NDS: N-Dimensional Storage

Yu-Chia Liu
University of California, Riverside
Riverside, California, USA

Hung-Wei Tseng
University of California, Riverside
Riverside, California, USA

ABSTRACT

Demands for efficient computing among applications that use high-dimensional datasets have led to *multi-dimensional* computers—computers that leverage heterogeneous processors/accelerators offering various processing models to support multi-dimensional compute kernels. Yet the front-end for these processors/accelerators is inefficient, as memory/storage systems often expose only entrenched linear-space abstractions to an application, and they often ignore the benefits of modern memory/storage systems, such as support for multi-dimensionality through different types of parallel access.

This paper presents *N-Dimensional Storage* (NDS), a novel, multi-dimensional memory/storage system that fulfills the demands of modern hardware accelerators and applications. NDS abstracts memory arrays as native storage that applications can use to describe data locations and uses coordinates in any application-defined multi-dimensional space, thereby avoiding the software overhead associated with data-object transformations. NDS gauges the application demand underlying memory-device architectures in order to intelligently determine the physical data layout that maximizes access bandwidth and minimizes the overhead of presenting objects for arbitrary applications.

This paper demonstrates an efficient architecture in supporting NDS. We evaluate a set of linear/tensor algebra workloads along with graph and data-mining algorithms on custom-built systems using each architecture. Our result shows a 5.73× speedup with appropriate architectural support.

CCS CONCEPTS

• **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Hardware** → **External storage**.

KEYWORDS

heterogeneous computing, hardware accelerators, data storage systems, storage interface

ACM Reference Format:
Yu-Chia Liu and Hung-Wei Tseng. 2021. NDS: N-Dimensional Storage. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3466752.3480122

1 INTRODUCTION

As the dimensionality of compute kernels in data-intensive applications grows and Dennard scaling discontinues, modern computers must support processing models in multiple dimensions. The processing model of a conventional CPU core operates on 0-order scalar data, graphics processing units (GPUs) provide 1-order vector processing capabilities, while Nvidia's Tensor Core Units (TCUs) and Google's Tensor Processing Units (TPUs) compute on 2-order tensors (multi-dimensional matrices). The internal structures of memory and non-volatile storage devices in heterogeneous computers are also multi-dimensional; memory chips typically contain multiple internal planes or banks, while a typical modern computing device contains multiple chips and organizes chips into parallel channels and banks to maximize access bandwidth.

Though hardware accelerators, memory architectures, and applications are multi-dimensional, data-storage and memory systems still leverage an entrenched, one-dimensional addressing mode that requires applications to *serialize* high-dimensional data along a selected dimension (e.g., column or row) before storing data in a memory device. When another application needs to retrieve data, the application must also *marshal*, or say, *deserialize*, the raw data to objects in the structures and dimensions that compute kernels require. Such abstraction leads to low utilization of high-dimensional, data-intensive compute kernels in hardware accelerators due to the processing overhead of changing data layouts and the inefficient use of interconnect and memory-device bandwidths.

Conventional wisdom holds that using application-defined, more efficient data storage formats [9, 84] and optimized algorithms [1, 102] can address the mismatch between memory/storage abstractions and compute kernels. However, finding the most appropriate storage format is exceedingly challenging. First, the data-object structures that maximize throughput compute kernels on hardware accelerators may not match the layout that maximizes storage bandwidth; such a mismatch can lead to inefficient data-storage and memory-device access. Second, because modern memory and storage interfaces hide hardware details from applications and/or dynamically relocate physical data locations, the logical layout upon which an application relies may not reflect the use of physical memory space. Third, even if data are presented in some optimal, high-dimensional data layout that maximizes storage and accelerator performance for an application, the desired structures still differ among applications.

This paper proposes NDS, *N-Dimensional Storage*, to address the aforementioned memory-abstraction mismatch and the demand of modern hardware-accelerated, high-dimensional compute kernels/applications. NDS provides an interface that allows applications acting as either dataset producers or consumers to define their own views (desired abstractions) of storage-data dimensionality. The NDS space-translation layer (STL) gauges application demands and memory/storage-device characteristics to break down datasets into building blocks that match the optimal granularity in devices. The STL also maintains data structures that record the dimensionality for each dataset and the mapping of the dataset's building blocks. Upon receiving an access request for a dataset, the STL presents the dataset as an application-defined abstraction by dynamically decomposing or constructing data into/from the building blocks.

NDS offers several benefits while resolving the aforementioned issues. NDS mitigates the computation overhead of serialization or deserialization in applications because NDS does not require applications to transform datasets from/to the lowest, linear dimension that the conventional linear memory abstractions require. NDS maximizes the utilization of system-interconnect bandwidth and processing elements in hardware accelerators by allowing for unique, application-specific datasets and by interacting with applications through optimally structured data objects. Further, NDS takes a multi-dimensional approach to underlying memory arrays and building blocks to take advantage of device-level parallelism, thereby achieving high performance for arbitrary access patterns.

This paper describes a prototype NDS system built to investigate the trade-offs of NDS features in different system components. In the software-only implementation, NDS demonstrates the effectiveness of the building blocks in lowering the overhead of constructing multi-dimensional objects that compute kernels desire and achieves a 5.07× speedup for a broad range of applications, including large-scale, dense matrix/tensor algebra applications, graph traversal applications, and high-dimensional data applications. With architectural support, specifically, a controller implementing the NDS interface with STL features inside the storage device makes such a speedup possible; NDS efficiently utilizes internal parallelism, reduces the number of commands crossing the I/O interface, and lowers overhead on a host computer. The hardware implementation of NDS goes even further, achieving a 5.73× speedup for the same set of applications.

In presenting NDS, this paper makes the following contributions: (1) It is the first work to present an application-defined, multi-dimensional memory/storage abstraction as an alternative to the entrenched linear memory abstraction. (2) It demonstrates that granularities and dimensionalities of data accesses are different among devices and that simply optimizing application-based file-storage formats is insufficient. (3) It presents an efficient data-allocation strategy that gives programmers and applications an agnostic memory/storage data layout while allowing arbitrary data-access patterns to fully utilize interconnect/device bandwidth and efficiently construct multi-dimensional application objects. (4) It evaluates different NDS system architectures and shows the performance gains from each system architecture.

2 BACKGROUND

The dimensionality mismatch among hardware accelerators, memory devices, and abstractions is a primary source of processing inefficiency. This section explains the effects of mismatching dimensionalities on application performance, the challenges of tackling such mismatches, and the limitations of current research attempting to address the problem.

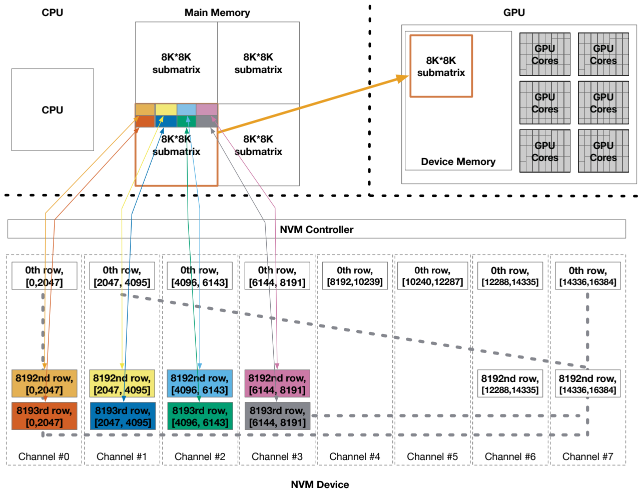


Figure 1: The blocked-matrix-multiplication datapath in a hardware-accelerated computing system having a conventional storage-system hierarchy with non-volatile memory (NVM)

2.1 Modern accelerator-based architectures

Figure 1 shows the key hardware components for processing high-dimensional datasets in a modern heterogeneous computer. Such a computer typically uses hardware accelerators (e.g., GPUs in Figure 1) to perform compute kernels on high-dimensional datasets. Modern GPUs provide two types of processing elements: (1) conventional GPU cores that operate on vectors and (2) TCUs that operate on matrices to more efficiently process high-dimensional datasets.

To provide high-performance data storage, a computer may use a solid-state drive (SSD) that stores data in non-volatile memory technologies such as NAND flash memory and phase-change memory (PCM). All non-volatile memory technologies have their own basic access granularities (e.g., page in flash memory). This is true for newer byte-addressable devices like PCM [90]. To improve bandwidth, modern SSD controllers use channel-level and bank-level parallelism. Modern SSDs typically organize their memory arrays into channels, with all parallel channels capable of accepting unique requests simultaneously. Each channel also contains multiple banks that allow free banks to accept requests while other banks are busy. Commercialized non-volatile memory technologies have limited program-erase cycles, so SSD controllers must carefully manage mapping between the logical addresses that software uses and physical addresses in memory.

Because modern storage systems still expose linear address spaces such as logical block addresses (LBAs) for use, producers of high-dimensional datasets must reduce data dimensionality to 1-dimensional representations, typically with row-oriented or column-oriented storage formats. If a hardware-accelerated compute kernel needs to compute on datasets, an application invoking the compute kernel must first fetch each piece of necessary data from the storage system. The application then assembles each received data chunk into memory objects with a layout that satisfies

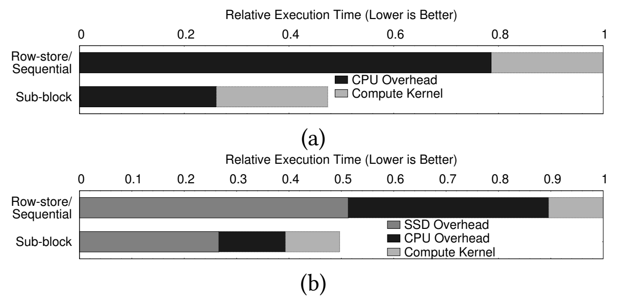


Figure 2: The relative execution time of matrix multiplications using row-store format (sequential) and sub-block format, with (a) data already in main memory and (b) from the SSD.

the demands of the target accelerator's architecture. Finally, the application copies the created memory object from the host main memory to the accelerator's device memory so compute kernels can use the object.

Figure 1 also shows matrix multiplication (MM) performed using a modern GPU. The GPU kernel delivers maximum computation throughput if the MM kernel is performed on 8K×8K submatrices. The origin dataset presents its matrix inputs using a row-oriented storage format, which is generally considered the most efficient for CPU-based compute kernels. Each raw input matrix is 16K×16K. The SSD has 8KB pages and 8 parallel channels; each SSD page stores 2K elements encoded in IEEE-754 32-bit floating-point format, and each matrix row occupies 8 pages that are logically consecutive in the SSD's LBA. In conventional SSD design, these consecutive pages are typically striped across different channels to enable sequential parallel accesses in LBAs; such striping is needed because most file systems and applications assume that underlying storage devices are more efficient when the devices perform accesses sequentially. As a result, each of the aforementioned 8 pages that belongs to a single row will be physically located in a distinct channel.

To maximize the efficiency on machines that have only limited high-speed memory capacity available for hardware accelerators, an application must create a pipeline to logically partition input matrices into submatrices and perform multiplications on pairs of submatrices. In each pipeline, the application first creates a submatrix by retrieving necessary elements from all required rows into a destination location in system main memory. Once all elements of a submatrix have arrived, the application copies the created submatrix to the accelerator's device memory and launches the compute kernel on the copied submatrix.

[P1]: The overhead of marshalling input data. To restructure stored data into the dimensionality hardware accelerators need, an application must use CPU instructions to calculate the mapping between the raw-data offset and the target memory locations. Then the application must issue I/O requests through the system software stack, again using CPU instructions, to fetch pieces of raw data and place (and potentially convert) the received data chunks in their designated memory locations. In Figure 1, if the application needs to fetch an 8K×8K submatrix, the row-store format will require the program to issue 8,192 I/O requests—to fetch 8,192 rows that contain the submatrix items.

Figure 2(a) illustrates the CPU overhead, with the raw data already in system main memory before the compute kernel launches. The system uses an AMD RyZen 3700X CPU and Nvidia's RTX 2080 GPU. The baseline comprises pipelined matrix multiplications—that is, the multiplication of two 32K×32K matrices using sub-blocks of 8K×8K matrices. As the source dataset uses sequential row-store format, the baseline needs an additional stage to form 8K×8K matrices before the compute kernel starts. In contrast, the alternate configuration with sub-block format already has 8K×8K submatrices stored in main memory and does not require the CPU to prepare data for compute kernels. In figure 2(a), the sequential baseline configuration requires 2.11× more time to avoid CPU overhead than the sub-block configuration does.

[P2]: Underutilization of interconnect bandwidth. As a result of the mismatching data layouts between storage and compute kernels, I/O requests to fetch data chunks are typically smaller than the sizes that can amortize the overhead of each I/O transaction, leading to underutilized interconnection bandwidth on the I/O side. In modern NVMe [6] interface, interconnect bandwidth saturates if each request is larger than 2 MB. However, when fetching a row containing 8K elements for an 8K×8K submatrix (Figure 1), each I/O request is for 32 KB of data, and the interconnect only achieves 66% of the peak bandwidth.

To maximize I/O bandwidth, an application may be designed to fetch consecutive chunks of storage data into a large memory buffer and gradually copy the buffered data into designated memory locations. However, the latter approach creates three performance issues: (1) it generates traffic from copying small data blocks on the CPU-memory bus, (2) it wastes precious main-memory capacity for memory buffers, and (3) it fetches data elements that the application might not use immediately and might need to fetch again due to limited buffer memory capacity and so wastes I/O bandwidth.

[P3]: Underutilization of device bandwidth. One-dimensional LBA space requires that an application materialize the original multi-dimensional data structure when presenting the data in storage. If the data dimensionality or the access pattern does not fit the storage device's internal structure, the application consuming the materialized data is likely to underutilize the device's internal bandwidth. As in Figure 1, when fetching 8K×8K submatrices, a program can only utilize 50% of the available parallel channels (device internal bandwidth) since the requesting data only reside in 4 of the 8 channels.

Figure 2(b) shows the impact of such underutilized bandwidth in a real application by extending the situation in Figure 2(a) for data placed in an SSD with 32 parallel channels. Besides the CPU overhead of transforming rows into submatrices, the baseline spends 1.92× more time fetching data compared to an SSD configuration with optimal data layout for the workload by storing 8K×8K submatrices consecutively due to the underutilized bandwidth.

2.2 Challenges

Given the observations in Section 2.1, the data layout of an input dataset from memory/storage must fulfill the following requirements to fundamentally address the locality and dimensionality

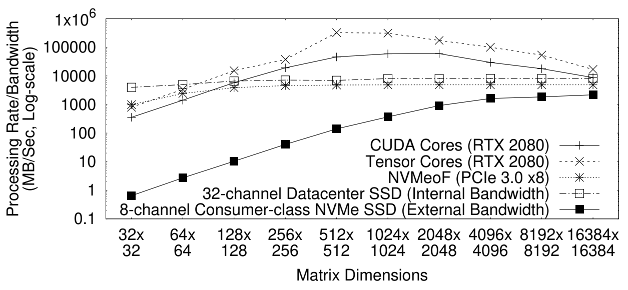


Figure 3: The effective data processing rates or I/O bandwidth of system components with various matrix sizes

mismatch between storage data and compute kernels on hardware accelerators: (1) The layout must minimize the overhead of restructuring data in compute kernels and hardware accelerators. (2) The layout must allow the application to fully utilize the internal bandwidth of the storage device. (3) The layout must maximize the granularity of I/O commands and thus minimize the number of I/O requests.

Under the linear address space that a conventional storage system exposes, designing an optimal data layout that satisfies the above three requirements is challenging, for the reasons given below.

[C1]: Unavailability of internal memory-device architecture to applications. A storage device abstracts memory locations into LBAs without informing applications about the device structure. Existing NVMe commands allow an application to query the parameters of an underlying device and to optimize the LBA layout for an individual device; however, garbage collection and wear-level functions in the SSD management layer (including the flash-translation layer [FTL] and PCM translation layers [33, 76, 99]) can lead to data-location shuffling and suboptimal performance.

In addition, the application is unable to optimize the LBA layout for all storage devices because internal designs and hardware structures differ among devices. Consider Figure 3, which shows the data-processing rate or I/O bandwidth of each hardware component. For two flash-based SSDs, one with 32 channels and the other with only 8 channels, the 32-channel SSD can utilize the maximum bandwidth with the 512×512 matrices that the application fetches (4 GB data sequentially from the SSD's LBA space), but the 8-channel SSD only achieves its own maximum bandwidth (again, with 4 GB data fetched sequentially). Figure 3 thus underscores the extreme difference in optimal data sizes due to different internal device architectures.

[C2]: Unpredictability of optimal dimensionality in compute kernels. Because any applications can share a dataset, and a compute kernel can execute on a variety of computing resources, there exists no optimal layout that can maximize the efficiency of all applications or accelerators. For example, a row-oriented or column-oriented pair-wise matrix can maximize GPU computation, but cannot be efficient for matrix-multiplication kernels.

Figure 3 also compares the data-processing rates and input-data sizes for processors/accelerators using different computation models—the vector processing model (used with general matrix multiply, GEMM, from Nvidia's cuBLAS [63] library) on conventional GPU cores (CUDA cores), and the 2-D matrix-processing model used with Nvidia's Tensor Cores. To measure pure compute-kernel performance, we made all matrices available on the GPU device memory before the GEMM functions were invoked. Setting aside the significant performance lead in Tensor Cores, the optimal submatrix size that maximizes performance on the CUDA cores was found to be 2048×2048, whereas the optimal submatrix size for the Tensor Cores was 512×512.

[C3]: Demand mismatch between storage devices and compute kernels. Even though a user can narrow down the use of a dataset to a specific type of compute kernel, hardware accelerator, and dedicated storage device, there is no guarantee that the optimal input data for the hardware accelerator will match the layout that can fully utilize the internal bandwidth of the storage device. For the situation covered in Figure 3, our matrix-multiplication kernel worked best on 512×512 submatrices, but the bandwidth for the consumer SSD is maximized if each I/O request fetches 16K×16K submatrices.

2.3 Alternatives

Prior work has addressed mismatching dimensionalities in modern computers through (1) more efficient data storage formats, (2) libraries supporting high-dimensional addressing, and (3) offloading of I/O operations to intelligent storage devices. Unfortunately, none of the three approaches tackles the challenges mentioned in Section 2.2.

File formats Much work has focused on dense data storage. Apache's Parquet [8] and ORC [9] offer efficient columnar storage formats, and Arvo [7] offers row stores for Hadoop. To reduce the CPU processing overhead of deserializing objects, Google's protocol buffers [87] and JSON's binary representation (BSON) [68] propose binary-encoded internal data representations. Albis [84] fuses features of columnar/row-major stores with binary-encoded formats to exploit both spatial locality and low-overhead deserialization. Domain-specific file formats are also available to address the demand of popular compute kernels in application subsets: aggregate genomic data (AGD) [19] for biological applications, G-Store [47] for graph applications, and JSOI [70] and ONNX [83] for machine learning models. Frameworks that automatically generate data layout for GPU kernels [51, 57] also exist. None of these formats can address challenges posed by **[C1]** and **[C3]**, leading to problems **[P2]** and **[P3]**.

High-dimensional software I/O libraries To reduce the overhead in applications that create high-dimensional data objects, cloud storage systems such as N5 [73] and Zarr [5] use n-dimensional addressing modes and chunk raw data to better utilize I/O bandwidth. Zarr [5] also uses data compression to maximize the effective I/O bandwidth. These software libraries still only access abstracted linear memory/storage address spaces, which leads to problems **[P2]** and **[P3]**.

In-storage processing (ISP) To present data objects in a way that compute kernels require and improves the bandwidth/memory demand and CPU overhead, near-data processing (NDP) or ISP models can use a device's internal bandwidth by working on the controller within the device. Existing ISP frameworks can transform rawdata into application objects for general-purpose compute kernels [85], mixed-precision/approximate computing workloads [35] or specialized applications like large graphs [58]. However, the above approaches are all lacking in that ISP cannot efficiently produce

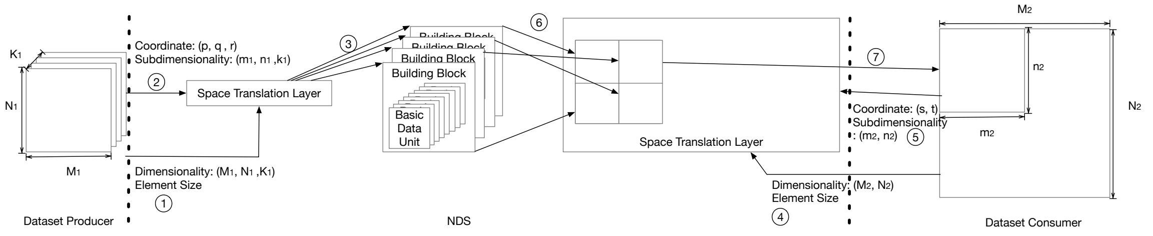


Figure 4: An overview of NDS data-access operations

application objects by fully utilizing interconnect bandwidth (**[C3]**) unless the in-device data layout fits the access patterns of ISP code.

3 OVERVIEW OF NDS

To address the issues and challenges outlined in Section 2.1 and Section 2.2, NDS follows three key design principles.

Provide commands in multi-dimensional addressing modes. As noted, linear address space forces applications to transform multi-dimensional data into one-dimensional data and leads to **[P1]**, **[P2]** and **[P3]**. In contrast, NDS offers multi-dimensional storage and I/O commands that enable a single I/O request to manage data in arbitrary dimensions. NDS thus minimizes the number of I/O requests while maximizing the data volume that each request moves.

Indicate compute-kernel demand through application-defined, multi-dimensional address spaces. In NDS, each application can define its own view of a data object's dimensionality, regardless of (1) the original data layout and dimensionality in data storage and (2) the view of the data object from other applications, to address **[C2]**. NDS automatically and implicitly transforms a data object into an application's required dimensionality. As an application accesses data objects using its natural view of the address space, NDS further reduces the conversion overhead of data dimensionality (**[P1]**), and the compute kernel can work more efficiently because NDS presents data using an optimized memory layout.

Make software agnostic to storage-device characteristics. NDS decouples storage-device granularity and data layout from an application's view, thereby reducing programming complexity and broadening data-layout applicability, addressing **[C1]**, **[C2]** and **[C3]**. NDS uses the dimensionality that a dataset producer provides to help determine storage space; NDS intelligently restructures storage data into *building blocks*—collections of basic access units (e.g., pages) in the memory device. These building blocks serve as internal structures that maximize internal access bandwidth and allow low-overhead conversion into different dimensionalities.

Figure 4 shows, on a conceptual level, how NDS presents an architecture to support multi-dimensional data accesses. By working between the software stack and storage hardware, NDS receives optimal access granularities and serves as an intermediary to fill the demands of both application and device. In NDS, each in-storage and in-application address space is determined by three properties:

- *Space identifier*: the identifier of the target address space, typically the starting address of the space in the linear view of the storage/memory
- *Element size*: the volume of each data element in the space

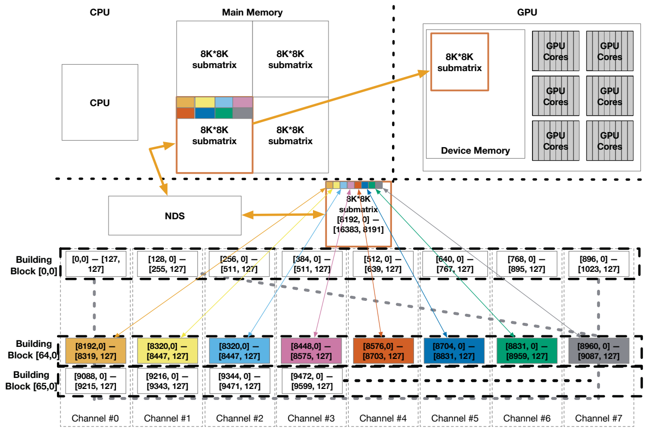


Figure 5: Accessing a submatrix from NDS

- *Dimensionality*: the number of dimensions and the size of each dimension, combined

Following the numbering system in Figure 4, an NDS dataset producer creates a multi-dimensional space using the three essential properties (①). The STL then parses the data structure, determines the dimensionality of building blocks in the address space, and returns the space identifier to the software. When transferring data into NDS (②), an application passes the source memory location and the address-space data position using two parameters:

- *Coordinate*: the position of each data chunk within the defined address space
- *Sub-dimensionality*: the dimensionality of each coordinate of a fixed-size partition within the defined address space

Next, the STL fetches data from the source location and uses a set of building blocks to store data (③). The STL assigns memory locations for each data portion to maximize access performance. The high-level allocation-policy guideline is to find the minimum overlap of basic access units from all available forms of parallelism in the device. When an application needs to consume data from NDS, the application notifies NDS of its view of the dimensionality (④) and requests the data-chunk coordinate and associated sub-dimensionality that the application will use for its data location (⑤). (Note that the dimensionality in the consumer program need not match the dimensionality of the dataset producer's address space, as long as the volumes of these two dimensionalities match.) The STL will then transform the coordinate, the sub-dimensionality, and the dimensionality to locate and fetch building blocks (⑥). Finally, NDS assembles the fetched building blocks in a structure aligned with the consumer's view and delivers the assembled object into the consumer's address space (⑦).

Figure 5 revisits the request shown in Figure 1 using NDS. In Figure 5, the producer application creates a three-dimensional space with a size of 8,192×8,192×4. After gauging the device capabilities and the dimensionality of the space, NDS uses 16,384 building blocks, where each building block is sized as 128×128, for a total of 8 pages from 8 different channels in the same bank. In this way, NDS maximizes the access bandwidth during the store operation.

For an application that treats the dataset in the space as four 8,192×8,192 sub-blocks and requests the [1,0] sub-block—that is, elements [8192, 0] through [16383, 8191] in a 16,384×16,384 matrix—the consumer simply needs to send one I/O command describing the location of the sub-block in the space. NDS translates this single request into accesses to 4,096 (64×64) building blocks. As each building block consists of pages from different channels in the same bank, each access to the building block can fully utilize the internal bandwidth of the device. NDS can issue an access request to a building block in another bank in the next cycle to pipeline building-block accesses, whereas the non-NDS approach in Figure 1 consistently wastes approximately 50% of the internal parallelism when accesses are fully pipelined.

Upon receiving each building block, NDS dynamically assembles the datasets into 8,192×8,192 sub-matrices that the compute kernel requires. NDS delivers each consecutive data chunk to the host computer as soon as the chunk is assembled. The application does not need additional code to restructure the object because the data are now presented to the compute kernel in the kernel's required layout. The complete process requires only one I/O command from the host computer. In contrast, the process in Figure 1 requires the host computer to either intensively access the CPU memory bus and use CPU instructions (to relocate data) or issue an excessive number of I/O requests (e.g., in the thousands).

4 THE STL

The STL is the core of NDS. The STL creates multi-dimensional address spaces and determines how building blocks are used. The STL receives access requests using coordinates in arbitrary dimensionalities and translates these requests into hardware commands to access the physical data locations of building blocks. Using building blocks, the STL dynamically transforms and presents the address space in the desired application view. In contrast, conventional FTLs and other block abstractions only present data in linear addressing modes and rely on applications to adjust data dimensionality.

4.1 Building blocks

In NDS, a building block is a fixed-size logical chunk of data storage. NDS considers a building block as the basic unit of data-storage elements that are colocated in their original address space. A building block contains a set of physical memory-access units (e.g., pages in NAND flash-based storage). The STL determines the size of a building block and assigns memory units into a building block by considering the efficiency of accessing the building block.

To minimize the latency of its retrieval, a complete building block stores its data in units available through all parallel channels. Therefore, STL determines the address space of a building block by finding the minimum building-block size (BB_Size_{min}) allowable for the underlying storage device as

$$BB\_Size_{min} = Max_{Number\ of\ Parallel\ Requests} \times Granularity_{Basic\ Access} \quad (1)$$

where $Max_{Number\ of\ Parallel\ Requests}$ represents the maximum number of parallel requests a memory device can perform (typically the number of parallel device channels), and $Granularity_{Basic\ Access}$ represents the fine-grained structure of the aforementioned basic-access units. For example, if an SSD contains flash chips having 4 KB pages and 8 parallel channels, the building block will be defined as a multiple of 32 KB (4 KB × 8) using Equation 1. A minimum building block will consist of 8 pages, each from a different parallel channel in the device. BB_Size_{min} is identical for devices that share the same internal architecture but can vary among devices with different architectures.

The STL creates a multi-dimensional address space using dimensionality parameters as well as a mapping between multi-dimensional coordinates and building blocks. Since $Max_{Number\ of\ Parallel\ Requests}$ and $Granularity_{Basic\ Access}$ can represent two dimensions of the building block in a modern NVM storage device, the STL uses each building block to store a two-dimensional sub-block if the space has at least two dimensions. To balance the unpredictable demands of access patterns from different compute kernels, the STL maintains equal-size sub-block dimensions whenever possible. The STL will therefore determine the building-block size, BB, of the address space as

$$BB = N \times (2^{\lceil \frac{log_2 \frac{BB\_Size_{min}}{N}}{2} \rceil})^2 \quad (2)$$

where N is the size of each element in the space, and with each dimension in the building block storing $2^{\lceil \frac{log_2 \frac{BB\_Size_{min}}{N}}{2} \rceil}$ elements. For example, when BB_Size_{min} is 32 KB, and the application creates a 2-D space to store 4-byte elements, the STL will use 64 KB as the size of each building block. Each building block will contain 2 pages from each channel and store 128 elements in each dimension.

If the address space has more than 2 dimensions, the STL can use banks to construct a 3-D sub-cube as a building block. In this case, the STL determines the minimum 3-D building-block size as

$$3D\_BB\_Size_{min} = BB\_Size_{min} \times Num_{banks} \quad (3)$$

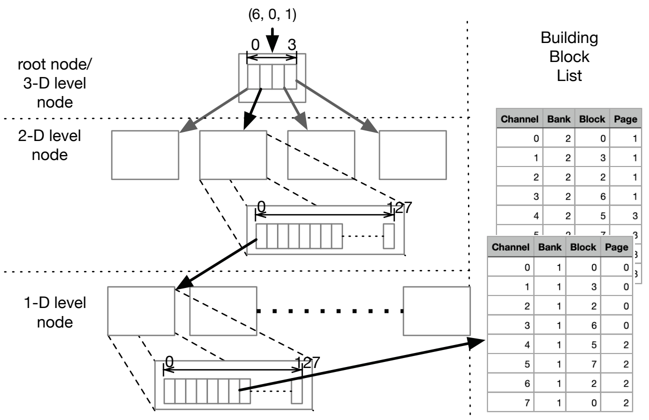
where Num_{banks} is the number of banks. Similarly, the STL determines building block size for a 3-D space as

$$BB = N \times (2^{\lceil \frac{log_2 \frac{BB\_Size_{min}}{N}}{3} \rceil})^3 \quad (4)$$

where each dimension stores $2^{\lceil \frac{log_2 \frac{BB\_Size_{min}}{N}}{3} \rceil}$ elements.

If the memory device provides another level of parallelism, NDS can further extend Equation 3 to make a building block. Note, however, that because modern volatile/non-volatile devices only exploit bank-level and channel-level parallelism, and hardware accelerators only support 1-D or 2-D operations (e.g., TPUs [39] and Tensor Cores), NDS as yet supports only 1-D, 2-D, or 3-D building blocks.

In the rest of this paper, we use (bb_1, bb_2, ..., bb_n) to represent the dimensionality of a building block in an n-dimensional space where bb_i represents the size of the ith-order dimension in the



Channel	Bank	Block	Page
0	2	0	1
1	2	3	1
2	2	2	1
3	2	6	1
4	2	5	3
5	2	7	3
			3
			3

Channel	Bank	Block	Page
0	1	0	0
1	1	3	0
2	1	2	0
3	1	6	0
4	1	5	2
5	1	7	2
6	1	2	2
7	1	0	2

Figure 6: An exemplary B-tree structure of STL

space. Since NDS currently supports 1-D to 3-D building blocks, NDS sets the bb_i value to 1 when $i > 3$. We use $(d_1, d_2, ..., d_n)$ to represent the size of an n-dimensional (N-D) space where d_i stands for the size of the ith-order dimension.

4.2 Locating and allocating building blocks

An NDS is based on building blocks, so the STL maintains a B-tree structure for each N-D space to locate a building block. The STL maintains an N-level tree data structure for an N-D space. In each B-tree, the root node corresponds to the highest order in the N-D space, the second-level from the root node corresponds to the second-highest order in the space, and the leaf node corresponds to the lowest order of the space. In each level of B-tree nodes, the node degree is $\frac{d_i}{bb_i}$, where d_i represents the size of the ith order spatial dimension. For a non-leaf node, each entry is a pointer to a next-level leaf node in the STL's memory space. For a leaf node, each entry points to a list of physical memory locations for basic access units (e.g., pages in an SSD) that belong to the corresponding building block. The list of access-unit locations is sorted according to the sequential order of the units in the building block.

Figure 6 illustrates such a B-tree structure for a 3-D space with 2-D building blocks; the multidimensional space is an (8192, 8192, 4) space that uses (128, 128) building blocks. Since the space has three dimensions, the B-tree has three levels. If a request goes to a building block with coordinate (6, 0, 1), the tree will visit the 1st entry in the root node to reach the next level and use the 0th entry in the 2-D node to reach the 1-D/leaf node. The leaf node points to a list of pages in storage. When accessing a building block, the STL will issue requests to fetch all pages in the list in parallel.

When a valid request leads to an unallocated entry in the B-tree structure, the request reaches an unallocated physical memory location. The STL will allocate all necessary tree nodes along the traversal path from the STL's memory space. If the request tries to overwrite an existing access unit in a building block, the STL simply picks a page from the same channel and bank as the overwritten unit.

If the STL attempts to reach an unallocated-entry leaf node, the STL must find a free physical memory location that maximizes parallelism when accessing the associated building block in the device's data arrays. In a typical NVM storage device, the STL can take several different approaches to access-unit selection, as follows: (1) If the STL has not created the building block, the STL randomly chooses an access unit from a channel and a bank. Alternatively, (2) if the building block exists, the STL picks an access unit from a parallel channel that the building block uses the least (a *least-used* channel); in this case, the unit is from the same bank as the most recently allocated unit in the building block. (3) The STL can also select an access unit from an unused or least-used bank if the building block has used a unit from every channel in the bank. (4) If the STL cannot find a page through the above rules because the building block has used a unit from every channel and bank, the STL chooses one of the least-used banks and repeats (1) through (3).

Note that if the number of free units for any combination of channel and bank is lower than a specified threshold (typically 10%), the STL triggers garbage collection to reclaim invalidated memory locations. Garbage collection in NDS is similar to that of a conventional NVM storage device, except that NDS can maintain a reverse lookup table that records the building blocks associated with the erasing unit (i.e., a block in flash SSD). The look data structure can use 8 bytes of the spare out-of-band area for each access unit to speeds up mapping updates between building blocks and the physical memory locations.

4.3 The space translator

NDS can present data locations to an application as coordinates in an arbitrarily dimensioned space, and so decouples the application's view of dimensionality from actual storage. The STL's *space translator* makes this possible by dynamically remapping coordinates to the building blocks in a designated address space.

As Section 3 describes, an application can work with its own multi-dimensional space of size $\delta_1, \delta_2, ..., \delta_m$ regardless of that space's representation in storage. With this m-dimensional space, an application can access NDS using a coordinate $(x_1, x_2, ..., x_m)$ and the sub-dimensionality $(\beta_1, \beta_2, ..., \beta_m)$ that represents the partition of the requested data.

For an n-dimensional space in NDS with dimension sizes $(d_1, d_2, ..., d_n)$ and each building block having dimensions $(bb_1, bb_2, ..., bb_n)$, the STL will remap the data request to a set of building blocks, with each building block having the n-dimensional coordinates $(y_0, ..., y_n)$, where each y_i belongs to a set of numbers in Y_i, and Y_i is defined and calculated dynamically as

$$Y_i = \{a \in \mathbb{Z} : a \geq \lfloor \frac{\sum_{j=2}^{m}[(x_j \times \beta_j) \prod_{k=1}^{j-1} \delta_k] + x_1 \times \beta_1}{\prod_{k=1}^{i-1} bb_i} \rfloor$$
$$\textit{and } a \leq \lfloor \frac{\sum_{j=2}^{m}\{[(x_j + 1) \times \beta_j)] \prod_{k=1}^{j-1} \delta_k\} + (x_1 + 1) \times \beta_1}{\prod_{k=1}^{i-1} bb_i} \rfloor\} \quad (5)$$

4.4 Data accesses, assembly, and composition

While the space translator decomposes a request into building-block coordinate accesses, the STL walks through the

B-tree structure to issue requests (and allocate space if necessary) to each access unit, as described in Section 4.2.

For a write request, the STL fetches the writing partition from the application and fills corresponding data chunks into one or more building blocks. When an application's sub-dimensionality is larger than a building block, the fetched partition contains content for several building blocks that are spatially colocated. The STL can easily optimize the sequence of programming storage arrays by writing to building blocks that the fetched partition covers. If the fetched partition is smaller than a building block, the STL will try to keep the partition in STL memory space and write to storage whenever the collected data is sufficient for a basic access unit in any building block.

For a read request, the STL will access a set of building blocks and creates a buffer in STL's memory space to place the received data into logical locations—from the application's perspective—with object assembly determined by the translation process described in Section 4.3. As soon as a segment of the assembled object reaches the optimal data-exchange volume for the system interconnect, NDS starts to move the assembled data. Once the STL has assembled all data for an application request and has moved the assembled multi-dimensional object to application's memory space, NDS designates the request as complete and returns to processing other application requests.

5 THE NDS PROTOTYPE

In this section, we describe a proof-of-concept storage system that demonstrates the validity and value of NDS. This prototypical NDS system includes an application programming interface (API) and implements the core of NDS—the STL functions. To fundamentally address the issues and challenges in Section 2, we implemented a hardware-assisted NDS and a software-only design for comparison by extending a baseline SSD.

5.1 APIs

Because NDS accepts high-dimensional coordinates instead of conventional 1-D offsets, NDS needs system-level API functions for front-end interactions. Our API allows applications, file systems, and programming-language libraries to access user-space and kernel-space functions that fall into the three categories below.

Space creation/management API functions in this category receive arguments that describe dimensionality parameters such as the number of dimensions, the size of each dimension, and the size of each element. If a function call passes null as the address identifier, then the function considers the call a request to create a new address space and will trigger the STL to determine the building-block size, create corresponding data structures, and allocate/return an identifier for the space. If the caller passes an existing valid address identifier, NDS triggers the STL to expand, shrink, or restructure the existing space. When necessary, the callee can return a different address/identifier.

Open/close Functions in this category do not perform real data accesses; rather, they hand over the application view of the address space to NDS and terminate application use of a space. These functions resemble the `open` and `close` calls in conventional systems. Conventional systems can, in fact, leverage these open/close API functions to extend existing functions and use NDS more efficiently.

Read/write API Functions in this category receive the following arguments: (1) a space identifier describing the source/target NDS space, (2) coordinates describing the source/target data locations from the application's perspective, and (3) a memory buffer in the application space for the source data or target. If a function writes to NDS, NDS triggers the STL to allocate building blocks and transfer the source data from the specified coordinates in the application to the designated locations in building blocks. If a function reads from NDS, the STL translates the coordinates from the application's perspective to the corresponding locations in the storage device. The read/write API functions can also work with multi-dimensional data-movement API functions (e.g., `cudaMemcpy2D` in CUDA) to move data more efficiently between high-dimensional accelerators and NDS.

Without NDS, the programmer needs to manually optimize applications in a way that the program can retrieve and store data chunks using an optimized size that maximizes the storage bandwidth for the specific storage device. Most of time, such chunk size does not match the chunk size maximizes the GPU utilization as Section 2.2 presented. Therefore, the programmer needs to add code to construct application objects in chuck sizes that maximize the performance of compute kernels, and finally, the compute kernel receives these objects can work on parts of the constructed data objects. In addition to the efforts of coding itself, programmers need to go through exhaustive design space exploration processes in searching the optimal size for both the storage front-end and the compute kernel backend. In contrast, the NDS version of code allows the programmer to just focus on describing the desired objects from the compute kernel's perspective. NDS transparently handles the performance optimizations in storage front-end. Though the programmer may still need to search for optimal parameters of the compute kernel, NDS should at least save half of the development time since the programmer does not need to take care of the optimal object and parameters for storage.

5.2 System implementations

To test the NDS concept, we developed a hardware-assisted NDS by extending the baseline SSD's controller and replacing the existing NVM management layer with the STL. In addition, we implemented a software-only NDS by using LightNVM [15], a storage protocol/interface that exposes software to the physical addresses of the underlying NVM device.

Figure 7 shows the system architectures, control paths, and data paths for the three implementations: the baseline SSD, the software-only NDS, and the hardware-assisted NDS. In applications involving conventional SSDs, like the configuration shown in Figure 7(a), the system stack incurs significant data accesses to system main memory in order to exchange data objects for compute kernels (①), serialize/deserialize objects (②), and exchange the serialized objects with the device (③–⑤).

In the software configuration shown in Figure 7(b), all NDS functions, including the API implementations and the STL, run on the host processor with system main memory. The NDS subsystem

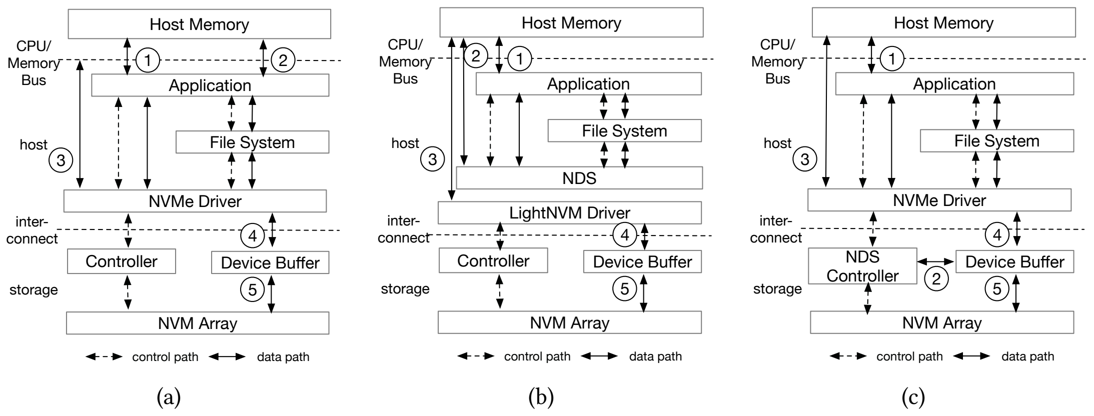


Figure 7: (a) The baseline, conventional SSD, (b) the software-only NDS implementation, and (c) the hardware-assisted, NVMe-based NDS implementation

interacts with the LightNVM-compliant storage device (e.g., open-channel SSD) through the NVMe device driver; NDS does this by using physical addresses that the storage device can directly use to access its internal data-array locations without translation. Though the software-only implementation can address all the challenges of the baseline case and avoids the object deserialization/restructuring overhead, the implementation fails to address other performance issues. Namely, the software-only implementation still accesses the host system main memory for data assembly(②), so the implementation under-utilizes the internal bandwidth because NDS still works behind the interconnect.

In contrast to the baseline system and the software-only system, the hardware-assisted NDS system moves data-assembly traffic to device memory (②). Because NDS works inside the device in the hardware-assisted system, NDS has access to the full internal bandwidth and can reduce the interconnect traffic in (④).

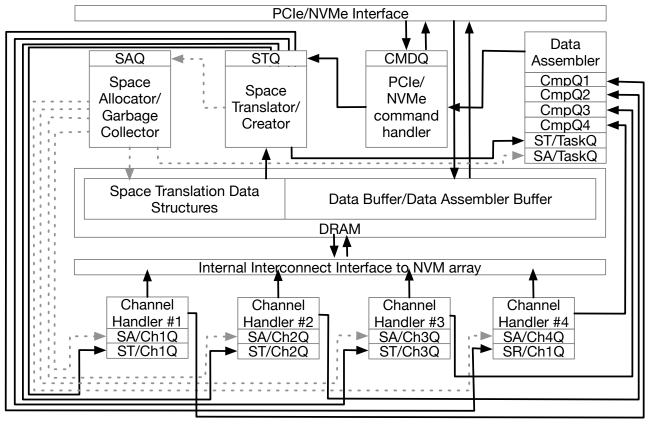


Figure 8: The NDS-compliant SSD controller

5.3 The NDS-compliant storage device

Our hardware-assisted, NDS-compliant storage device provides an interface that supports NDS’s multi-dimensional address mode and a controller to implement the STL functions. This section describes the extended command set and the controller architecture in our prototype implementation.

5.3.1 PCIe/NVMe command extension. Our prototype NDS-compliant storage supports an extended NVMe command set while remaining compatible with existing NVMe commands [6]. An extended NVMe command uses a reserved bit in the first 64-bit command word in NVMe standard to distinguish itself from conventional NVMe commands. Upon receiving a conventional NVMe command, NDS simply treats the request as a request to a one-dimensional address space.

For read/write operations, the extended NVMe commands are almost identical to conventional read/write commands except that the NDS version uses the second 64-bit command word. This command word points to a memory page that contains the coordinates and sub-dimensionality from the application’s perspective. In a system with 4 KB memory pages, each extended command can support coordinates up to 32 dimensions and 2^{64} elements in each dimension.

The NDS/NVMe command extension has three commands for managing multi-dimensional address space: `open_space`, `close_space`, and `delete_space`. The `open_space` command can create a new space or change the dimensionality of an existing space depending on the flag set in the command header. The second `open_space` command word points to a memory page that lists the dimensionality of the space, with up to 32 dimensions and 2^{64} elements in each dimension. The `open_space` command returns a 64-bit identifier and a *dynamic space ID*; the software system can use the space ID to distinguish between different views an application uses for the space. In contrast, the `close_space` command reclaims the dynamic space ID and disables the use of the previously defined space view. The `delete_space` command permanently deletes an address space by invalidating all space building blocks and removing the translation data structures for the space.

5.3.2 The controller architecture. To support the STL features, our prototype NDS storage device has a controller that extends an existing NVMe/SSD controller, as shown in Figure 8. Similar to conventional NVMe controllers, the NDS controller exploits pipeline parallelism to maximize command-handling throughput. The NDS controller contains the following pipeline elements: (1) a PCIe/NVMe command handler, (2) a space translator/manager, (3) a space allocator with garbage collector, (4) a data assembler, and (5) 4× channel handlers.

To communicate with one another, the NDS controller's pipeline elements use a message-passing interface with dedicated message-queue pairs between each neighboring element to avoid locking and race conditions. The NDS controller can use DRAM within the storage device as data-structure storage for space/address translations and data buffering. The controller also contains a DMA engine to move data between the host computer and the device DRAM or the device DRAM and the device's NVM arrays.

To build our prototype NDS controller, we extended the firmware code in the baseline SSD controller using ARM A72 cores with each pipeline element statically mapped to one of eight cores. While the NVMe baseline controller also uses eight cores to implement essential functions, the baseline controller replaces the STL creator/translator with an address-lookup function and replaces the data assembler with a command-control manager. Both the baseline and NDS controller have the same amount of channel handlers.

5.3.3 Supporting cryptography. Modern computer systems provide cryptography features in software modules and hardware accelerators to protect sensitive information. Modern datacenter-class SSD controllers [60] typically incorporate intellectual property cores to provide high-throughput data encryption/decryption. The most popular standard block-based advanced encryption mechanisms (AES) [62] work by dividing data into fixed-sized sections and performing pseudorandom permutation within the same section of data. The resulting data size remains the same before and after encryption/decryption.

NDS can easily cooperate with popular block-based encryption mechanisms. Though NDS translates coordinates between abstracted memory spaces, divides datasets into building blocks and constructs building blocks into application objects, NDS does not alter the content of datasets in very fine grains. Therefore, the current NDS workflow functions well regardless of where the system performs cryptography functions, as long as the data size in each dimension of the building block is larger than the section size of encryption. As the section size is simply 256 bits that can store 8× 4-byte elements, and each page in modern memory chips is at least 4KB, the cases where the encryption section size is larger than the dimension size of a building block is near zero.

5.3.4 Supporting Data Compression. Modern storage subsystems may implement data compression/decompression to reduce the cost of storage. NDS can work with or integrate data compression/decompression features in different ways.

If the storage device transparently performs data compression functions, the hardware-assisted NDS can work with existing data compression features if (1) the data compression/decompression process occurs before the space allocation stage and (2) the data compression/decompression occurs in units of building blocks. As NDS's space allocation policy in Section 4.2 randomly chooses access units from channels/banks, NDS can still ensure performance and even-wearing, but simply uses fewer access units for each building block.

In case the system compresses/decompresses data in software or using accelerators on the host computer, the data compression mechanism needs to be part of the software-only NDS framework. As software-only NDS decides/maintains the building block sizes and space allocation using host system software stack, software-only NDS can use this information to treat each building block as a basic unit of data compression/decompression and allow NDS to function correctly.

6 EXPERIMENTAL METHODOLOGY

To evaluate the built NDS systems, we modified the I/O functions of a set of applications while allowing the applications' multi-dimensional compute kernels to remain the same. This section describes the system configurations and experimental setup we developed.

6.1 Experimental platform

We used an octa-core AMD RyZen 3700X processor with a clock rate up to 4.4 GHz. We installed Ubuntu 16.04 (Linux kernel version 4.15), implemented the NDS API and NDS subsystem (for the software-only version) and an extended NVMe driver (for the hardware-assisted version) to support the NDS model. We built a TLC-NAND flash-based SSD with the controller described in Section 5.

Our prototype SSD has 32 parallel channels with 4 KB pages in 8 banks. The total capacity of the SSD is 2 TB, with 10% over-provisioning space reserved for background garbage collection. The prototype SSD also has 4 GB of DRAM buffer available for all FTL/STL data structures and data buffers. The host machine for our prototype has 32 GB of main memory with a motherboard containing a PCIe 3.0 I/O hub that connects the processor and other peripherals; these peripherals include a Mellanox InfiniBand NIC with 8 PCIe 3.0 lanes to connect to the prototype SSD through the NVMeoF protocol. The host machine also contains an NVIDIA RTX 2080 GPU with 8 GB of device memory to enable compute-intensive kernels.

6.2 Benchmarks

As shown in Table 1, we used 10 applications to test our NDS system, with the applications falling into 6 categories: graph (traversal), linear algebra, physics simulation, data mining, image processing, and tensor-algebra operations. We selected the applications because each one (1) provides a highly efficient open-source implementation (or one that is easily optimized) for large datasets, (2) works on multi-dimensional datasets, and (3) provides or allows for the generation of datasets that exceed the capacity of GPU device memory. As the data volume of each workload is larger than the device-memory buffer on the GPU (where we execute compute kernels), the compute kernels execute algorithms in a block fashion and must restructure input data into sub-blocks prior to data processing. Each application is pipelined so that its I/O and data restructuring (if required) overlap with the I/O and data restructuring of the compute kernels.

Workload Name	Category	Dimensionality		Data Size (Elements)	Kernel Sub-dimension size	Source of Baseline
		Data	Kernel			
Breadth-First Search (BFS)	Graph Traversal	2D	1D	65536×65536	65536	Rodinia [71]
Bellman-Ford (SSSP)	Graph Traversal		2D		65536×4096	Parallel Implementation of Bellman Ford Algorithm [82]
Block-GEMM (GEMM)	Linear Algebra	2D	2D	65536×65536	8192×8192	MSplitGEMM with cuBLAS using Tensor Cores [63, 94]
Hotspot (Hotspot)	Physics Simulation	2D	2D	65536×65536	4096×4096	Rodinia [36, 71]
K-Means (KMeans)	Data Mining	2D	1D	65536×65536	65536	Rodinia [71], Parallel K-Means Data Clustering [29, 54]
K-Nearest Neighbor (KNN)	Data Mining		1D		65536	Rodinia [71], knn-CUDA [27, 88]
PageRank (PageRank)	Graph	2D	2D	65536×65536	4096×65536	GraphBlast [91], GraphChi [48]
2D Convolution (Conv2D)	Image Processing	2D	2D	65536×65536	4096×4096	CUDA Separable Convolution [67]
Tensor Times Vector (TTV)	Tensor Algebra	3D	2D/1D	2048×2048×2048	512×512	NVIDIA [23, 77]
Tensor Contractions (TC)	Tensor Algebra		2D		512×512	NVIDIA [23, 77]

Table 1: Workloads, the sources of their baseline implementations, the dimensionality, size of their raw data and the dimensionality of their compute kernels.

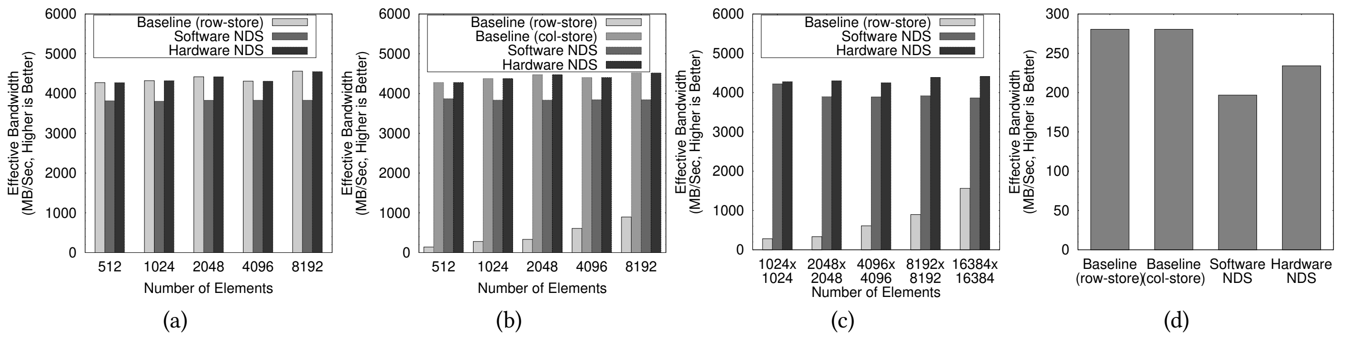


Figure 9: The performance of the baseline SSD, the software-only NDS, and the hardware-assisted NDS for fetching data with different dimensionalities

For the baseline implementation, we carefully partitioned the I/O size and selected the sub-block for compute kernels to minimize end-to-end latency. For the applications used to test the implementations, we chose the parameters of each application's compute kernel to agree with the dataset format in the host main memory so that the dataset was ready for the compute kernel to consume regardless of the underlying NDS implementation. With the size of each application dataset exceeding host main-memory capacity, the applications required intensive I/O that was always the longest pipeline stage in the baseline scenario.

Among the applications used to test our NDS system, 3 pairs of applications shared their inputs: BFS and SSSP, K-Means and KNN, and TTV and TC. The application compute kernels were allowed to vary in block size and dimension to demonstrate the elasticity of NDS in accommodating different application demands with identical datasets in NDS. Each run of any workload's baseline implementation lasts longer than 1 minute (BFS) and up to 40 minutes (KMeans).

7 RESULTS

This section summarizes our evaluation of NDS and the nature of the 5.73× speedup observed for the hardware-assisted NDS implementation.

7.1 Microbenchmarks

To understand the pure I/O performance of NDS, we created a set of microbenchmarks with raw data comprising a 32,768×32,768 2-D matrix and a storage device with 32 channels and 4 KB pages. The prototype NDS system selected 256×256 as the building-block size for dataset storage, with elements in double floating-point format. Figure 9 shows the effective bandwidth measured from the application side in fetching and structuring the data.

In Figure 9(a), the application requests data in dimensions ranging from 512×32,768 to 4096×32,768 until the application finishes reading the whole matrix (basically a row-fetch from matrices of different dimensions). As the data are already in row order, the microbenchmark with the baseline SSD achieves an effective bandwidth of around 4.3 GB/sec. The hardware-assisted NDS achieves a performance that is almost identical to that of the baseline without optimizing building-block dimensions for the access pattern. These results indicate that the determined building-block structure effectively permits the hardware NDS to fully utilize a device's internal bandwidth and thereby cover the overhead of constructing rows from those building blocks. In contrast, the software-only NDS creates significant overhead by copying 256 elements (2 KB; smaller than a host DRAM page size) in each operation involved in constructing a row from related building blocks. As a result, the software NDS's effective bandwidth was observed to be only 3.8 GB/sec.

In Figure 9(b), the application performs column accesses for matrices with dimensions ranging from 32,768×512 to 32,768×4096. For the baseline/row-store condition, if the dataset is not presented optimally in the baseline SSD, then the effective bandwidth of fetching a column is at most 600 MB/sec, with the largest granularity appearing when the system cache is allowed to serve later requests without visiting the SSD. On the other hand, NDS still works efficiently in constructing a column for each access, and the resulting performance is comparable to storing column-ordered data in the baseline SSD (the baseline/column-store condition).

In Figure 9(c), the application fetches submatrices of various sizes. Because NDS's building blocks match the 2-D space the application requests, NDS significantly outperforms the baseline SSD, regardless of NDS implementation type. For an application whose data is stored in the baseline SSD, the application must generate many small I/O requests, with each request fetching a row from the baseline SSD; when accessing content for a submatrix, this process underutilizes both the interconnect and the device's internal bandwidth.

Figure 9(d) shows the performance of writing the microbenchmark 32,768×32,768 2-D data matrix into the baseline SSD and NDS. For the baseline SSD, data is arranged in both row-store and column-store formats before writing. For NDS, the dataset is arranged in row-oriented 2-D dense-matrix format before writing to NDS (using the NDS API). For these experiments, we disable asynchronous writes, so latency is measured until the end of the whole programming process. The baseline SSD has an effective write bandwidth of 281 MB/sec for both row-store or column-store formats.

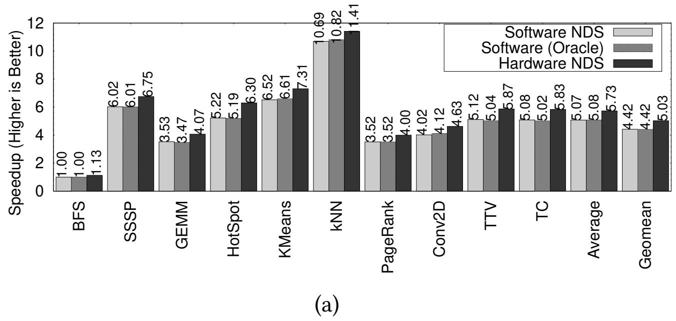
For the software-only NDS, the NDS subsystem requires the CPU code to dynamically break up large matrices into building blocks, and each building block must copy 256 elements (2 KB) for 256 times, which creates intensive memory operations with low bandwidth utilization on the host computer. Consequently, the effective write bandwidth of the software-only NDS is 30% slower than that of the baseline.

For hardware NDS, the SSD requests host main memory content in 4 KB pages and breaks them up later, allowing the to better utilize the host main memory bus and system interconnect. However, the increased overhead in the storage controller along with the controller's lower performance (compared to the host processor), resulted in an observed write performance loss of 17%. That being said, modern data-center workloads are more read-intensive than write-intensive, so the pronounced benefits of reading in NDS make the 17% loss acceptable for most workloads. Moreover, because NDS is compatible with conventional NVMe devices, write-intensive workloads can still be managed effectively via NDS's conventional storage capabilities.

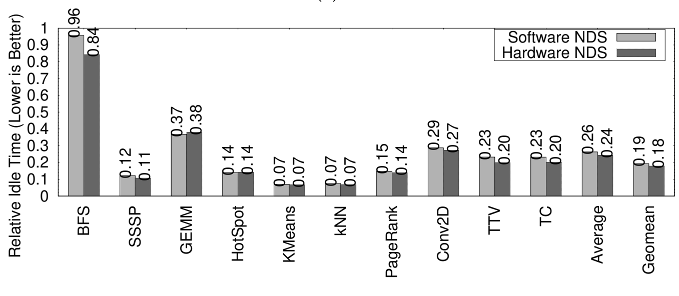
7.2 End-to-end application latency

Figure 10(a) shows the speedup of end-to-end latency of running applications. The software-only implementation can achieve a speedup of 5.07×, which shows the benefits of using NDS building blocks to store data and dynamically rebuild objects. Using building blocks, software-only implementation allows the NDS software to speed up the process of building multi-dimensional objects by 1.52× on average. With building blocks reducing the chances of fetching temporally unnecessary data from the SSD, software NDS also reduces 74% of average idle time before each pipelined compute kernel as Figure 10(b) shows.

An alternative to software NDS is using a software library and carefully layout data on the storage device. To investigate the maximum potential of all software-based approaches, we created an



(a)



(b)

Figure 10: (a) The speedup of end-to-end latency and (b) the reduction of idle time in compute kernels of running applications using NDS

oracle configuration where we exhaustively search for the best storage data layout that incurs zero overhead on the host and minimum end-to-end latency when executing these workloads. In this configuration, we have to store two copies of data in different shapes for workloads sharing the same datasets (i.e., BFS and SSSP, KMeans and KNN, TTV and TC). Figure 10(a) shows that even assuming these software libraries have zero overhead, the performance gain is just about the same as the software NDS.

Compared with software-only solution, the hardware NDS can further accelerate applications by 5.73×. As Section 5.2 explains, the hardware-assisted implementation removes both computation overhead and traffic on the host processor/memory needed to rebuild data objects from building blocks. Hardware NDS completely skips the process of assembling multi-dimensional objects on the host computer. The hardware NDS also allows the STL to fully access a storage device's internal parallelism. Even though the NDS controller is less powerful than the host processor, the hardware NDS still outperforms the software-only solution by 1.13×. As a result, hardware NDS reduces the idle time before compute kernels by 76%. Within these test applications, we found that BFS receives almost no benefit from the software-only NDS; although the original dataset was created and stored in 2-D building blocks, the compute kernel relies on sequential access along each row in the 2-D graph representation. As the baseline SSD version stores data in row-ordered format, the compute kernel works efficiently with the storage format.

The software-only NDS shows that building blocks work well with mismatched access patterns; accessing matrices in row-order format (discussed in Section 7.1, above) produces similar results. In contrast, the hardware-only NDS outperforms the baseline version because the hardware-only version (1) passes objects to the application that exactly match compute-kernel demands and (2) accesses building blocks with more bandwidth than any host-side software can provide.

Though hardware NDS's 1.13× speedup over software NDS seems limited, we argue the presence of hardware NDS is meaningful for the following reasons. (1) Software NDS relies on lightNVM [15] that is currently still not widely adopted by manufacturers. (2) Software NDS increases the CPU workload and makes it less preferable by heavily loaded servers. (3) As software NDS always requires the host-side module to perform space translation and address lookup, software NDS makes the use of modern system interconnects' peer-to-peer data exchange between storage devices and hardware accelerators [4, 55, 65, 85, 95] inefficient. (4) Hardware NDS allows STL to work closer with NVM array using the rich internal bandwidth to handle NDS tasks. Our baseline SSD has an internal-to-external bandwidth ratio as 8-to-5. With faster NVM technologies that raise the internal-to-external bandwidth ratio, the advantage of hardware NDS will become more significant.

7.3 Overhead of NDS

As the STL requires more complex data structures and arithmetic operations in handling requests, NDS increases the latency and creates space overhead in storage devices, but to a very limited degree. We evaluated the worst-case scenario where a request only asks for a page of data from the baseline SSD and NDS. All requests to the baseline and NDS are carefully designed to avoid any transformation to help to identify the increases of data access latency from B-tree traversal. The result shows $41\mu s$ additional latency in software NDS and $17\mu s$ in hardware NDS; both are shorter than or the same order as the average latency of accessing a modern NAND flash page (typically $30\mu s$–$100\mu s$) [31, 61]. However, as a leaf node in NDS's B-tree structure can point to up to 512 flash pages, if the request asks for a larger block of data, NDS simply requires one B-tree traversal for all accesses and can easily amortize the additional latency. In the worst-case scenario where every page is in-use, the whole STL lookup data structure occupies 0.1% of the storage space.

8 OTHER RELATED WORK

In addition to the related work described in Section 2.3, several other lines of NDS-relevant research deserve mention.

Tensor algebra libraries, algorithms, compilers, and accelerators For decades, tensor algebra has been explored through algorithms [10, 13, 20, 42, 59, 92], libraries [14, 56, 78, 86], code generators [24, 44, 52, 79], and accelerators [2, 28, 30, 34, 49, 66, 69, 80, 81, 98, 100, 101]. Most prior work has focused on improving the efficiency of tensor computations. In contrast, NDS offers a streamlined compute-kernel front-end to address the bottleneck caused by data transfer/restructuring.

Other in-storage processing approaches The hardware NDS is similar to in-storage processing in that NDS extends the storage controller to dynamically assemble data from building blocks. Aside from the projects mentioned in Section 2.3 that use ISP/NDP to directly present data as applications require, existing general-purpose ISP/NDP platforms can enable object deserialization functions [3, 26, 32, 46, 74, 96, 97]. Specialized ISP/NDP systems can also run compute kernels on storage controllers, thereby accessing the rich internal device bandwidth to accelerate file system operations [22] and data analytics [40, 41, 89]. As noted previously, however, the data layout may not align with in-storage compute-kernel access patterns, in which case in-storage applications may perform inefficiently even though the internal bandwidth is accessible to code/accelerators in the storage device.

Sparse formats The Tensor Algebra Compiler (TACO) [45] can generate efficient code based on iteration graphs, merge lattices, and a tensor storage tree for both sparse and dense matrices. One work [25] demonstrates that a sparse tensor-algebra compiler should be agnostic to data layouts [11, 17, 37, 43, 72]. Among data representations, the compressed sparse-block (CSB) format [18] suggests that building blocks may be equally effective for both row-wise and column-wise sparse-matrix processing. NDS focuses more on dense formats because the data-processing throughput of compute kernels on dense datasets is significantly higher and NDS's storage demands are greater. Nonetheless, NDS can store sparse content efficiently through a checking/optimization process that is similar to page-zero optimization in VAX/VMS [50].

Smart main memory controllers Without a storage system like NDS to present data in a way that aligns with a compute kernel's perspective, the problem [P2] will significantly bottleneck application performance for various access patterns. Both Impulse and Gather-Scatter DRAM (GS-DRAM) proposed smart memory controllers or adding additional circuits that adds another layer of main memory address translation and dynamically creates condensed application objects without redundant elements/values going through the CPU-main memory bus [21, 75]. However, both Impulse and GS-DRAM still lead to [P3] as the internal page data layout is still either row or column oriented. RC-NVM [53] further confirms that a dual-addressing mode is unrealistic with DRAM architectures. In contrast, NDS can release the burden of Impulse or GS-DRAM and ultimately address both [P2] and [P3] without the presence of Impulse or GS-DRAM if the raw data comes from the storage subsystem. NDS also needs zero modifications in NVM chips as RC-NVM.

9 CONCLUSION

This paper introduces a memory/storage system called NDS and describes prototype NDS implementations. NDS provides multi-dimensional address spaces for applications and decouples storage dimensionality from application-optimal dataset dimensionality by dynamically reconstructing data objects. NDS successfully tackles the challenges posed by hidden device parameters, the unpredictability of application kernels, and dimensional mismatches among devices. NDS thus addresses the overhead of restructuring input data and the underutilization of both interconnect bandwidth and device bandwidth. Through prototype evaluation, we show that the hardware-assisted NDS version achieves an average 5.73× speedup over a datacenter-class SSD baseline for a representative set of real-world applications.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments. We would like to thank Fazil Osman and his team from Broadcom Inc. for their support in developing the NDS prototype. This work was sponsored by the two National Science Foundation (NSF) awards, CNS-1940048 and CNS-2007124. This work was also supported by new faculty start-up funds from University of California, Riverside. We also owe a debt of gratitude to Christopher Fraser for his excellent copyediting skills.

A ARTIFACT APPENDIX

A.1 Abstract

This document describes the artifact of "NDS: N-Dimensional Storage" and the process of reproducing the experimental results in this paper. To run the baseline applications that this paper evaluates, the evaluator must have a computer equipped with (1) an NVIDIA GPU and (2) an Infiniband network interface card (NIC) and (3) capable of running a Linux distribution supporting the software stacks for the GPU, the NIC and NVMe over Fabrics. To build a prototype storage device that supports NDS in hardware, the evaluator must also have a prototype similar to Broadcom's SST-100 or PS1100R platform [16] where the platform contains (1) a multi-core ARM processor that is capable of running the NVMe controller software stack and the firmware programs that NDS requires, and (2) PCIe slot(s) that can host flash-based storage medium supporting NVMe or LightNVM protocols. If the evaluator installs multiple flash-based storage devices in the prototype, the resulting platform will be able to emulate the performance of datacenter SSDs in a way that the ARM-based controller has access to rich internal bandwidth through many (typically more than 16×) parallel channels.

As the paper describes, we present NDS in two different types of implementations. Though the software-only version can work on conventional SSDs, we still recommend running experiments on the same hardware as hardware-assisted NDS for fair comparisons.

A.2 Artifact check-list (meta-information)

- **Program:** The NDS software stack, Breadth-First Search (BFS) [71], Bellman-Ford (SSSP) [82], MSplitGEMM with cuBLAS using Tensor Cores (GEMM) [63, 94], Hotspot [71], K-Means [29], K-Nearest Neighbor [88], PageRank [91], 2D Convolution (Conv2D) [67], Tensor Times Vector (TTV) [23, 77] and Tensor Contractions (TC) [23, 77].
- **Compilation:** NVCC 10.2, GCC 5.4.0
- **Data set:** Synthetic datasets from each program's dataset generator. We also provide those in our GitHub repository [93].
- **Run-time environment:** Ubuntu 16.04, SPDK v19.07.1, CUDA 10.2
- **Hardware:** A host computer with an x86 processor, an NVIDIA GPU using Turing architecture or more advanced, a Mellanox RDMA-compliment 40 Gbps network interface card, and a Broadcom Stingray based prototyping system hosting NVMe SSDs [16].
- **Execution:** To reduce the disturbance from other workloads, we recommend running experiments with a sole user. Each experiment can run from 1 minute to 10 minutes, depending on the dataset size.
- **Metrics:** End-to-end latency (secs) and throughput (MB/sec).
- **Output:** Each benchmark program will display its execution result through console or log files. We added timestamps in their standard output formats for measurement purposes.
- **Publicly available?:** Yes
- **Archived DOI:** https://doi.org/10.5281/zenodo.5495743
- **Code licenses (if publicly available)?:** We will be using MIT license for our code.
- **Data licenses (if publicly available)?:** The datasets are publicly available through their original licensing terms.

A.3 Description

A.3.1 How to access. We archive the source code and workloads at https://doi.org/10.5281/zenodo.5495743 For the latest version, you can access our GitHub page: https://github.com/escalab/NDS

A.3.2 Hardware dependencies. To build a host machine supporting NDS, the user will need the following hardware components.

- **Processor:** An x86 processor to host required software stack.
- **DRAM:** The system should have at least 16 GB of DRAM to support the execution of GPU programs.
- **GPU:** A CUDA-compatible NVIDIA GPU. However, we strongly recommend the user to have high-end GPUs with Tensor Cores based on Turing or Ampere architecture to re-produce the I/O bottleneck in workloads.
- **Network Interface Card:** A Mellanox RDMA-compliment network interface card supports 40 Gbps connection speed.
- **Motherboard:** The host machine's motherboard should contain at least two 8-lane to 16-lane PCIe 3.0 slots to allow the NIC at its full speed while letting the GPU enjoy a sufficient amount of I/O bandwidth.

To build a prototype hardware-assisted NDS-compliant SSD, the user will need the following hardware components.

- **SoC:** An embedded-system-class processor based on ARM 64-bit architectures with the capability of handling RDMA/NVMe over Fabrics protocols.
- **DRAM:** The embedded system platform should have at least 4 GB of DRAM to support basic STL functions of NDS and data buffering.
- **I/O Slots:** The prototype board should have multiple I/O slots (e.g., PCIe) for flash-based storage devices (e.g., LightNVM or NVMe SSDs) to communicate with the SoC.

A.3.3 Software dependencies. The baseline storage system stack and NDS rely on the following software components.

- RDMA kernel modules and libibverbs
- SPDK v19.07.1 [38]
- CUDA 10.2 [64] and cuBLAS [63]

A.3.4 Data sets. An evaluator can find the dataset generator for each benchmark application in https://github.com/escalab/NDS/tree/main/data/generator/

We briefly describe these data generators and their workflows.

- **Matrix** The matrix data generator produces random numbers as many as $M \times N$, where M and N are the dimensions of the target matrix. The resulting matrix can serve as an input of Block-GEMM, conv2D and hotspot. The evaluator can find the matrix data generator at `data/generator/matrix/` from the repository. The same directory provides a `datagen.sh` script file to invoke the program and produce the matrix efficiently in binary-encoded format accordingly.
- **Tensor** The tensor generator produces random numbers as many as $M \times N \times K$, where M, N and K are dimensions of a 3-D tensor. The resulting 3-D tensor can provide an input for TTV and TC. The evaluator can find the tensor data generator at `data/generator/tensor` from the repository. The same directory provides a `datagen.sh` script file to invoke the program and produce the tensor in efficient binary-encoded format accordingly.
- **Clustering** This dataset generator leverages the source code of kNN-CUDA [88] to produce inputs for K-Means and K-NN. The evaluator can find the tensor data generator at `data/generator/clustering` from the repository. The same directory provides a `datagen.sh` script file to invoke the program. The script produces a binary-encoded file containing M data points with N attributes and another binary-encoded file containing K points with N attributes, depending on the user inputs of M, N and K.
- **Graph** The data generator leverages Rodinia's data generator for BFS benchmark [71]. The main difference between NDS's and Rodinia's version is that the NDS version creates graphs in binary-encoded adjacency matrices, but the Rodinia version stores the matrix in less efficient ASCII format. The resulting matrices can provide inputs for BFS and SSSP in our workloads. The modified data generator is located at `data/generator/graph/bfs` from the repository. The same directory provides a `gen_dataset.sh` script file to invoke the program. The user should specify the desired number of nodes M and the number of edges N. The generator will create an $M \times M$ adjacency matrix with N non-zero random values in the matrix.
- **Pagerank** This paper modifies the dataset generator from the 10th DIMACS Implementation Challenge [12]. Similar to our modifications to other dataset generators, we have made the data in binary format to reduce the host processing overhead. The modified data generator is located at `data/generator/graph/pagerank` from the repository. The same directory provides a `pagerank_graph_gen.sh` to download the graph with 65536 nodes from the website and transform it into a binary adjacency matrix.

A.4 Installation

Before installing any NDS software/library, the user should install packages described in Section A.3.3 appropriately. The system should also configure the RDMA connection between the host computer and the NDS prototype SSD to at least 40Gbps.

Then, the user can install the NDS software stack through the following steps.

```
git clone https://github.com/escalab/NDS
cd NDS
make all
```

This step will generate static libraries for NDS applications to link at compile time.

A.5 Experiment workflow

A.5.1 Running Baseline Applications. Before running the baseline, the evaluator needs to configure the prototype SSD and the host system software stack as a conventional NVMe SSD by using the script `NDS/scripts/setup_baseline.sh`.
Once the configuration is done, the user can enter a directory with the prefix of `NDS/apps/baseline_`. For example, if the user would like to run an experiment for GEMM, the user should use the following a sequence of commands.

```
cd NDS/apps/baseline_block_gemm
make
./run 65536 8192
```

The evaluator can also run all baseline experiments once by executing the following instructions

```
cd NDS/scripts
./run_baseline.sh
```

A.5.2 Running Software NDS apps. Before running any software NDS experiments, the evaluator needs to configure the prototype SSD and the host system software stack for software NDS by using the script `NDS/scripts/setup_software_NDS.sh`.
Once the configuration is done, the user can enter a directory with the prefix of `NDS/apps/software_`. For example, if the user would like to run an experiment for GEMM, the user should use the following sequence of commands.

```
cd NDS/apps/software_block_gemm
make
./run 65536 8192
```

The evaluator can also run all software-only NDS experiments once by executing following instructions

```
cd NDS/scripts
./run_software_nds.sh
```

A.5.3 Hardware-assisted NDS. Before running any hardware-assisted NDS experiments, the evaluator needs to configure the prototype SSD and the host system software stack for software NDS by using the script `NDS/scripts/setup_hardware_NDS.sh`.
Once the configuration is done, the user can enter a directory with the prefix of `NDS/apps/hardware_`. For example, if the user would like to run an experiment for GEMM, the user should use the following sequence of commands.

```
cd NDS/apps/hardware_block_gemm
make
./run 65536 8192
```

The evaluator can also run all hardware-assisted NDS experiments once by executing following instructions

```
cd NDS/scripts
./run_hardware_nds.sh
```

A.6 Evaluation and expected results

A.6.1 Evaluate Results. To ease the process of reproducing the complete set of NDS experiments, we provide a script that executes all our experiments once and analyzes results through `results.py`

```
cd NDS/scripts
./run_evaluation.sh
```

The evaluator can also redirect the outputs to three log files, baseline.txt, software_nds.txt and hardware_nds.txt, and carefully examine the timestamps.

A.6.2 Expected Results. Compared to baseline, sequential format on the traditional storage device, Software NDS can offer 5.07× speedup, while Hardware NDS is able to give 5.73× speedup. Please reference Section 7 of our paper for the expected results.

REFERENCES

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-Stores vs. Row-Stores: How Different Are They Really?. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 967–980. https://doi.org/10.1145/1376616.1376712

[2] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, et al. 2020. Think fast: a tensor streaming processor (TSP) for accelerating deep learning workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 145–158.

[3] Anurag Acharya, Mustafa Uysal, and Joel Saltz. 1998. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS VIII)*. ACM, New York, NY, USA, 81–91. https://doi.org/10.1145/291069.291026

[4] Advanced Micro Devices, Inc. 2014. FirePro DirectGMA Technical Overview. http://developer.amd.com/tools-and-sdks/graphics-development/firepro-sdk/firepro-directgma-sdk/.

[5] Francesc Alted, Martin Durant, Stephan Hoyer, John Kirkham, Alistair Miles, Mamy Ratsimbazafy, Matthew Rocklin, Vincent Schut, Anthony Scopatz, and Prakhar Goel. [n. d.]. Zarr. https://github.com/zarr-developers/zarr-python

[6] Amber Huffman. 2012. NVM Express Revision 1.1. http://nvmexpress.org/wp-content/uploads/2013/05/NVM_Express_1_1.pdf.

[7] Apache Software Foundation. [n. d.]. Apache Avro(TM) 1.10.2 Documentation. https://avro.apache.org/docs/current/

[8] Apache Software Foundation. [n. d.]. Apache Parquet. https://parquet.apache.org/

[9] Apache Software Foundation. [n. d.]. The smallest, fastest columnar storage for Hadoop workloads. https://orc.apache.org/

[10] Woody Austin, Grey Ballard, and Tamara G Kolda. 2016. Parallel tensor compression for large-scale scientific data. In *2016 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 912–922.

[11] Brett W Bader and Tamara G Kolda. 2008. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing* 30, 1 (2008), 205–231.

[12] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner (Eds.). 2013. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings.* Contemporary Mathematics, Vol. 588. American Mathematical Society. https://doi.org/10.1090/conm/588

[13] Muthu Baskaran, Tom Henretty, Benoit Pradelle, M Harper Langston, David Bruns-Smith, James Ezick, and Richard Lethin. 2017. Memory-efficient parallel tensor decompositions. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[14] Gerald Baumgartner, Alexander Auer, David E Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert J Harrison, So Hirata, Sriram Krishnamoorthy, et al. 2005. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proc. IEEE* 93, 2 (2005), 276–292.

[15] Matias Bjørling, Javier González, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies* (Santa clara, CA, USA) *(FAST'17)*. USENIX Association, USA, 359–373.

[16] Broadcom Inc. 2019. White Paper: NVMe over Fabrics Performance Stingray(TM)-Based Storage Appliance. https://docs.broadcom.com/doc/broadcom-stingray-100G-NVMe-oF-performance.

[17] Aydin Buluç and John R Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–11.

[18] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures* (Calgary, AB, Canada) *(SPAA '09)*. Association for Computing Machinery, New York, NY, USA, 233–244. https://doi.org/10.1145/1583991.1584053

[19] Stuart Byma, Sam Whitlock, Laura Flueratoru, Ethan Tseng, Christos Kozyrakis, Edouard Bugnion, and James Larus. 2017. Persona: A High-Performance Bioinformatics Framework. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 153–165. https://www.usenix.org/conference/atc17/technical-sessions/presentation/byma

[20] Jonathon Cai, Muthu Baskaran, Benoît Meister, and Richard Lethin. 2015. Optimization of symmetric tensor computations. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[21] J. Carter, W. Hsieh, L. Stoller, M. Swanson, Lixin Zhang, E. Brunvand, A. Davis, Chen-Chi Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. 1999. Impulse: building a smarter memory controller. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*. 70–79. https://doi.org/10.1109/HPCA.1999.744334

[22] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. 2010. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 385–395. https://doi.org/10.1109/MICRO.2010.33

[23] C Cecka. 2018. Pro Tip: cuBLAS Strided Batched Matrix Multiply. https://developer.nvidia.com/blog/cublas-strided-batched-matrix-multiply/

[24] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[25] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276493

[26] 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* (New York, New York, USA) *(SIGMOD '13)*. ACM, New York, NY, USA, 1221–1230. https://doi.org/10.1145/2463676.2465295

[27] Vincent Garcia, Eric Debreuve, Frank Nielsen, and Michel Barlaud. 2010. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *2010 IEEE International Conference on Image Processing*. IEEE, 3757–3760.

[28] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. 2020. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 922–936.

[29] Serban Giuroiu. 2012. CUDA K-Means Data Clustering. https://github.com/serban/kmeans

[30] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 151?V165. https://doi.org/10.1145/3352460.3358291

[31] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. 2009. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. Association for Computing Machinery, New York, NY, USA, 24–33. https://doi.org/10.1145/1669112.1669118

[32] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A Framework for Near-data Processing of Big Data Workloads. *SIGARCH Comput. Archit. News* 44, 3 (June 2016), 153–165. https://doi.org/10.1145/3007787.3001154

[33] N. Hajinazar, P. Patel, M. Patel, K. Kanellopoulos, S. Ghose, R. Ausavarungnirun, G. F. Oliveira, J. Appavoo, V. Seshadri, and O. Mutlu. 2020. The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 1050–1063.

[34] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. ExTensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual*

IEEE/ACM International Symposium on Microarchitecture. 319–333.

[35] Yu-Ching Hu, Murtuza Taher Lokhandwala, Te I, and Hung-Wei Tseng. 2019. Dynamic Multi-Resolution Data Storage. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2019)*.

[36] Wei Huang, Shougata Ghosh, Sivakumar Velusamy, Karthik Sankaranarayanan, Kevin Skadron, and Mircea R Stan. 2006. HotSpot: A compact thermal modeling methodology for early-stage VLSI design. *IEEE Transactions on very large scale integration (VLSI) systems* 14, 5 (2006), 501–513.

[37] Eun-Jin Im and Katherine Yelick. 1998. Model-based memory hierarchy optimizations for sparse matrices. In *Workshop on Profile and Feedback-Directed Compilation*, Vol. 139.

[38] Intel. 2015. Storage Performance Development Kit. https://spdk.io/doc/

[39] 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* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[40] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/2749469.2750412

[41] Yangwook Kang, Yang-Suk Kee, Ethan L. Miller, and Chanik Park. 2013. Enabling cost-effective data processing with smart SSD. In *Mass Storage Systems and Technologies (MSST)*.

[42] Oguz Kaya and Bora Uçar. 2016. High performance parallel algorithms for the Tucker decomposition of sparse tensors. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 103–112.

[43] David R Kincaid, Thomas C Oppe, and David M Young. 1989. ITPACKV 2D user's guide. Report CNA-232. *Center for Numerical Analysis, University of Texas at Austin, Austin, TX, USA* (1989).

[44] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor algebra compilation with workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 180–192.

[45] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[46] Gunjae Koo, Kiran Kumar Matam, Te I, Hema Venkata Krishna Giri Narra, Jing Li, Steven Swanson, Hung-Wei Tseng, and Murali Annavaram. 2017. Summarizer: Trading Bandwidth with Computing Near Storage. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2017)*.

[47] Pradeep Kumar and H Howie Huang. 2016. G-Store: High-performance graph store for trillion-edge processing. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 830–841.

[48] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 31–46.

[49] Sangwon Lee, Gyuyoung Park, and Myoungsoo Jung. 2020. TensorPRAM: Designing a Scalable Heterogeneous Deep Learning Accelerator with Byte-addressable PRAMs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020, July 13-14, 2020*, Anirudh Badam and Vijay Chidambaram (Eds.). USENIX Association. https://www.usenix.org/conference/hotstorage20/presentation/lee

[50] Levy and Lipman. 1982. Virtual Memory Management in the VAX/VMS Operating System. *Computer* 15, 3 (1982), 35–41. https://doi.org/10.1109/MC.1982.1653971

[51] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou. 2016. Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 633–644. https://doi.org/10.1109/SC.2016.53

[52] Jiajia Li, Casey Battaglino, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. 2015. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[53] Shuo Li, Nong Xiao, Peng Wang, Guangyu Sun, Xiaoyang Wang, Yiran Chen, Hai Helen Li, Jason Cong, and Tao Zhang. 2019. RC-NVM: Dual-Addressing Non-Volatile Memory Architecture Supporting Both Row and Column Memory Accesses. *IEEE Trans. Comput.* 68, 2 (2019), 239–254. https://doi.org/10.1109/TC.2018.2868368

[54] Wei-Kang Liao. 2003. Parallel K-Means Data Clustering. http://users.eecs.northwestern.edu/~wkliao/Kmeans/index.html

[55] Yang Liu, Hung-Wei Tseng, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. 2016. Hippogriff: Efficiently moving data in heterogeneous computing systems. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 376–379. https://doi.org/10.1109/ICCD.2016.7753307

[56] Tze Meng Low, Robert A Van de Geijn, and FLAME Working Note. 2004. *An API for manipulating matrices stored by blocks.* Computer Science Department, University of Texas at Austin.

[57] Deepak Majeti, Kuldeep S. Meel, Rajkishore Barik, and Vivek Sarkar. 2016. Automatic Data Layout Generation and Kernel Mapping for CPU+GPU Architectures. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 240–250. https://doi.org/10.1145/2892208.2892210

[58] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: graph semantics aware SSD. In *Proceedings of the 46th International Symposium on Computer Architecture*. 116–128.

[59] Devin A Matthews. 2018. High-performance tensor contraction without transposition. *SIAM Journal on Scientific Computing* 40, 1 (2018), C1–C24.

[60] Microchip Technology Inc. 2020. Flashtec NVMe Controllers. https://www.microsemi.com/product-directory/storage/3687-flashtec-nvme-controllers.

[61] Micron. 2015. MT29F32G08CBADAWP Datasheet. https://www.micron.com/parts/nand-flash/mass-storage/mt29f32g08cbadawp?pc=%7B80EFFAAD-26CB-4D06-84BC-0E3274B960A9%7D

[62] National Institute of Standards and Technology. 2021. Cryptographic Standards and Guidelines. https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development.

[63] NVIDIA. 2019. cuBLAS. https://docs.nvidia.com/cuda/cublas/index.html.

[64] NVIDIA Corporation. 2014. CUDA C Programming Guide v6.0. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[65] NVIDIA Corporation. 2014. Developing a Linux Kernel Module Using RDMA for GPUDirect. http://docs.nvidia.com/cuda/pdf/GPUDirect_RDMA.pdf.

[66] 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. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 27–40.

[67] Victor Podlozhnyuk. 2007. Image convolution with CUDA. *NVIDIA Corporation white paper, June* 2097, 3 (2007).

[68] Srđan Popić, Dražen Pezer, Bojan Mrazovac, and Nikola Teslić. 2016. Performance evaluation of using Protocol Buffers in the Internet of Things communication. In *2016 International Conference on Smart Systems and Technologies (SST)*. 261–265. https://doi.org/10.1109/SST.2016.7765670

[69] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 58–70.

[70] Horacio Hoyos Rodriguez and Beatriz Sanchez Piña. 2019. JSOI: A JSON-Based Interchange Format for Efficient Model Management. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 259–266. https://doi.org/10.1109/MODELS-C.2019.00041

[71] M. Boyer S. Che, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '09)*. 44–54.

[72] Yousef Saad. 2003. *Iterative methods for sparse linear systems.* SIAM.

[73] Stephan Saalfeld. [n. d.]. N5. https://github.com/saalfeldlab/n5

[74] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 67–80. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/seshadri

[75] Vivek Seshadri, Thomas Mullins, Amirali Boroumand, Onur Mutlu, Phillip B Gibbons, Michael A. Kozuch, and Todd C Mowry. 2015. Gather-Scatter DRAM: In-DRAM address translation to improve the spatial locality of non-unit strided accesses. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 267–280. https://doi.org/10.1145/2830772.2830820

[76] Z. Shao, N. Chang, and N. Dutt. 2012. PTL: PCM Translation Layer. In *2012 IEEE Computer Society Annual Symposium on VLSI*. 380–385. https://doi.org/10.1109/ISVLSI.2012.75

[77] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka. 2016. Tensor Contractions with Extended BLAS Kernels on CPU and GPU. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. 193–202. https://doi.org/10.1109/HiPC.2016.031

[78] Edgar Solomonik, Devin Matthews, Jeff R Hammond, John F Stanton, and James Demmel. 2014. A massively parallel tensor contraction framework for coupled-cluster computations. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3176–3190.

[79] Paul Springer and Paolo Bientinesi. 2018. Design of a high-performance GEMM-like tensor–tensor multiplication. *ACM Transactions on Mathematical Software (TOMS)* 44, 3 (2018), 1–29.

[80] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[81] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 689–702.

[82] Qian Sun. 2018. Parallel Implementation of Bellman Ford Algorithm. https://github.com/sunnlo/BellmanFord

[83] The Linux Foundation. [n. d.]. Open Neural Network Exchange – The open standard for machine learning interoperability. https://onnx.ai/

[84] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schüpbach, and Bernard Metzler. 2018. Albis: High-Performance File Format for Big Data Systems. In *USENIX Annual Technical Conference*.

[85] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2016. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 53–65. https://doi.org/10.1109/ISCA.2016.15

[86] Field G Van Zee, Ernie Chan, Robert A Van de Geijn, Enrique S Quintana-Orti, and Gregorio Quintana-Orti. 2009. The libflame library for dense matrix computations. *Computing in science & engineering* 11, 6 (2009), 56–63.

[87] Kenton Varda. 2008. *Protocol Buffers: Google's Data Interchange Format*. Technical Report. http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html

[88] Michel Barlaud Vincent Garcia, Éric Debreuve. 2018. kNN-CUDA. https://github.com/vincentfpgarcia/kNN-CUDA

[89] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading. *Proc. VLDB Endow.* 7, 11 (July 2014), 963–974. https://doi.org/10.14778/2732967.2732972

[90] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2019. Towards an Unwritten Contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotstorage19/presentation/wu-kan

[91] Carl Yang, Aydin Buluç, and John D. Owens. 2019. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *CoRR* abs/1908.01407 (2019). arXiv:1908.01407 http://arxiv.org/abs/1908.01407

[92] Fan Yang, Fanhua Shang, Yuzhen Huang, James Cheng, Jinfeng Li, Yunjian Zhao, and Ruihao Zhao. 2017. LFTF: A framework for efficient tensor analytics at scale. *Proceedings of the VLDB Endowment* 10, 7 (2017), 745–756.

[93] Yu-Chia Liu and Hung-Wei Tseng. 2021. N-Dimensional Storage. https://anonymous.4open.science/r/NDS-CA87/.

[94] Zach Zimmerman. 2016. MSplitGEMM: Large matrix multiplication in CUDA. https://github.com/zpzim/MSplitGEMM.

[95] Jie Zhang, David Donofrio, John Shalf, Mahmut Kandemir, and Myoungsoo Jung. 2015. NVMMU: A Non-Volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures. In *The 24th International Conference on Parallel Architectures and Compilation Techniques (PACT 2015)*.

[96] Jie Zhang and Myoungsoo Jung. 2018. Flashabacus: A Self-governing Flash-based Accelerator for Low-power Systems. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. ACM, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/3190508.3190544

[97] J. Zhang, G. Park, D. Donofrio, J. Shalf, and M. Jung. 2020. DRAM-Less: Hardware Acceleration of Data Processing with New Memory. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 287–302. https://doi.org/10.1109/HPCA47549.2020.00032

[98] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. SpArch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.

[99] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. 14–23.

[100] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 15–28.

[101] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 359–371.

[102] Marcin Zukowski, Niels Nes, and Peter Boncz. 2008. DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing. In *Proceedings of the 4th International Workshop on Data Management on New Hardware* (Vancouver, Canada) *(DaMoN '08)*. Association for Computing Machinery, New York, NY, USA, 47–54. https://doi.org/10.1145/1457150.1457160