evaluate it with a variety of vision, text, and audio based intelligent queries. Compared with the state-of-the-art GPU+SSD approach, DeepStore improves the query performance by up to 17.7×, and energy-efficiency by up to 78.6×.

*Co-primary authors.
†Work done while visiting the Systems and Platform Research Group at UIUC.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO-52, October 12–16, 2019, Columbus, OH, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358320

CCS CONCEPTS

• **Computer systems organization** → *Special purpose systems*; *Secondary storage organization*; • **Information systems** → *Information retrieval*.

KEYWORDS

Intelligent Query, In-Storage Computing, Solid-State Drive, Hardware Accelerators, Information Retrieval

ACM Reference Format:
Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia de Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mei Hwu. 2019. DeepStore: In-Storage Acceleration for Intelligent Queries. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3352460.3358320

1 INTRODUCTION

Thanks to recent advancements in deep neural networks (DNNs) and the explosive increase and ubiquitous accessibility of data, deep learning has been enabling intelligent queries in versatile data retrieval applications to improve the quality of data services [20, 58, 59, 85]. Typical examples include identifying the same person in an image database [16], retrieving music with the input of music styles and instrumentations [72], and online shopping using a picture of garment item [48]. With the increasing accuracy and performance of deep learning techniques, we envision such types of intelligent queries will dominate the emerging data services.

Unlike conventional key-based queries in transactional databases and key-value stores that organize data in structured manner, intelligent queries leverage feature vectors extracted from unstructured data to facilitate similarity comparison with deep learning techniques. The intelligent queries can be images, texts, or others describing the user intention of data to be retrieved [48, 93]. Given an intelligent query, the query engine first extracts the query feature vector, and then executes similarity comparison with the feature vectors of the source data. After that, the query engine sorts the results based on a similarity score to send top-K results to the user.

As the core component of intelligent queries, the similarity comparison often uses neural networks to improve the comparison accuracy. However, the neural networks have to be trained and they are highly non-linear, which cannot preserve the geometric

properties (e.g., triangle inequality) between feature vectors. Therefore, it is hard to build an efficient index for feature vectors [22]. Thus, we have to scan the entire database to fulfill the queries with high accuracy. Furthermore, as the feature size of each source-data item (e.g., an image) is relatively large (i.e., 0.8 - 44KB, see Table 1), it is hard to place all the extracted features in main memory to serve queries on large-scale datasets like that of Facebook's photo/video services [55, 56]. As different applications may have different types of intelligent queries, this problem is exacerbated. Therefore, to achieve high-performance intelligent queries against massive datasets, a common approach is to use GPUs in-conjunction with SSDs for fast data retrieval and parallel similarity comparison.

In this paper, we conduct the first characterization study of typical intelligent query workloads. We use two recent generations of high-end NVIDIA GPUs to run the query and a high-end NVMe SSD for data storage (see § 3). We evaluate and profile five different types of intelligent query workloads that include visual, audio, and text search (see Table 1). We observe that (1) these intelligent queries are still bottlenecked by the storage I/O. As we increase the computational resource, the I/O bottleneck becomes more severe. (2) The core function (similarity comparison) of intelligent queries mainly involve convolutional and fully-connected neural network layers, making it an ideal candidate for hardware acceleration.

To overcome the aforementioned bottlenecks, we present DeepStore, an in-storage acceleration system for intelligent queries. Unlike the existing in-storage computing solutions [12, 38, 63, 64, 81, 94] that rely on the embedded multi-core CPUs in the SSD controller to perform simple computations, DeepStore employs neural-network accelerators to compute similarity comparison operations.

However, developing accelerators for intelligent queries in SSD controllers is not easy. First, due to the limited resources in SSDs, we have to resize the in-storage accelerator with the design space exploration methodology to meet the power, memory bandwidth, and area budgets. We abstract the common neural network operations for similarity comparison, and customize the in-storage accelerators to maximize the resource efficiency for SSD controllers.

Second, to achieve maximum energy efficiency for DeepStore, we explore different SSD parallelisms that include SSD-level, channel-level, and chip-level design space. We find that mapping in-storage accelerators to channel-level parallelisms provides the most energy-efficient result, as it achieves the best trade-off between performance and resource constraints for accelerators. This provides the guideline for real-world system design for in-storage accelerators.

Third, to further improve the performance for intelligent queries, we develop an in-storage query cache that leverages the similarity-comparison techniques to conduct query lookups. Such a cache is designed with the insight that DNN-based queries have already tolerated a certain level of errors. Therefore, given a query which is similar to a cached query in the cache, DeepStore can directly return the cached query result without conducting the similarity comparison against the entire feature database. This is especially useful for intelligent queries that cannot be indexed by simple keys or hash values for exact matching.

Finally, to enable the applicability and flexibility for different types of intelligent queries, DeepStore provides a software abstraction with a set of programming APIs to enable developers to deploy their models for similarity comparison. DeepStore also develops a runtime system for in-storage accelerators, which is responsible for dispatching intelligent queries and collecting results.

To the best of our knowledge, DeepStore is the first in-storage acceleration system for intelligent queries. Overall, we make the following contributions in this paper.

- We conduct a characterization study on the typical deep-learning based intelligent queries, quantify the performance bottlenecks, and find that storage I/O bandwidth is the major bottleneck for intelligent queries.
- We develop energy-efficient in-storage accelerators for similarity comparison of feature vectors, which facilitates the offloading of intelligent queries to SSDs.
- We exploit SSD parallelisms for in-storage accelerators, conduct a thorough design space exploration at the SSD-level, channel-level, and chip-level parallelism, and find that channel-level provides the best energy-efficiency.
- We design a similarity-based query cache for intelligent queries that inherently tolerate a certain level of accuracy loss. Such a cache design would also benefit other types of queries that do not require exact matching.

We implement DeepStore using an SSD simulator constructed with SSD-Sim[15] and SCALE-Sim[80]. We evaluate DeepStore with a variety of visual, audio, and text-based applications supporting intelligent queries, and collect query traces from real-world application workloads. Experimental results show that DeepStore improves query performance by up to 17.7×, and energy efficiency by up to 78.6×, compared to the state-of-the-art GPU + SSD system.

The rest of this paper is organized as follows: § 2 provides an introduction to intelligent queries and SSD architecture. We present the characterization study of typical intelligent query workloads in § 3. § 4 discusses the DeepStore design, followed by implementation details in § 5. We evaluate DeepStore in § 6 and discuss its related work in § 7. We summarize the paper in § 8.

2 BACKGROUND

In this section, we provide a brief overview of intelligent query systems and the internal architecture of SSDs.

2.1 Intelligent Query Systems

Recent advancement in deep neural networks (DNNs) and the explosive increase and ubiquitous accessibility of data has enabled intelligent queries in multiple applications [16, 20, 25, 30, 48, 51, 58, 59, 67, 85, 93, 95, 100, 106]. The recent popular applications include person re-identification [16], style-based music retrieval [72], and question and answering systems [82]. For these applications, they require high accuracy and thus DNNs are commonly used for similarity comparison in the queries [16, 22, 90, 106]. Furthermore, since the user input of queries are no longer restricted to text (e.g., the keys in transactional database and key-value store), and can comprise of images/image-patches [92, 96, 106], sketch [70], color map [91], context map [98, 99], music [72, 97], and many more, intelligent query systems also leverage DNNs to automatically extract the features from query input to bridge the semantic gap between queries and source data [16, 22, 90, 106].

Table 1: Intelligent query applications and their characteristics

Application	Type	Description	Feature Size (KB)	#CONV layers	#FC layers	#Element wise layers	Total FLOPs	Total Weight Size	Dataset
Person Re- Identification (ReId) [16]	Visual	Identify the same person across database of stored images	44	2	2	1	9.8M	10.7MB	CUHK03 [67]
Music Information Retrieval (MIR) [72]	Audio	Retrieve music based on styles and instrumentations	2	0	3	0	1.05M	2MB	MagnaTagTune [72]
Exact Street to Shop (ESTP) [48]	Visual	Online shopping of garment item using a real-world garment item	16	0	3	0	4.72M	9MB	Street2Shop [48]
Text-based image Retrieval (TIR) [93]	Text/ Image	Retrieve images based on the description provided over a sentence query	2	0	3	1	0.79M	1.5MB	MSCOCO [27], Flickr30K [77]
Question and Answer (TextQA) [82]	Text	Rerank short text pairs that are closely related to given query	0.8	0	1	1	0.08M	0.16MB	TREC QA [82]

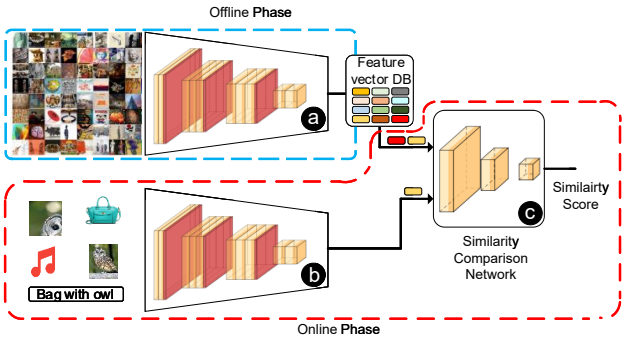


Figure 1: Workflow of DNN-based intelligent query system. In the (a) offline phase, a DNN model extracts and stores the feature vectors in storage devices such as SSDs. During the online phase, (b) the feature vector extracted from a given query, is compared against feature vectors in the stored database using (c) a similarity comparison network and a score is obtained. Finally, dataset images are then sorted based on their given score to get top-K responses (not shown).

A typical intelligent query system uses a two-branch neural network model architecture [24]. We demonstrate an example in Figure 1. During training, the network learns to score similarity by being fed with positive or negative pairs of queries and feature vectors of the source dataset. The trained network is then deployed to perform intelligent query execution in two phases, the first of which is offline and the second online. In the offline phase, for each image in the dataset, a feature vector is extracted from the intermediate layer of the dataset representation network model and stored in a feature database, as shown in Figure 1a.

In the online phase, a query feature representation network model (Figure 1b) extracts feature vectors from a given intelligent query [48, 93]. The query feature vector is then fed to similarity comparison network (SCN) model (Figure 1c). The SCN model computes the similarity between the query and the dataset feature vectors, and generate a similarity score. The similarity scores are sorted to find the top-K items that match the given query (where K is the number of data items to be retrieved). The application then fetches the corresponding matched contents from the database using the top-K results.

Compared with the traditional key-based query systems that rely on structured data, intelligent query systems extract feature vectors from unstructured data using deep learning techniques [17, 24, 66]. Since DNN models are highly non-linear and do not preserve geometric properties, such as triangle inequality, between input and feature vectors, it is hard to build an effective index over the dataset using traditional fixed metrics like Euclidean or cosine distances [16, 22, 90, 106]. Thus, we have to scan the entire database of feature vectors using the SCN to find similar results.

As the SCN execution involves compute-intensive DNN operations, intelligent query systems typically employ GPUs, which are efficient for such operations [3, 5, 31], to accelerate the SCN computation. To get the optimal GPU resource utilization, a batch of database feature vectors are compared against an intelligent query on a GPU at the same time. As feature databases are usually large since they store billions of feature vectors [40, 59] and each vector can be upto a few kilobytes in size [16], SSDs are used to store feature vector databases [59], as they provide terabytes of capacity with access latencies in the order of tens of microseconds [8]. In this paper, we focus on the in-storage acceleration design for intelligent queries with SSDs.

2.2 Modern SSD Architecture

The internal architecture and organization of an SSD are shown in Figure 3 (in gray and blue). The SSD controller contains 2-4 embedded CPU cores that execute the SSD management with Flash Translation Layer (FTL), whose functionalities include parsing block I/O commands, garbage collection, and wear-leveling [47]. To provide terabytes of capacity [54, 73], SSDs have a large number of dense NAND flash memory elements organized into multiple levels of hierarchy such as channels, chips, and planes. An SSD can have about 16-32 channels [53, 64]. Each channel consists of 4-8 flash chips that are controlled by the same flash controller and accessed via the shared channel. Each flash chip consists of 2-4 planes. Each plane has a group of blocks and each block has multiple pages.

The flash is accessed at page granularity. A page buffer is present in each plane to cache accessed flash page. During a read operation, the required page is read from a block and stored in the page buffer of the associated plane. SSDs have massive internal bandwidth [53, 64]. However, the external bandwidth of modern SSDs is limited by flash channel arbitration [79, 86], the weak processor cores in the SSD controller [81], and the bandwidth of the PCIe interface [2, 14].

3 INTELLIGENT QUERY STUDY

As intelligent queries cater to a diverse set of applications, we envision they will dominate emerging data services. However, no prior work has systematically studied these intelligent query workloads. In this section, we provide an in-depth study of a variety of intelligent query workloads and identify the performance bottlenecks and opportunities for system-level optimizations.

Experimental setup. We study five different types of intelligent query workloads that span across visual, audio, and text search, as shown in Table 1. They include (1) Person Re-identification

(ReId [16]) that searches for the same person in the image dataset; (2) Exact Street to Shop (ESTP [48]) that searches for the queried product across the produce dataset; (3) Text-based image retrieval (TIR [93]) that takes a textual query to retrieve the matched images based on the understanding of the user description. (4) Music information retrieval (MIR [72]) that searches for music samples matching a given audio clip; (5) Text-based Question and Answer (TextQA [82]) that takes a question as a query to find relevant answers from a large corpus of documents. We believe these identified intelligent query applications show the generality of emerging intelligent query systems [23, 30, 43, 46, 51, 57, 69, 83, 84, 102].

We re-implement these intelligent queries using the Tensorflow framework [13] (see details in §6). Initially, we train the models for the applications until the model accuracy is within 5% of the advertised accuracy. We extract and store the feature vectors for each application in the SSD. We use two recent generations of high-end NVIDIA GPUs: Titan Xp (Pascal) and Titan V (Volta) [3, 5]. We implement the online phase by extracting the feature vector of the query using the query feature representation network model. The stored feature database is loaded in multiple batches to perform the similarity comparison on the GPU. The GPU+SSD system is optimized such that batches of features are prefetched to host memory while the GPU computes the SCN for the previous batch. The batch sizes are taken such that the GPU utilization is nearly at 100% during the similarity comparison operation.

Study results. We breakdown the query latency into three components: GPU compute time (`Compute Time`), CPU to GPU data transfer time (`CudaMemcpy Time`), and SSD to CPU data transfer time (`SSD Read Time`). The profiling results are shown in Figure 2. As we can see, the storage I/O constitutes 56-90% of total query execution time. Although feature dataset is prefetched from the SSD while the computation executes on the GPU, the I/O time is so significant that prefetching barely improves the performance of the system. Furthermore, as we move to the newer generation of GPUs (from NVIDIA Pascal to Volta), the compute-intensive layers of the SCN perform faster by 33%. However, the overall performance of these intelligent-query workloads is not improved, since they are limited by the storage I/O bandwidth.

Observation 1: New and emerging intelligent query applications are primarily bottlenecked by the storage I/O bandwidth. With the emergence of faster GPUs [5, 31] and DNN accelerators [7, 29, 60], we expect that the performance gap between the compute and storage I/O will continue to widen.

To further understand the intelligent-query workload, we also characterize the computational patterns in the SCN and summarize the quantified results in Table 1. We observe that SCNs mainly comprise of convolutional, fully connected, and element-wise layers. Take the text-based image retrieval TIR for example, it consists of a vector dot product and three fully connected layers with sizes of 512×512, 512×256, 256×2.

Observation 2: New and emerging intelligent-query workloads involve complex operations such as convolutional and fully connected layers. Wimpy processors are not sufficient to perform the computation of similarity comparison networks, as it would add significant overhead to query latencies.

To address the challenge in the **Observation 1**, a natural solution is to move the computation closer to the data inside the storage. However, the wimpy processors in the SSD controller are not performant in executing the SCN operations. Therefore, to address the challenge in the **Observation 2**, we propose to develop in-storage accelerators to perform the SCN computation in SSDs.

4 DEEPSTORE DESIGN

According to our study of intelligent-query workloads, we show that the storage I/O is the major bottleneck, and the similarity comparison is critical to the intelligent queries. Our observations motivate us to pursue an in-storage acceleration system for DNN-based intelligent queries. However, developing accelerators in the resource-constrained SSD is non-trivial.

4.1 Design Goals and Principles

The goal of DeepStore is to achieve high performance for intelligent queries against massive dataset while providing the maximal energy-efficiency under the resource constraints in SSD controllers. In our design, we will follow the following specific principles.

- First, as the SSD controllers have limited power budget, memory capacity, and area sizes, we have to explore the design space to achieve the maximal resource efficiency for in-storage accelerators in DeepStore.
- Second, in-storage accelerators need to be scalable to exploit the internal parallelisms of SSDs. As the storage capacity could be increased by packing more flash chips, DeepStore should scale as well to achieve the maximal energy-efficiency.
- Third, like conventional database systems, DeepStore should also provide an efficient query cache to exploit the locality of user queries for better performance.
- Fourth, DeepStore should support diverse intelligent-query applications, which enables programmers to manipulate feature databases, specify neural network models, and execute queries with simple interface.

4.2 System Overview

We show the DeepStore architecture in Figure 3. DeepStore exploits the internal parallelism of the SSD and places the accelerator at three levels: the SSD-level, channel-level and chip-level as shown by (❶), (❷) and (❸) in Figure 3, respectively. DeepStore executes a lightweight query engine that parses incoming queries, manages the Query Cache (`QC`), and uses the map-reduce [36] computation model to execute the qurey across the accelerators inside the storage. It maps the SCN computational model to the accelerators and collects the results from each accelerator to generate the query response.

For a given query, its feature vector is extracted by the host and sent to DeepStore. The engine parses the query specific information and checks if a similar query exists in the Query Cache. The comparison between the new query and each cached queries is done with a Query Comparison Network (QCN) that executes on the channel-level accelerators. If there is a hit, the engine schedules the SCN on the cached entries, reports top-K results, and updates the `QC`. If it is a miss, the query engine loads the model weights to the SSD DRAM.

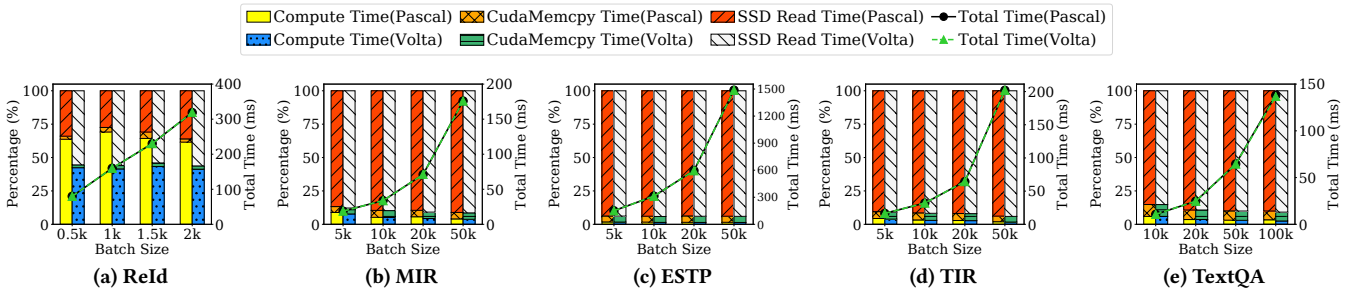


Figure 2: Performance breakdown of compute and I/O time for different intelligent query workloads, when running with two different generation of GPUs: Pascal and Volta. For these applications, 56%–90% of the execution time is spent on reading the feature dataset from SSD.

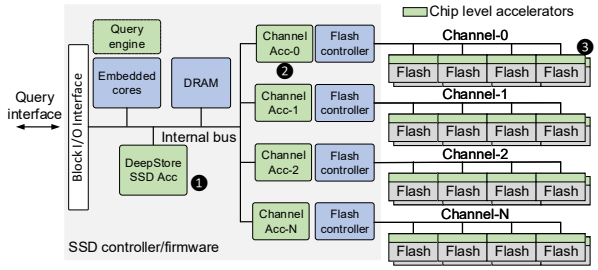


Figure 3: DeepStore augments a traditional SSD with a SSD-level (❶) and channel-level accelerators (❷) interfaced to the NAND flash controllers, and chip-level accelerators (❸) interfaced to the NAND flash chips. The SSD's embedded cores execute the query engine.

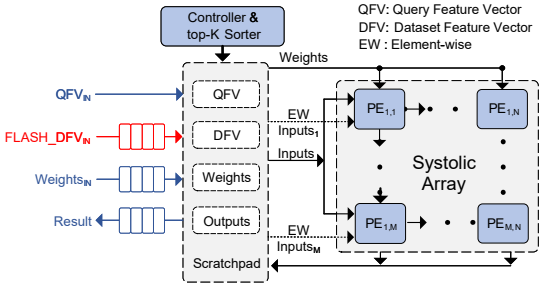


Figure 4: In-storage accelerator design of DeepStore.

It then maps the execution of the SCN to in-storage accelerators. Individual accelerators execute the computational model in parallel. Each accelerator then writes its top-K results to the SSD DRAM where the query engine merges them to generate final top-K results (i.e., the K matching results with the highest similarity scores). The query engine inserts the query with its results into the QC. When the application requests query results, the results are copied to a host specified memory location using a Direct Memory Access (DMA) operation. Each result is associated with an ObjectID, physical address of the feature vector, for reading the respective raw data.

To support different types of queries and neural network models, a programmer can interact with the DeepStore query engine using the DeepStore API (see Table 2) to specify the SCN computation model and query for their application.

4.3 In-Storage Accelerator

Based on our study in §3 of intelligent-query workloads, we classify the core intelligent-query operations into four types: fully connected, convolutional, element-wise operations and top-K sorting. An intelligent query accelerator must support these operations in an efficient manner. We demonstrate its architecture in Figure 4.

DeepStore accelerator consists of three major components: (1) a systolic array of processing engines (PEs), (2) a scratchpad memory, and (3) a controller. We use rectangular systolic array based spatial architecture, as it enables efficient mapping of fully connected and convolutional layers. We modify the regular systolic array architecture to support element-wise operations such as dot-product, subtraction, and addition, which are required for intelligent queries. This is enabled by adding an input line for each row in the first column of the systolic array. Over a simple systolic array, this speeds up the throughput of element-wise operations by the number of rows in the systolic array.

The scratchpad memory, implemented in SRAM, is used to buffer the query feature vector (QFV), database feature vector (DFV), SCN model weights, intermediate results, and final outputs. The scratchpad memory is highly banked to support the multiple parallel requests raised by the systolic array. The systolic array and scratchpad memory are controlled by the controller's finite state machine.

The controller is responsible for loading the model weights from the SSD DRAM location provided by the query engine. Given a QFV, the controller prefetches the DFVs from the flash chips and then schedules the SCN computation. To support top-K sorting, the controller is equipped with a priority queue that keeps the temporary top-K results during intelligent query execution. The priority queue is implemented with the help of a sorted tag array and mapping table. The mapping table is indexed with a tag and each entry consists of an accuracy value and feature ID. When the systolic array computes a similarity score, the controller does a binary search on the tag array, comparing the new accuracy with those in the mapping table. When the position of the new entry is identified, all entries in the tag array with a lower priority are shifted down by one, the last element is dropped and its tag is given to the new entry. The mapping table is then appropriately updated.

4.4 Interaction with Flash Memory

The DeepStore accelerator reads database feature vectors from flash memory and compares them with the query feature vector as shown in Figure 5. The accelerator's controller is responsible for programming the flash controller to move data from the flash chips to accelerator scratchpad. It does this by generating flash page

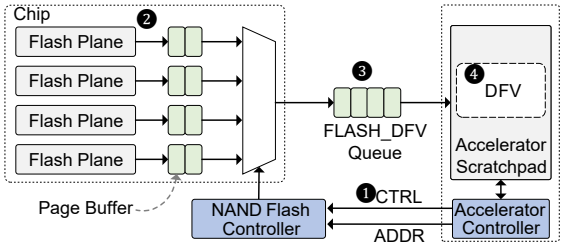


Figure 5: DeepStore accelerator interaction with the flash memory. DeepStore employs a queue to isolate prefetching data feature vector from the flash chips while performing the SCN computation.

addresses for the pages of a feature vector from the address range provided by the query engine ❶. The flash controller issues a read request to the flash chips. The pages are read from the planes in the flash chip to their associated page buffer ❷. Afterward, the flash controller moves the pages from the page buffers to the `FLASH_DFV` queue ❸. When the accelerator has completed the computation with all the features in its scratchpad, the accelerator controller consumes the pages for the next set of features from the `FLASH_DFV` queue ❹. The addition of the `FLASH_DFV` queue isolates the computation in the accelerator and the data loading from flash chip. This enables feature vectors to be prefetched while the accelerator computes on a different set of features.

To perform computation, the accelerator needs to know where to read the dataset feature vectors (`DFV`) from. To exploit the internal parallelisms of SSDs, DeepStore stripes the feature database of each application across channels and chips. Each of the feature vectors is page aligned. DeepStore employs a regular block-level FTL, and uses the FTL to get a starting physical address for the database. DeepStore stores this physical address along with the metadata to specify the `db_id`, feature-vector size, and the number of feature-vectors. This metadata is persisted in a reserved flash block, but will be cached in SSD DRAM for fast look-up for query execution. During query execution, the query provides the `db_id` of the database. The query engine uses the `db_id` to send the database's metadata along with the number of channels and chips of the SSD to the accelerator controller. The accelerator controller uses this information to compute the offset for the physical address of each feature vector it needs to fetch, avoiding the FTL address translation overhead. After that, the controller schedules the SCN computation to the systolic array. We next describe how DeepStore exploits the SSD's internal parallelism to improve performance.

4.5 Exploiting Internal Parallelisms of SSDs

Mapping the general accelerator design to the different levels of parallelism inside the SSD requires optimization along two dimensions: power budget and memory bandwidth. Modern SSDs usually have only a few GBs of DRAM memory that provides 15-26GBps of memory bandwidth to the SSD controller [14, 64], while each flash chip can provide up to 1.2GBps of bandwidth[11, 73]. Furthermore, SSDs are constrained by limited power budget (up to 75W) provided by the PCIe interface [2], of which ~20W is consumed by the existing SSD hardware during peak operation [8], leaving a power budget of 55W for DeepStore's design.

To maximize DeepStore's energy efficiency, we conduct a design space exploration for in-storage accelerators at different parallelism levels in SSDs. We constrain our exploration with the available power budget of 55W, DRAM bandwidth of 20GBps, and flash chip bandwidth of 800MBps. We use the simulation platform described in §5 and the intelligent query workloads presented in Table 1.

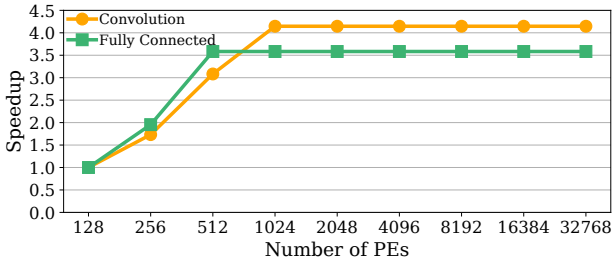


Figure 6: Performance of a systolic array accelerator with varying number of PEs for 'Convolution' and 'Fully Connected'. At each point, the aspect ratio with the fastest performance is considered.

To find the highest performing systolic array configuration, we vary the number of PEs (up to 32K) and aspect ratio of the systolic array, under the assumption of having infinite memory bandwidth. First, we gradually increase the number of PEs considering all possible aspect ratios. As shown in Figure 6, for the largest FC and ConvD layers in the studied applications, there is no performance gain beyond 512 and 1024 PEs, respectively. This is because these neural network layers require less than 1024 floating-point multiply-accumulate operations per cycle for a feature vector. After that, we search for the systolic array aspect ratios that work well for all studied intelligent query workloads. With the experiments for Figure 6, we observed that the best performing aspect ratio for the FC layer is 512 PEs in one row, and for the ConvD layer is 1024 PEs in one column. Since our applications contain both types of layers, we use these ratios as the range to bound our design space search.

To further reduce the design space, we now introduce the memory bandwidth constraints of DRAM and flash, and vary the scratchpad sizes of each accelerator. We eliminate all the design choices that cannot meet the power budget allocated for each accelerator, which covers the power consumed by the PEs as well as both on-chip and off-chip memory accesses during SCN computation. We summarize the results of our design space exploration and decisions for the accelerator at each parallelism level in SSDs as follows.

SSD-level design: For the SSD-level accelerator, the full power budget of 55W and full DRAM bandwidth are available. To maximize the reuse for computing FC layers, we use output stationary (OS) data flow in the SSD-level accelerator.[29, 49]. The accelerator has access to an 8MB scratchpad to minimize the number of DRAM transactions. This sized scratchpad avoids unnecessary off-chip memory accesses while remaining within the allocated power budget. Increasing the scratchpad size does not obtain further performance improvement, because, for applications like ReId whose weights are larger than the scratchpad size, fetching weights in DRAM and computing the SCN with the accelerator can be fully pipelined. Based on the allocated power budget, the SSD-level accelerator uses a systolic array of 2048 PEs organized in 32 rows and 64 columns (32×64) to maximize the performance of element-wise operations in DeepStore. We end up with a systolic array with more columns than rows, because of the studied applications predominantly consist of FC layers, and the accelerator's width has a direct impact on the performance for these layers.

Channel-level design: Assuming the SSD has 32 channels, each channel-level accelerator has a power budget of 1.71W and has to share the DRAM memory bandwidth with the 31 other channel-level accelerators. Since SRAMs are expensive in terms of power, each channel-level accelerator scratchpad has a small size. Similar to the SSD-level accelerator, the channel-level accelerator uses OS data flow. However, OS data flow requires data access with high memory bandwidth, as the weights and inputs need to be fetched frequently [29]. To support high bandwidth for accessing inputs and weights, we create a multi-level scratchpad memory hierarchy. The channel-level accelerator uses the SSD-level scratchpad as second level memory thus minimizing the DRAM traffic and improving the re-use of weights across multiple channel-level accelerators. The reduction in DRAM accesses results in lower power requirements. Instead of using the SSD-level accelerator's configuration, each channel-level accelerator uses a systolic array of 1024 PEs organized in a 16×64 configuration, due to the channel-level accelerator's limited power budget. In this case, for applications with large ConvD and FC layers, such as ReId, the channel-level accelerator will be limited by the performance of executing SCN with one input feature vector. For applications with smaller layers, such as TextQA, the flash channel bandwidth becomes the bottleneck, as it can only access one flash channel at 800MBps to fetch the input feature vectors.

Chip-level design: Assuming the SSD has 32 channels and 4 chips per channel, each chip-level accelerator has a power budget of 0.43W. Similar to the channel-level accelerator, the chip-level accelerator has a small scratchpad as adding large scratchpad increases design and area complexity. Recall that the chip level accelerator accesses data over the channel bus. The flash interface of these chips have minimal bandwidth and it runs at a slow frequency [11, 73]. Thus, the chip-level accelerator uses weight stationary (WS) data flow, maximizing the reuse of the weights and minimizing the bandwidth requirement across the channel bus. But the chip-level accelerator cannot be the master of the bus. Thus, the channel-level accelerator has an additional responsibility of scheduling the weights in a lockstep manner to perform the execution across all the chip-level accelerators in its channel. The chip-level accelerator uses a systolic array of 128 PEs organized in a 4×32 configuration. Adding more PEs to the chip-level accelerator would require a larger on-chip scratchpad memory or higher off-chip memory bandwidth. However, adding either would increase the power consumption of the chip-level design. Therefore, the chip-level accelerator in DeepStore is mainly limited by its computing capability.

Accelerator Placement: DeepStore supports regular read and write operations for accessing application specific feature vectors. Since DeepStore accelerators do not perform any write operations to the flash, the accelerators are placed only in the read path at any level of the design. To implement this, the read path from the flash chips are multiplexed between the regular read and accelerator response. The accelerator path is selected during query operations and the SSD controller responds to regular read/write operations with a busy signal. Thus, DeepStore accelerators do not introduce much overhead to regular storage operations.

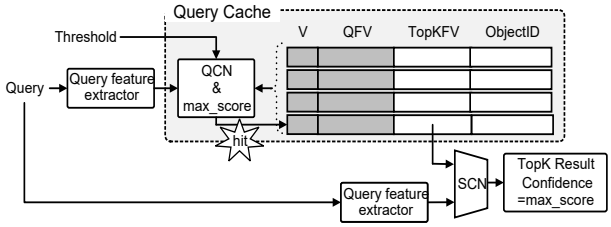


Figure 7: Learning based query cache design. Miss in the query cache results in searching over the dataset.

4.6 Query Cache

We now discuss how we leverage temporal locality and semantic similarity of queries to further improve the overall performance of DeepStore. We add a programmable similarity-based software query cache (QC). It resides in the DRAM of commodity SSD controllers for fast look-up. Each QC entry is tagged with a query feature vector (QFV). It has a valid bit (Valid), top-K database feature vectors (TopKFV), and top-K address location (ObjectID) fields, as shown in Figure 7. We use the ObjectID to store the physical address of feature vectors in the SSD.

To exploit temporal locality of queries [35], a well-known solution is to cache recent queries and their results. For any incoming query, if it exists in QC, the cached results are returned without scanning over the dataset. However, such a solution cannot exploit the semantic similarities that might exist between queries [93], like in the example described below:

(1) A brown dog is running in the sand
(2) A brown dog plays at the beach

Compared to traditional caches, where the cache is checked for exact matches, QC can exploit semantic similarity between queries to boost the performance of intelligent query workloads. QC is designed with the intuition that intelligent queries tolerate certain level of errors, detecting similarity between the queries with high confidence can further improve the system performance. A highly accurate model can guarantee greater confidence in its comparison score, thus, avoiding unnecessary misses for similar queries. Therefore, we propose to determine query similarity using a Query Comparison Network (QCN) whose structure is similar to the SCN described in § 2. Although neural network based comparison adds additional overhead (see §6.5), it is far less than the time required to scan over the dataset with the SCN.

The QCN compares two queries and returns a similarity score (qcn_score) that is used along with the QCN's accuracy (QCN_Acc) to quantify whether the cached query meets the required confidence or not. As shown in Algorithm 1, the query engine compares QFV_{new} against each of the cached entries. The query engine offloads the execution of the QCN to the DeepStore channel-level accelerators for fast comparison. It generates a score, the product of qcn_score, and the QCN_Acc. A hit occurs when the query engine finds that a QFV_{new} has a match where the complement of the score is within the specified threshold. The threshold is a hyper-parameter that depends on the model and can be tuned during deployment. The query engine selects the entry with the maximum score. When the complement of the score is beyond the threshold, QFV_{new} is

Variables: QC(n), QCN_Acc, Dataset
1: **procedure** LOOKUP(QFV_{new}, $threshold$)
2: max_index ← 0 ▹ Index of highest scoring hit
3: max_score ← 0 ▹ Score of highest scoring hit
4: **for** i ← 1,n **do**
5: **if** QC[i].valid **then**
6: qcn_score ← QCN(QFV_{new}, QC[i])
7: score ← qcn_score × QCN_Acc
8: **if** score > max_score **then**
9: max_index ← i
10: max_score ← score
11: **if** max_index ≠ 0 **and** (1− max_score) ≤ $threshold$ **then**
12: QC.promote(max_index)
13: **return** SCN(QFV_{new}, QC[max_index].features)
14: **else**
15: features ← SCN(QFV_{new}, Dataset)
16: QC.insert(QFV_{new}, features)
17: **return** features

Algorithm 1: Query engine's algorithm for Query Cache. QCN and SCN are query and similarity comparison networks, respectively.

compared against the entire dataset using the SCN. In this paper, we use the product of the QCN's accuracy and the QCN's similarity score to compare against the threshold, we believe other metrics can also be exploited.

The size of each QC entry depends on the query feature vector, application being used, and the number of database feature vectors cached. In DeepStore, each query and database feature vectors ranges from 0.8KB to 44KB, the top-K values are user defined, and the ObjectID is 8 bytes. Taking ReId, the studied application with the largest query feature vector, as an example, each query/database feature vector is 44KB and 10 top-K results for each query, then the total query cache size is 484KB. The QC is updated using an LRU replacement policy. Developers can train their similarity-comparison-network models for the QueryCache, and define the upper-bound for the error threshold that can be tolerated in their applications.

4.7 DeepStore Runtime System

In this section, we discuss how DeepStore supports diverse intelligent query applications. We first discuss the query engine and how it schedules work on the DeepStore accelerators. We then discuss the programming API the developer uses to manipulate feature databases and specify SCN models.

4.7.1 Query Engine. Query engine is a software running on the SSD embedded cores. It is responsible for consuming queries, managing the QC, scheduling work on the DeepStore accelerators, and aggregating the results. Before an accelerator can start the computation due to QC miss, the query engine must provide the accelerator with the physical addresses of the feature database. The query engine caches the metadata of stored databases in the SSD DRAM to facilitate fast access of the dataset for query execution (see § 4.4).

We use the map-reduce [36] parallel model to schedule SCN computation in DeepStore accelerators. On a QC miss, query engine maps the user specified model to the address space of the accelerators. This informs each accelerator the location of the model in the SSD-DRAM and the computation it needs to perform per feature vector. Query engine distributes the physical start and end addresses of feature database to the accelerators. The query engine instructs each accelerator to write its current top-K results to the SSD-DRAM. Each result contains dataset feature vector, associated ObjectID, and the similarity score. The query engine merges the results to generate the final top-K.

Table 2: DeepStore API.

API	Description
readDB(db_id, addr, num)	Read num features starting at addr in the database specified by db_id.
writeDB(addr, num, sz)	Create a new feature vector database and write num features to it, where each feature is of size sz bytes. The source of the data is read from a location, specified by addr, in system memory. The newly created database's db_id is returned.
appendDB(db_id, addr, num)	Appends num features to a feature vector database specified by db_id. The source of the data is read from a location, specified by addr, in system memory.
loadModel(cg, cg_size)	Load the SCN computational model and model weights, specified by cg, of cg_size bytes to DeepStore. The loaded model's model_id is returned.
query(qfv, sz, K, model_id, db_id, db_start, db_end, accel_level)	Submit an query feature vector, specified by qfv, of sz bytes in size. K specifies how many top-K results to retrieve. The SCN model, specified by model_id, is used to search over the sub-range of the database, specified by db_id and db_start and db_end locations.accel_level specifies which accelerator level to use. The query_id is returned for retrieving results.
getResults(query_id, addr, sz)	Retrieve top-K results, of sz bytes in size, for a query, specified by query_id, to a location, addr.
setQC(qcn_cg, qcn_cg_sz, sz, thr)	Configures the QC with the QCN model specified by qcn_cg and qcn_cg_sz, feature vector size of sz bytes, and a threshold of thr.

4.7.2 Programming API. DeepStore enables programmers to read and write feature vector databases and use the accelerators by means of five proposed APIs: readDB, writeDB, loadModel, query, and getResults, described in Table 2. These APIs internally use new NVMe commands to interact with the query engine.

An application developer can read and write databases of feature vectors to DeepStore using the readDB and writeDB, respectively. After writing a new feature database, DeepStore will generate 32-byte metadata that includes a db_id (8-byte), starting physical address of the database (8-byte), size of each feature (8-byte), and the number of features (8-byte). A mapping table stores the database metadata along with its db_id. DeepStore guarantees that when a feature database is written, it is stored in the format discussed in 4.4. On a read of a feature database, the user will provide the db_id as well as a range of features to read.

However, it is noted that intelligent queries are generally read-only workloads, they typically write the database once, and then query it many times. Upon a database write, typically in the form of updating the whole database with new feature vectors for each data item or adding feature vectors for new data items, DeepStore will handle it in an append fashion with writeDB or appendDB APIs. DeepStore buffers writes to ensure the alignment criteria are fulfilled and the metadata for the database is updated accordingly.

The loadModel API transfers the computational graph and the model weights, specified in the ONNX format [6], and registers it in the SSD. This enables the DeepStore APIs to be integrated with any deep learning framework. On successful execution of the API, it returns a model ID that can be used by the application to target queries with specific models.

The query API transfers the query feature from the host to the SSD DRAM. With the query API, application developers provide the db_id to specify the feature database. Query information such as model_id, db_start, and db_end is also transferred to the SSD. The

Table 3: DeepStore accelerator configurations. DeepStore exploits output stationary (OS) dataflow for SSD and channel-level accelerators and weight stationary (WS) dataflow for chip-level accelerators.

Properties	SSD-level	Channel-level	Chip-level
Technology	32nm		
Configuration	Systolic, OS	Systolic, OS	Systolic, WS
PEs	32×64	16×64	4×32
Arithmetic Precision	32bit FP		
Frequency	800MHz	800MHz	400MHz
Scratchpad Size	8MB (shared)	512KB	512KB
Accelerator Area (mm^2)	31.7	7.4	2.5

query engine uses the `accel_level` value to determine which level of accelerators to use for the query. On successful transfer of these values to the SSD, the API returns a `query_id`. The `getResults` API returns the result of a query, specified by the `query_id`, to a host memory location given by an API argument. The `setQC` API allows users to configure the `QC` with the `QCN`, feature vector size, and threshold for their applications.

5 DEEPSTORE IMPLEMENTATION

We implement DeepStore using a simulator constructed with SSD-Sim [15] and SCALE-Sim [80]. We modify SCALE-Sim to support query cache and element wise layers, and add the memory hierarchy described in § 4. SCALE-Sim is modified to generate the access patterns for the different levels of the memory hierarchy as well as the traces for loading dataset feature vectors from flash.

For each feature database, multiple flash pages can be accessed (it depends on the size of the feature vector) from the flash. We use the flash access trace generated by the modified SCALE-Sim as the input to SSD-Sim. We modified SSD-Sim to generate the overall execution time for a given query batch size and different levels of accelerators. To support multiple channel and chip-level accelerators, both SCALE-Sim and SSD-Sim are modified to generate and accept parallel accesses to flash channels and chips.

In our simulator, we implement the query engine that takes a trace of queries. To keep the simulated system and baseline system comparable, we collect the query traces from the applications running on the baseline GPU+SSD system, and pass them as input to the query engine in our simulator.

The design parameters used in our simulation are shown in Table 3. We assume the access latency for SSD scratchpad is 4 cycles and for channel/chip level scratchpads is 1 cycle. The design supports 32-bit floating point units to maintain the same accuracy as the original application.

6 EVALUATION

Our evaluation shows that: (1) DeepStore improves performance of intelligent query applications by removing the storage I/O bottleneck and exploiting different levels of parallelism inside an SSD. (2) It remains performant even when using flash chips with higher latency. (3) It provides energy efficiency over the state-of-the-art system for intelligent queries. (4) The Query Cache helps improve overall performance by exploiting temporal locality and semantic similarity of intelligent queries.

6.1 Experimental Setup

We compare DeepStore against the state-of-the-art system used for intelligent query processing consisting of an SSD for storing feature databases and a GPU for executing similarity comparison [58, 100]. For all systems, it is assumed that the query features have been extracted in a pre-processing step. We use the five applications described in § 3 and Table 1. For all evaluations, we use 32-bit floating point operations.

We compare DeepStore design with the GPU+SSD system using the latest NVIDIA Titan V (Volta) GPU. We use a server machine with a 16-core Skylake based Intel CPU running at 3.6GHz with 64GB of DRAM and a 1TB Intel DC P4500 PCIe-based SSD. The measured external bandwidth of this SSD is up to 3.2GBps. We use the simulator described in § 5 to evaluate DeepStore. We assume a flash array access latency of 53μs, 32 channels, 4 flash chips per channel, 8 planes per chip, 512 blocks per plane, and 128 pages per block. Each flash page is 16KB in size[4] and each flash channel has a bandwidth of 800MBps [11]. All DeepStore accelerator designs are evaluated in a 32nm process with a frequency of 800MHz for the SSD and channel-level accelerators, and 400MHz for the chip level accelerators.

To compute the overall accelerator energy utilization, we use existing models described in [29, 52]. We collect the number of arithmetic operations, read/write access to memory PE utilization factor, and the number of flash chip accesses from SCALE-Sim. A linear energy model is used to convert these metrics to the energy consumption of each accelerator for the neural network layers of the applications, which is similar to the ones used in [29, 52]. We scale the energy numbers for arithmetic units to 32nm [101] and use CACTI 6.5 [74] to estimate energy utilization of all SRAMs in the 32nm technology node. We assume the `itrs-hp` model for SRAMs of the SSD and channel-level accelerators, and the `itrs-low` model for the SRAMs of the chip-level accelerators due to the power constraints. DRAM energy is assumed to be 20-pJ/bit [101]. We use the power consumption of flash page access in the Intel DC P4500 SSD to compute the energy consumed by flash accesses. We extrapolate the network-on-chip energy based on the estimated wire lengths and area from CACTI.

We first disable the Query Cache to evaluate the performance of individual components of the DeepStore architecture. We warm the system by populating the SSD with 20 feature databases, each with 25GB of feature vectors. We summarize the experimental results in Table 4. We discuss the Query Cache performance in §6.5

6.2 DeepStore Performance

We first evaluate the performance of the three levels of accelerators in DeepStore, and compare them with wimpy cores inside SSD and also the GPU+SSD system. We use a high-end 8-core ARM-A57[64] CPU as wimpy cores inside the SSD controller. We pick 2K, 50K, 50K, 50K, and 100K batch sizes for ReId, MIR, ESTP, TIR, and TextQA, respectively. The batch sizes are picked such that the GPU utilization is maximized during SCN computation.

We show the performance gains of wimpy cores and DeepStore against the solution of using a GPU for all tested applications in

[1]The chip-level accelerator can not execute ReId due to limited compute and on-chip memory resources.

Table 4: Applications used in our evaluation. We summarize DeepStore's improvements on both performance and energy compared to the traditional system using GPUs.

Application	Level of Parallelism	Speedup	Energy Efficiency Improvement
ReId [1]	SSD	0.1×	0.7×
	Channel	3.9×	17.1×
MIR	SSD	0.3×	1.6×
	Channel	8.3×	28.0×
	Chip	1.0×	2.6×
ESTP	SSD	0.6×	2.8×
	Channel	13.2×	38.6×
	Chip	1.9×	3.2×
TIR	SSD	0.4×	2.1×
	Channel	10.7×	35.6×
	Chip	1.5×	3.7×
TextQA	SSD	0.4×	2.2×
	Channel	17.7×	78.6×
	Chip	4.6×	13.7×

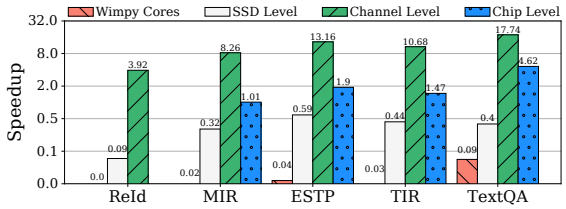


Figure 8: Performance comparison of wimpy cores, SSD, channel, and chip-level accelerators with the traditional GPU+SSD system. DeepStore performs 1.0-4.7× and 3.1-17.7× faster than GPU+SSD with the chip and channel-level accelerators, respectively.

Figure 8. The wimpy cores are 4.5-22.8× slower than GPU+SSD baseline system. Since in-storage accelerators have much higher parallelism than wimpy cores, it is obvious that DeepStore performs much better than using wimpy cores. Henceforth, we exclude the comparison with wimpy cores in the remaining evaluation.

SSD-Level Accelerator: According to Figure 8, DeepStore's SSD-level accelerator performs the worst of all systems compared in all applications. The SSD-level accelerator performs 1.7-11.0× slower than the GPU+SSD system for the intelligent query workloads. Although the SSD-level accelerator has access to high SSD internal bandwidth and does not suffer from the storage I/O bottleneck, its performance becomes limited by the performance of the similarity comparison between the query and feature database, and the lack of parallelism (see Figure 6).

Channel-Level Accelerators: Using channel level accelerators gives the best performance out of all compared systems. The channel-level accelerators perform 3.9-17.7× and 14.8-44.5× better than GPU+SSD baseline and SSD-level accelerator, respectively. The performance gains are attributed to the removal of the storage I/O bottleneck, exploitation of the SSD's internal channel-level parallelism, and higher reuse of weights (32×) in the shared scratchpad.

Chip-Level Accelerators: DeepStore's chip-level accelerator has limited computing and on-chip memory resources, and thus cannot execute large models, like the one for ReId. The same limitations lead to higher latency when compared to the channel-level accelerators, in the applications using fully connected layers like MIR, ESTP, and TIR. However, due to the 128-way (32 channels × 4 chips) parallelism exploited, and the removal of the high latency copy over PCIe, the chip-level accelerator design performs 1.1×, 2.1×, 1.6×, and 5.2× better than the GPU+SSD system for MIR, ESTP, TIR, and TextQA, respectively.

DeepStore's channel-level accelerator design achieves the best performance, as it provides the best trade-off between the parallelism level exploited and resource utilized per accelerator.

6.3 SSD Sensitivity Analysis

In this section, we change different SSD parameters to evaluate the sensitivity of DeepStore.

Impact of Flash Page Read Latency. We vary the read latency of the flash array from 7μsec, modeling a fast high-end SSD [10], to 212μsec, modeling a more commodity SSD [1, 9], and show that our designs remain performant even for slow flash chips. Figure 9c and Figure 9d show the effect on the performance of the channel and chip level accelerators, respectively. The performance of the system with a flash read latency of 53μsec was used as the baseline for this evaluation. The flash read latency does not affect the performance of the SSD-level accelerator and GPU+SSD system as they are bounded by compute latency and external SSD bandwidth, respectively.

For the channel and chip-level accelerators, decreasing the flash latency does not improve the performance of the accelerators significantly. However, if the flash latency is increased by a factor of 4, to 212μsec, the performance is reduced by only 10.1% and 3.9% for the channel and chip-level accelerators, respectively. This low variation in performance is because the accelerator is bounded by compute latency. *Thus, DeepStore accelerators can be used with cheaper and higher-latency flash chips while obtaining reasonable performance.*

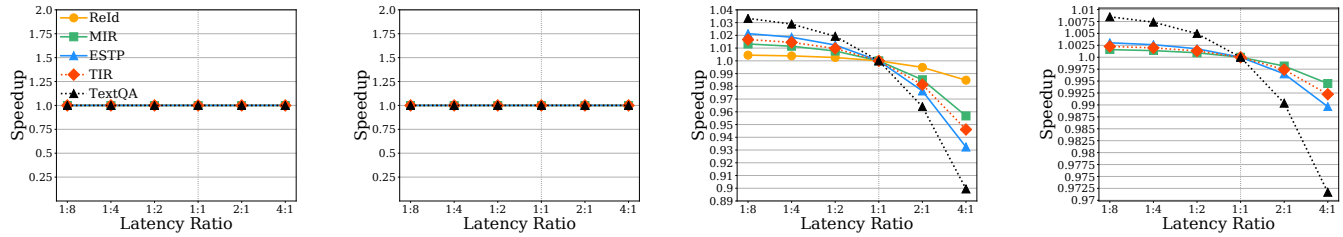
Impact of External and Internal SSD Bandwidth. We now evaluate the performance of all systems, as we vary the internal and external I/O bandwidth. We use MIR for this evaluation, however all of the intelligent query applications mentioned in § 3 exhibit the same behavior.

First, we vary the internal SSD bandwidth by varying the number of channels inside the SSD. As shown in Figure 10a, as the number of channels increases beyond 8, the performance of the GPU+SSD system does not change because the system is limited by the external SSD bandwidth over PCIe (3.2GBps). Due to the limited external bandwidth, the high internal SSD bandwidth can not be exposed to the GPU+SSD system. The SSD-level accelerator does not see any change in performance because its bounded by its compute latency for these applications and it can not exploit the higher parallelism. *However, the performance of the channel and chip-level accelerators scales linearly with the number of channels, as this improves both the internal bandwidth and the parallelism exploited by these designs.*

The GPU+SSD system can exploit multiple SSDs to get a higher aggregate I/O bandwidth when reading the dataset batch from SSDs to the host memory. As shown in Figure 10b, although the performance of the traditional system improves as more SSDs are added, it does not scale at the same rate as the number of SSDs, like the case of DeepStore. This is because in the GPU+SSD system, the storage I/O performance improves but the time to compute the SCN remains constant. *On the other hand, the compute capability of all DeepStore designs scales linearly with the number of SSDs.*

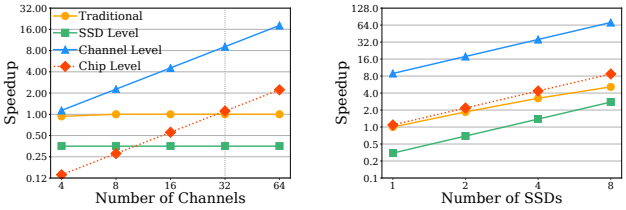
6.4 Energy Efficiency

In this section, we evaluate the energy efficiency of DeepStore. We first evaluate the energy efficiency of each level of acceleration in DeepStore against the Volta GPU. The power consumption of



(a) Traditional System (b) DeepStore - SSD Level (c) DeepStore - Channel Level (d) DeepStore - Chip Level

Figure 9: Effect of varying flash read latency on the performance of each tested system. All values are normalized to the tested system's performance with a flash read latency of 53μsec. As the flash read latency is quadrupled to 212μsec, DeepStore remains within 89.9% of its performance with a flash read latency of 53μsec.



(a) Varying Internal SSD BW (b) Varying External I/O BW

Figure 10: Effect of varying the internal SSD bandwidth (BW) by varying the number of channels in the SSD and varying the external bandwidth by varying the number of SSDs on the performance of MIR. All values are normalized to the traditional system's performance with one SSD consisting of 32 channels.

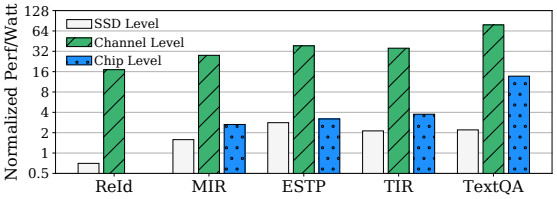


Figure 11: Energy efficiency of DeepStore designs normalized to the Volta GPU in the traditional system.

Volta GPU is measured using `nvidia-smi`. As shown in Figure 11, the channel-level accelerators are the most energy efficient design, providing up 78.6× better performance per watt, compared to the Volta GPU. This is because the channel-level accelerators provide the best trade-off between systolic array size and on-chip SRAM utilization with the shared second-level scratchpad. The SSD-level accelerator is only 0.7× as efficient as the GPU for ReId, a compute intensive workload, as it is only 0.1× as fast as the GPU for this workload. However, it is up to 2.8× more energy efficient than the GPU for data-intensive workloads, like TextQA, as it takes only 12.5% of the power of the GPU. The design using chip-level accelerators provides up to 13.7× better energy efficiency than the Volta GPU. However, due to its stricter resource constraints on the systolic array and on-chip SRAM, it only achieves 8.2-17.5% of the energy efficiency of channel-level accelerators.

We also show the energy breakdown of DeepStore. The SSD-level accelerator mainly consumes energy on memory accesses and flash accesses, as shown in Figure 12. The channel-level accelerators' energy consumption is dominated by memory accesses. This is mainly due to the higher reuse of the 8MB scratchpad shared by all

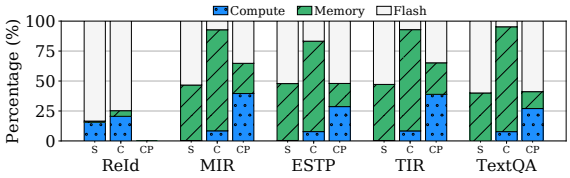


Figure 12: Power consumption breakdown of DeepStore for SSD level accelerator (S), channel level accelerators (C), and chip level accelerators (CP) for the different applications.

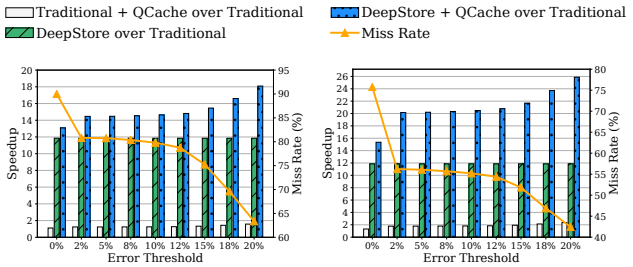
32 channel-level accelerators. The chip-level accelerators consume most of their energy in accessing the flash. As for ReId, most of the SSD and channel-level accelerators' energy is caused by the flash accesses, as each of its feature vector uses three flash pages.

6.5 Query Cache Performance

We now demonstrate the benefit of the Query Cache for intelligent query applications. We use TIR [93] for evaluation. We add noise to the Flickr30K Entities test dataset [77, 104] without affecting the ground truth, to get 100M images (192GB of feature vectors) and 100K queries. We use the Universal Sentence Encoder [26], which has been trained on the SNLI corpus [21], to compare incoming queries with queries cached in the Query Cache. The Universal Sentence Encoder gives a similarity score between two intelligent queries. We use the product of this score and the encoder's average test accuracy to compare against the defined threshold. We generate a stream of queries by sampling the dataset queries with two different distributions: uniform and Zipfian with α equal to 0.7. We warm-up the Query Cache using the query trace and then measure the query performance.

First, we evaluate the performance of the Query Cache with 1K cache entries across the different query distributions and error thresholds. The cost of searching the entire query cache of 1K entries for this application is 0.3 milliseconds, which is significantly less than the cost, 34.1 milliseconds, of scanning the entire feature database with SCN.

As shown in Figure 13, adding the Query Cache to the GPU+SSD system and DeepStore provides performance gains of up to 2.8× and 25.9×, respectively, compared to the GPU+SSD system without the Query Cache. Although the GPU+SSD baseline system benefits from the Query Cache, DeepStore benefits 10× more because of the significantly lower miss penalty for intelligent queries. To reach the same performance as DeepStore configured without a Query Cache, the GPU+SSD baseline system needs to cache at least 64.7% of the dataset. Relaxing the query comparison error threshold from



(a) Uniform (b) Zipfian α=0.7

Figure 13: Query Cache performance and Query Cache miss rate with Uniform and Zipfian(α = 0.7) distributions on the queries.

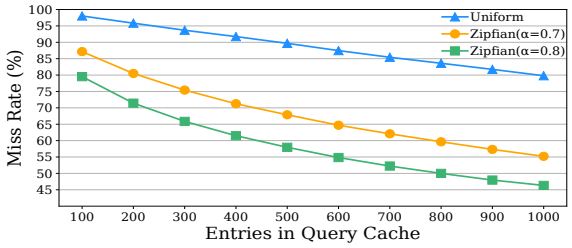


Figure 14: Query Cache miss rate as a function of cache size. With distributions providing locality, the benefit of larger caches reduces.

0% to 20% improves the query performance by up to 1.7×, as the miss rate decreases. We evaluate different α values for Zipfian, and observe similar performance trends.

Next, we evaluate the impact of the Query Cache size on the miss rate of the cache. We choose a query comparison threshold of 10% for this evaluation. As shown in Figure 14, although larger Query Cache sizes reduce the miss rate, for query distributions exhibiting locality (i.e., Zipfian), the benefit reduces with larger caches. Thus, it suffices to have a small Query Cache (about 22MB in size for TIR with top-K=10 and caching 1K entries) in the DRAM of the SSD.

7 RELATED WORK

In-storage Computing: There have been several works exploring near-storage or in-storage processing for applications such as database query processing [38, 39, 63, 64, 81], key-value store [81], map-reduce workloads [45, 63], signal processing [19], and data analysis [87, 88]. They leverage the embedded CPU in the SSD's controller to perform compute operations to avoid data transfer overhead over PCIe. Unlike these applications, intelligent query workloads require support for complex compute operations such as FC and ConvD layers as discussed in §3. Although it is possible to execute these workloads using the wimpy embedded cores [45, 64, 81, 89], it is significantly slower than DeepStore. Prior work has tried to place application specific hardware accelerators in the SSD such as [12, 18, 33, 61, 62]. However, to the best of our knowledge, we are the first to explore intelligent-query workloads for in-storage acceleration as well as to discuss the trade-offs for exploiting different levels of parallelism in the SSD.

DNN Accelerator: Several accelerator designs have been proposed to speed up the training and inference computation of popular DNN models [28, 29, 34, 44, 49, 50, 52, 60, 65, 68, 71, 75, 78]. Additional optimizations such as quantization, weight pruning [49, 103, 105], data-flow optimizations [29, 44, 65] are discussed to speed up the computation and improve energy efficiency. However, none of these accelerators are incorporated in flash storage and are not optimized for intelligent query workloads. Although we do not perform any optimization like quantization, low-precision operations, and others, we believe the optimization work in the accelerator community can be incorporated into the DeepStore architecture to gain higher performance and energy efficiency. We consider these possibilities as the extensions of our DeepStore work.

Exploiting Query Properties: Conventional query systems leverage similarity between their data to build indices for fast lookups [20, 37, 58, 59, 85]. In the case of intelligent-query systems, such indices cannot be built due to the non-linearity introduced by extracting feature vectors from DNN models and query diversity [16, 22, 90, 106]. To gain performance, traditional systems have exploited caching of queries and their results [32, 42]. They use linear methods to search the cache for exact or similar matches. However, these cannot provide highly accurate semantic similarity between queries, resulting in unnecessary expensive miss penalties. In DeepStore, the query cache uses a DNN-based similarity comparison network to perform similarity lookup in the cache. Recent work [41, 76] has explored reorganizing feature vectors in-storage for efficient search operations. Such techniques can also be exploited by DeepStore to further improve performance. The query cache proposed in DeepStore can be leveraged in several other domains as well, which we wish to explore in the future.

8 CONCLUSION

In this work, we study a diverse set of representative intelligent query workloads, and show that on a state-of-the-art GPU+SSD system, these applications are limited by storage I/O bandwidth. To address this, we propose DeepStore, an in-storage accelerator system. It consists of energy-efficient in-storage accelerators, a light-weight runtime query engine, and a similarity based in-storage query cache. We study various trade-offs associated with designing accelerators for intelligent queries at different parallelism levels of SSD, and provide a detailed system design and implementation of DeepStore. We show that DeepStore improves the query performance by up to 17.7× and energy efficiency by up to 78.6×, compared to the state-of-the-art system using GPUs.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and shepherd for their helpful comments and feedback. This work was partially supported by the Applications Driving Architectures (ADA) Research Center and Center for Research on Intelligent Storage and Processing-in-memory (CRISP), JUMP Centers co-sponsored by SRC and DARPA, IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizon Network, NSF grant CNS-1850317, and NSF grant CCF-1919044.

REFERENCES

[1] 2007. Micron C200 1.8inch NAND Flash SSD.
[2] 2015. PCIe 3.0 Specification. https://pcisig.com/specifications.
[3] 2016. NVIDIA Tesla P100 Architecture Whitepaper. https://www.nvidia.com/object/pascal-architecture-whitepaper.html.

[4] 2017. Intel/Micron 64L 3D NAND Analysis.
[5] 2017. NVIDIA Tesla V100 GPU Architecture Whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.
[6] 2017. Open Neural Network Exchange format. https://onnx.ai/.
[7] 2018. Intel Nervana Nueral Network Processors. https://ai.intel.com/nervana-nnp/.
[8] 2018. Intel SSD DC P4500 Series.
[9] 2018. Micron 9200 NVMe SSD.
[10] 2018. Ultra-Low Latency with Samsung Z-NAND SSD.
[11] 2019. Open NAND Flash Interface Specification 4.1. http://www.onfi.org/-/media/client/onfi/specs/onfi_4_1_gold.pdf?la=en.
[12] 2019. See Our Machine Learning Accelerator at Embedded World.
[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA.
[14] Ahmed Abulila, Vikram S Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. 2019. FlatFlash: Exploiting the Byte-Accessibility of SSDs within A Unified Memory-Storage Hierarchy. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. Providence, RI, USA.
[15] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *Proceeding of the USENIX 2008 Annual Technical Conference (USENIX ATC'08)*. Boston, MA.
[16] Ejaz Ahmed, Michael Jones, and Tim K Marks. 2015. An Improved Deep Learning Architecture for Person Re-identification. In *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition(CVPR'15)*. Boston, MA.
[17] Artem Babenko, Anton Slesarev, Alexandr Chigorin, and Victor Lempitsky. 2014. Neural codes for image retrieval. In *Proceedings of the European conference on computer vision (ECCV'14)*. Zurich, Switzerland.
[18] Duck-Ho Bae, Jin-Hyung Kim, Sang-Wook Kim, Hyunok Oh, and Chanik Park. 2013. Intelligent SSD: A Turbo for Big Data Mining. In *Proceedings of the 22nd ACM International Conference of Information Knowledge Management (CIKM'13)*. San Francisco, CA.
[19] S. Boboila, Y. Kim, S. S. Vazhkudai, P. Desnoyers, and G. M. Shipman. 2012. Active Flash: Out-of-core Data Analytics on Flash Storage. In *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST'12)*. Monterey, CA.
[20] Fedor Borisyuk, Albert Gordo, and Viswanath Sivakumar. 2018. Rosetta: Large Scale System for Text Detection and Recognition in Images. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18)*. London, United Kingdom.
[21] Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. 2015. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP'15)*. Lisbon, Portugal.
[22] Tolga Bozkaya and Meral Ozsoyoglu. 1997. Distance-based Indexing for High-dimensional Metric Spaces. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '97)*. Tucson, AZ.
[23] Joel Brogan, Paolo Bestagini, Aparna Bharati, Allan Pinto, Daniel Moreira, Kevin Bowyer, Patrick Flynn, Anderson Rocha, and Walter Scheirer. 2017. Spotting The Difference: Context Retrieval and Analysis for Improved Forgery Detection and Localization. In *Proceedings of the IEEE International Conference on Image Processing (ICIP'17)*. Beijing, China.
[24] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1993. Signature Verification Using a "Siamese" Time Delay Neural Network. In *Proceedings of the 6th International Conference on Neural Information Processing Systems (NIPS'93)*. San Francisco, CA.
[25] Matthew Brown, Gang Hua, and Simon Winder. 2011. Discriminative Learning of Local Image Descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI'11)* 33, 1 (2011).
[26] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. 2018. Universal Sentence Encoder. *arXiv e-prints* (March 2018).
[27] Xinlei Chen, Hao Fang, Tsung-Yi Lin, Ramakrishna Vedantam, Saurabh Gupta, Piotr Dollar, and C. Lawrence Zitnick. 2015. Microsoft COCO Captions: Data Collection and Evaluation Server. arXiv:cs.CV/1504.00325
[28] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. Cambridge, England.
[29] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits (SSC'17)* 52, 1 (Jan 2017).
[30] Z. Cheng, X. Wu, Y. Liu, and X. Hua. 2017. Video2Shop: Exact Matching Clothes in Videos to Online Shopping Images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*. Honolulu, HI.
[31] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).
[32] Flavio Chierichetti, Ravi Kumar, and Sergei Vassilvitskii. 2009. Similarity Caching. In *Proceedings of the Twenty-eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'09)*. Providence, Rhode Island, USA.
[33] Benjamin Y. Cho, Won Seob Jeong, Doohwan Oh, and Won Woo Ro. 2013. XSD: Accelerating MapReduce by Harnessing the GPU inside an SSD. In *Proceedings of the 1st Workshop on Near-Data Processing in Conjunction with the 46th IEEE/ACM International Symposium on Microarchitecture (WoNDP)*. Davis, CA.
[34] Jason Clemons, Chih-Chi Cheng, Iuri Frosio, Daniel Johnson, and Stephen W Keckler. 2016. A Patch Memory System for Image Processing and Computer Vision. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. Taipei, Taiwan.
[35] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. ACM, New York, NY, USA.
[36] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[37] J. Deng, A. C. Berg, and L. Fei-Fei. 2011. Hierarchical semantic indexing for large scale image retrieval. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR'11)*.
[38] 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 (SIGMOD'13)*. New York, NY.
[39] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. New York, NY, USA.
[40] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys'18)*. Porto, Portugal.
[41] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. 2018. Bandana: Using non-volatile memory for storing deep learning models. *In proceedings of SysML Conference (SysML'18)* (2018).
[42] Fabrizio Falchi, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fausto Rabitti. 2008. A Metric Cache for Similarity Search. In *Proceedings of the 2008 ACM Workshop on Large-Scale Distributed Systems for Information Retrieval*. Napa Valley, California, USA.
[43] Yuxun Fang, Qiuxia Wu, and Wenxiong Kang. 2018. A Novel Finger Vein Verification System Based on Two-stream Convolutional Network Learning. *Neurocomputing* (2018).
[44] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. Xi'an, China.
[45] B. Gu, A. S. Yoon, D. H. Bae, I. Jo, J. Lee, J. Yoon, J. U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang. 2016. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. Seoul, Korea.
[46] X. Gu, Y. Wong, L. Shou, P. Peng, G. Chen, and M. S. Kankanhalli. 2018. Multi-Modal and Multi-Domain Embedding Learning for Fashion Retrieval and Analysis. *IEEE Transactions on Multimedia* (2018).
[47] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. Washington, DC, USA.
[48] M Hadi Kiapour, Xufeng Han, Svetlana Lazebnik, Alexander C Berg, and Tamara L Berg. 2015. Where to Buy It: Matching Street Clothing Photos in Online Shops. In *Proceedings of the IEEE international conference on computer vision (ICCV'15)*. Santiago, Chile.
[49] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. Seoul, Republic of Korea.
[50] Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'16)*. Vancouver, Canada.

[51] Xufeng Han, Thomas Leung, Yangqing Jia, Rahul Sukthankar, and Alexander C Berg. 2015. Matchnet: Unifying Feature and Metric Learning for Patch-based Matching. In *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition(CVPR'15)*. Boston, MA.

[52] Kartik Hegde, Rohit Agrawal, Yulun Yao, and Christopher W Fletcher. 2018. Morph: Flexible Acceleration for 3D CNN-based Video Understanding. In *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*.

[53] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST'17)*. Santa clara, CA.

[54] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. 2015. Unified Address Translation for Memory-mapped SSDs with FlashMap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. Portland, OR.

[55] Qi Huang, Petchean Ang, Peter Knowles, Tomasz Nykiel, Iaroslav Tverdokhlib, Amit Yajurvedi, Paul Dapolito VI, Xifan Yan, Maxim Bykov, Chuen Liang, Mohit Talwar, Abhishek Mathur, Sachin Kulkarni, Matthew Burke, and Wyatt Lloyd. 2017. SVE: Distributed Video Processing at Facebook Scale. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'17)*. Shanghai, China.

[56] Qi Huang, Ken Birman, Robbert van Renesse, , Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. 2013. An Analysis of Facebook Photo Caching. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'13)*. Farmington, PA.

[57] Y. Huang, W. Wang, and L. Wang. 2017. Instance-Aware Image and Sentence Matching with Selective Multimodal LSTM. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*. Honolulu, HI.

[58] Yushi Jing, David Liu, Dmitry Kislyuk, Andrew Zhai, Jiajing Xu, Jeff Donahue, and Sarah Tavel. 2015. Visual Search at Pinterest. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'15)*. Sydney, Australia.

[59] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734* (2017).

[60] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. Toronto, Canada.

[61] S. Jun, A. Wright, S. Zhang, S. Xu, and Arvind. 2018. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*. Los Angeles, CA.

[62] 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. *SIGARCH Comput. Archit. News* 43, 3 (June 2015).

[63] Y. Kang, Y. Kee, E. L. Miller, and C. Park. 2013. Enabling cost-effective data processing with smart SSD. In *Proceedings of the 28th IEEE Conference on Mass Storage Systems and Technologies (MSST'13)*. Lake Arrowhead, CA.

[64] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading Communication with Computing Near Storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. Cambridge, Massachusetts.

[65] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. *SIGPLAN Not.* 53, 2 (March 2018).

[66] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep Learning. *Nature* (2015).

[67] Wei Li, Rui Zhao, Tong Xiao, and Xiaogang Wang. 2014. Deepreid: Deep Filter Pairing Neural Network for Person Re-identification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'14)*. Columbus, OH.

[68] Youjie Li, Xiaohao Wang, Iou-Jen Liu, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating Distributed Reinforcement Learning with In-Switch Computing. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*. Phoenix, AZ.

[69] Hongye Liu, Yonghong Tian, Yaowei Yang, Lu Pang, and Tiejun Huang. 2016. Deep relative distance learning: Tell the difference between similar vehicles. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. Las Vegas, NV.

[70] Li Liu, Fumin Shen, Yuming Shen, Xianglong Liu, and Ling Shao. 2017. Deep Sketch Hashing: Fast Free-hand Sketch-based Image Retrieval. In *Proceedings of the 30th IEEE Conference on Computer Vision and Pattern Recognition(CVPR'17)*. Honolulu, HI.

[71] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An Instruction Set Architecture for Neural Networks. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. Seoul, South Korea.

[72] R. Lu, K. Wu, Z. Duan, and C. Zhang. 2017. Deep ranking: Triplet MatchNet for music metric learning. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'17)*.

[73] Micron. 2017. Micron 3D NAND technology. https://www.micron.com/products/nand-flash/3d-nand.

[74] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A Tool to Model Large Caches. *HP laboratories* (2009).

[75] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. *SIGARCH Comput. Archit. News* 45, 2 (June 2017).

[76] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. 2018. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. *arXiv preprint arXiv:1811.09886* (2018).

[77] Bryan A. Plummer, Liwei Wang, Christopher M. Cervantes, Juan C. Caicedo, Julia Hockenmaier, and Svetlana Lazebnik. 2017. Flickr30K Entities: Collecting Region-to-Phrase Correspondences for Richer Image-to-Sentence Models. *International Journal of Computer Vision (IJCV'17)* 123 (2017).

[78] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. Taipei, Taiwan.

[79] Dong-ryul Ryu. 2012. Solid State Disk Controller Apparatus. US Patent 8,159,889.

[80] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2018. SCALE-Sim: Systolic CNN Accelerator. *arXiv preprint arXiv:1811.02883* (2018).

[81] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. Broomfield, CO.

[82] Aliaksei Severyn and Alessandro Moschitti. 2015. Learning to Rank Short Text Pairs with Convolutional Deep Neural Networks. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'15)*. New York, NY, USA.

[83] Yantao Shen, Tong Xiao, Hongsheng Li, Shuai Yi, and Xiaogang Wang. 2017. Learning Deep Neural Networks for Vehicle Re-id with Visual-spatio-temporal Path Proposals. In *Proceedings of the International Conference on Computer Vision (ICCV'17)*. Venice, Italy.

[84] Yantao Shen, Tong Xiao, Hongsheng Li, Shuai Yi, and Xiaogang Wang. 2018. End-to-End Deep Kronecker-Product Matching for Person Re-Identification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'18)*. Salt Lake City, UT.

[85] Cooper Smith. 2013. Facebook users are uploading 350 million new photos each day. *Business insider* 18 (2013).

[86] Vinay Ashok Somanache, Timothy W Swatosh, Pamela S Hempstead, Jackson L Ellis, Michael S Hicken, and Martin S Dell. 2013. Flash controller hardware architecture for flash devices. US Patent App. 13/432,394.

[87] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. 2013. Active Flash: Towards Energy-efficient, In-situ Data Analytics on Extreme-scale Machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. San Jose, CA.

[88] Devesh Tiwari, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Simona Boboila, and Peter J. Desnoyers. 2012. Reducing Data Movement Costs Using Energy Efficient, Active Computation on SSD. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems (HotPower'12)*. Hollywood, CA.

[89] H. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson. 2016. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. In *Proceedings of the 43rd IEEE Annual International Symposium on Computer Architecture (ISCA'16)*. Taipei, Taiwan.

[90] Ji Wan, Dayong Wang, Steven Chu Hong Hoi, Pengcheng Wu, Jianke Zhu, Yongdong Zhang, and Jintao Li. 2014. Deep Learning for Content-Based Image Retrieval: A Comprehensive Study. In *Proceedings of the 22nd ACM International Conference on Multimedia (ACM Multimedia'14)*. Orlando, FL.

[91] Jingdong Wang and Xian-Sheng Hua. 2011. Interactive Image Search by Color Map. *ACM Trans. Intell. Syst. Technol.* 3, 1 (Oct. 2011).

[92] Kaiye Wang, Qiyue Yin, Wei Wang, Shu Wu, and Liang Wang. 2016. A comprehensive survey on cross-modal retrieval. *arXiv preprint arXiv:1607.06215* (2016).

[93] Liwei Wang, Yin Li, Jing Huang, and Svetlana Lazebnik. 2018. Learning Two-branch Neural Networks for Image-text Matching Tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI'18)* (2018).

[94] Xiaohao Wang, You Zhou, Chance C. Coats, and Jian Huang. 2019. Project Almanac: A Time-Traveling Solid-State Drive. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*. Dresden, Germany.

[95] S. Winder, G. Hua, and M. Brown. 2009. Picking the best DAISY. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'09)*. Miami, FL.

[96] Pengcheng Wu, Steven C.H. Hoi, Hao Xia, Peilin Zhao, Dayong Wang, and Chunyan Miao. 2013. Online Multimodal Deep Similarity Learning with Application to Image Retrieval. In *Proceedings of the 21st ACM International Conference on Multimedia (MM'13)*. New YorK, NY.

[97] Baixi Xing, Kejun Zhang, Shouqian Sun, Lekai Zhang, Zenggui Gao, Jiaxi Wang, and Shi Chen. 2015. Emotion-driven Chinese Folk Music-image Retrieval Based on DE-SVM. *Neurocomputing* 148 (2015).

[98] Hao Xu, Jingdong Wang, Xian-Sheng Hua, and Shipeng Li. 2010. Image Search by Concept Map. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'10)*. Geneva, Switzerland.

[99] Hao Xu, Jingdong Wang, Xian-Sheng Hua, and Shipeng Li. 2010. Interactive Image Search by 2D Semantic Map. In *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*. Raleigh, NC.

[100] Fan Yang, Ajinkya Kale, Yury Bubnov, Leon Stein, Qiaosong Wang, Hadi Kiapour, and Robinson Piramuthu. 2017. Visual Search at eBay. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'17)*. Halifax, Canada.

[101] Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qiaoyi Liu, Steven Emberton Bell, Jeff Ou Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, and Mark Horowitz. 2018. DNN Dataflow Choice is Overrated. *arXiv preprint arXiv:1809.04070* (2018).

[102] Hantao Yao, Shiliang Zhang, Dongming Zhang, Yongdong Zhang, Jintao Li, Yu Wang, and Qi Tian. 2017. Large-scale person re-identification as retrieval. In *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME'17)*. Hong Kong.

[103] R. Yazdani, M. Riera, J. Arnau, and A. González. 2018. The Dark Side of DNN Pruning. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*. Los Angeles, CA.

[104] Peter Young, Alice Lai, Micah Hodosh, and Julia Hockenmaier. 2014. From image descriptions to visual denotations: New similarity metrics for semantic inference over event descriptions. *Transactions of the Association for Computational Linguistics (TACL'14)* 2 (2014).

[105] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-x: An accelerator for sparse neural networks. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. Taipei, Taiwan.

[106] Wengang Zhou, Houqiang Li, and Qi Tian. 2017. Recent Advance in Content-based Image Retrieval: A Literature Survey. *CoRR* (2017).