Dynamic Multi-Resolution Data Storage

Yu-Ching Hu
University of California, Riverside
yhu130@ucr.edu

Murtuza Taher Lokhandwala*
North Carolina State University
mlokhan@ncsu.edu

Te I*
Google Inc.
tei@google.com

Hung-Wei Tseng
University of California, Riverside
htseng@ucr.edu

ABSTRACT

Approximate computing that works on less precise data leads to significant performance gains and energy-cost reductions for compute kernels. However, without leveraging the full-stack design of computer systems, modern computer architectures undermine the potential of approximate computing.

In this paper, we present Varifocal Storage, a dynamic multi-resolution storage system that tackles challenges in performance, quality, flexibility and cost for computer systems supporting diverse application demands. Varifocal Storage dynamically adjusts the dataset resolution within a storage device, thereby mitigating the performance bottleneck of exchanging/preparing data for approximate compute kernels. Varifocal Storage introduces Autofocus and iFilter mechanisms to provide quality control inside the storage device and make programs more adaptive to diverse datasets. Varifocal Storage also offers flexible, efficient support for approximate and exact computing without exceeding the costs of conventional storage systems by (1) saving the raw dataset in the storage device, and (2) targeting operators that complement the power of existing SSD controllers to dynamically generate lower-resolution datasets.

We evaluate the performance of Varifocal Storage by running applications on a heterogeneous computer with our prototype SSD. The results show that Varifocal Storage can speed up data resolution adjustments by 2.02× or 1.74× without programmer input. Compared to conventional approximate-computing architectures, Varifocal Storage speeds up the overall execution time by 1.52×.

KEYWORDS

Approximate Computing, Heterogeneous Computer Architectures/Systems, In-Storage Processing, Intelligent Storage Systems, Near-Data Processing

ACM Reference Format:
Yu-Ching Hu, Murtuza Taher Lokhandwala, Te I, and Hung-Wei Tseng. 2019. Dynamic Multi-Resolution Data Storage . In *Proceedings of The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3352460.3358282

*Murtuza Taher Lokhandwala and Te I contributed to this research when they were students at North Carolina State University

1 INTRODUCTION

Approximate computing is gaining traction in commercialized systems because many applications can now tolerate small errors in input data [12, 19, 44, 47, 51, 53, 58, 94]. By receiving fewer details from the raw data, processors and hardware accelerators that use approximate computing can make trade-offs in accuracy to improve performance, energy, power and cost by applying simplified circuits or reducing computation. Therefore, approximate computing also creates a demand for different dataset *resolutions*, meaning details about the raw data (e.g., data precision levels, summarized results, intermediate results, and sampled contexts) from the raw-data storage system that are essential to support exact computing.

Because existing approximate-computing research only focuses on accelerating compute kernels by improving the design of architectural components, programming frameworks, or algorithms, modern computer systems that host approximate computing still use storage system stacks that are designed for conventional exact computing. Using the latest generation of GPGPUs to execute approximate compute kernels, the overhead of preparing input datasets (due to receiving data from the storage device, adjusting data resolutions, etc.) becomes the most critical stage in the data-processing pipeline. Recent advances in approximate hardware accelerators, as embodied in TPUs [29], NGPUs [92], NPUs [18], and mixed-precision support in GPGPUs [60], have further shrunk the execution time in compute kernels and deepened the gap between data preparation and computation in approximate applications.

To fundamentally address the aforementioned bottleneck in approximate computing, the storage device needs to work with the running application to deliver datasets in the required resolution. Since lowering resolution reduces dataset size, such a cross-layer design can lessen the total bandwidth demand from the data source, thus decreasing the most latency-critical data-transfer overhead. Compute kernels can directly use these low-resolution inputs to avoid unnecessary data conversion. In spite of the clear benefits of a storage device that can effectively implement data-resolution reduction, building such a storage device is challenging, as the design must consider all of the following:

Performance The computations required to adjust data resolutions in the storage device need to be efficient enough to not exceed the latency of transferring the adjusted data and should not affect normal I/O workloads.

Quality Reducing data resolutions lowers the latency in data transfer but also has the potential to degrade output quality [25, 37, 41, 42]. If the input data leads to significantly inaccurate results, the application must recompute and/or iteratively retrieve the reduced data, both of which increase end-to-end latency.

Flexibility The design should preserve the ability to provide datasets in the diverse resolutions that applications require. Any design that fails to do this will limit the usefulness of the system.

Cost Costs must be minimized. Datacenter architectures are prohibitively expensive, and hardware components that require large capital outlays will likely prevent a new design from being widely adopted.

As a solution, we propose *Varifocal Storage* (*VS*), a dynamic, multi-resolution storage-system architecture that improves performance while addressing the aforementioned challenges. VS extends storage-interface semantics by introducing a set of operators that applications can apply to make data-resolution adjustments. VS uses computing resources already present in modern storage devices with non-volatile memory (NVM) to support operators that work on the stored raw data—without using additional hardware components. Since the VS architecture only needs to store the raw data, VS adds no storage-space overhead to the storage device.

To provide quality control when applying approximate computing in applications, VS offers the *Autofocus* mechanism to automatically specify resolution: Autofocus selects the lowest resolution that satisfies all control variables for the VS operator and the data inside storage devices before compute kernels on the host computer or other heterogeneous computing resources start processing the data. With Autofocus serving as a kind of approximate-computing vanguard, VS can (1) prevent compute kernels from processing data that will produce low-quality results, (2) reduce performance loss due to recomputation and input data being re-sent in higher resolutions, and (3) allow an application to tolerate a wider range of datasets. To further reduce programmer burden in designing applications and enable the potential of resolution adjustments, VS introduces the *iFilter* mechanism to specify both the approximate operator and the appropriate resolution.

In the later sections of this paper, we evaluate VS by designing and implementing a VS-compliant solid-state drive (SSD) that is an extension of an existing datacenter-class SSD. The current VS-compliant SSD allows applications to adjust data resolutions using operators for value approximations, packing, data filtering, and content selection in firmware programs without modifying hardware design.

In summary, this paper makes the following contributions: (1) It presents VS, a system architecture that optimizes the performance of approximate computing in full-stack design by dynamically changing data resolutions in storage devices to address the demands of performance, flexibility, cost, and quality; (2) it demonstrates the potential benefits of adding another layer of quality control to reduce programmer burden by introducing the Autofocus and iFilter mechanisms that automatically determine data resolutions or even operators; and (3) it describes an implementation of VS to demonstrate the feasibility of VS architecture in modern storage devices and to evaluate the performance of VS using a wide range of approximate-computing applications.

By running a wide range of applications, we show that the manually controlled VS can speed up the most critical data preparation by 2.02× without significantly affecting accuracy. With the Autofocus

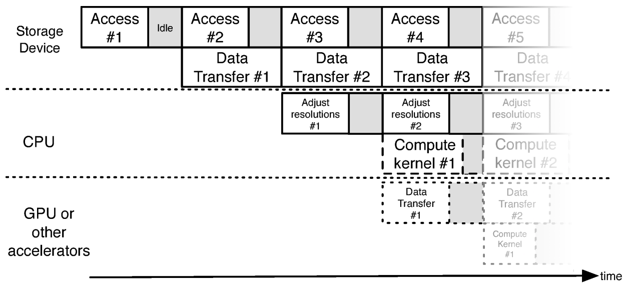

Figure 1: The data-processing pipeline of approximate applications using the conventional execution model.

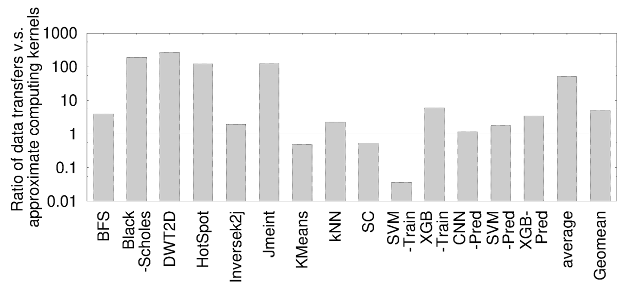

Figure 2: The data-exchange overhead compared against the execution time of performing compute kernels on the same amount of data.

mechanism determining the resolution, VS speeds up performance by 1.70× on average. Using the fully automatic iFilter mechanism, VS can achieve a speedup of 1.74×. Comparing the end-to-end latency of VS with that of conventional approximate-computing architecture, VS is 1.52× faster with programmer optimization, 1.43× faster using Autofocus, and 1.46× faster using iFilter.

2 MOTIVATION AND BACKGROUND

VS not only improves data-supply performance, but also accommodates the demands of a wide range of both approximate- and exact-computing applications. This section presents the motivation for our design, describes missed opportunities in modern computer architecture, and discusses alternative solutions.

2.1 The Overhead of Presenting Datasets in Different Resolutions

Figure 1 illustrates the data-processing pipeline of approximate-computing applications in modern heterogeneous computers. The computer first needs to issue I/O commands for the storage device to access raw data from its internal data arrays and then transfer the raw data through the underlying system interconnect while simultaneously serving other data-access requests. Once the host computer receives a chunk of data, the CPU can start producing datasets in lower resolutions. The approximate-compute kernel can then perform computations using the resolution-adjusted datasets. If the kernel can leverage a GPU, TPU, NPU, or other hardware accelerator, the system must additionally exchange among different components through the interconnects before the accelerator can compute on the prepared data.

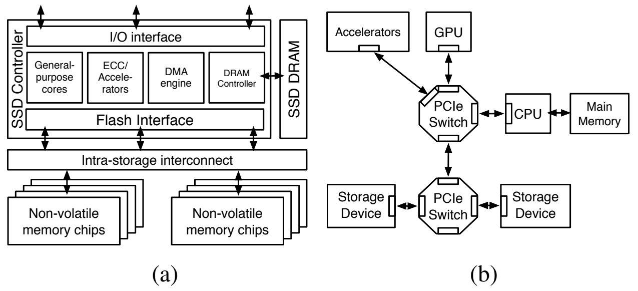

Figure 3: (a) The architecture of modern SSDs. (b) The modern PCIe system-interconnect architecture.

With these approximate-computing-based acceleration techniques, the latency of retrieving data from the storage system becomes the most critical stage in the data-processing pipeline. Figure 2 compares the latency of receiving raw data chunks from a high-end NVM-Express (NVMe) storage device against the execution time of performing approximate/mixed-precision compute kernels on the same data chunks using an NVIDIA Tesla T4 GPU [60] for a set of applications [10, 69, 87, 91] (detailed description in Section 7). Using a highly optimized I/O library that saturates the NVMe I/O bandwidth, the overhead of receiving datasets exceeds the kernel execution as the most critical stage in a majority of these applications.

2.2 Missed Opportunities in Modern NVM-Based Storage Systems

Without revisiting the hardware/software interface for storage devices, conventional approximate-computing frameworks fail to optimize the increasingly critical data-preparation process from the following opportunities:

Reduced data size Since approximate computing works on lower-resolution datasets, the compute kernels usually consume fewer bytes than exact computing ones. However, conventional storage interfaces, including those based on the latest NVMe standard [4], only support read/write commands that exchange raw data between source and destination; applications can never reduce the bandwidth demand of exchanging raw data between the storage device and the host.

Rich device-internal bandwidth Conventional storage interfaces waste the rich internal bandwidth of storage devices. The controllers found in modern datacenter SSDs, including the controller in the prototype SSD that we used for this paper (Section 7.1), support up to 32 channels. The internal bandwidth of our prototype SSD can reach up to 8 GB/s if the SSD uses MLC flash memory chips with an average reading latency at 35 μs for each 8 KB page [22, 55, 66]. However, the application only works on the host computer and exchanges data with the SSD using limited PCIe bandwidth. With newer, faster NVM technologies (e.g., ZNAND [74] or 3DXPoint [27]), the mismatch between the internal and external bandwidths can become more significant.

In-storage processing power Conventional interfaces also hide the freely available processing power in SSD controllers. Figure 3(a) shows the architecture of a modern datacenter SSD. In addition to NVM chips, an SSD contains general-purpose cores and DRAM to execute firmware programs and to cache/buffer data. In spite of the limitations and dynamics of the outgoing bandwidth, the SSD controller can still access its own data-storage arrays with channels and banks. Nonetheless, the SSD's general-purpose cores remain unavailable to applications because conventional interfaces only support access to raw data.

Due to the relatively longer latency of accessing NVM devices and the over-provisioning of processing power to avoid the cost of adding an embedded operating system, SSD cores are idle for significant amounts of time. To accurately determine processor idle time, we analyzed the loading of each processor core in our baseline data-center SSDs under different scenarios. The maximum utilization appeared when we saturated the outgoing PCIe bandwidth by continually issuing 32 MB read requests. Under this scenario, the busiest SSD processor core spent 70.4% of its time parsing NVMe requests, and the second busiest core spent 46.5% of its time receiving commands from the PCIe interconnect. All other processors responsible for managing data accesses for flash data were only busy 12.5% of the time. When the SSD is performing garbage collection, none of the processors are busy for more than 20% of the time due to the long latency of erase and write operations characteristic of SSDs. Consistent with these results, previous studies of data-center-class SSDs and common SSD prototypes [66, 95] have shown the average utilization of their SSD processors to be lower than 30%. With frameworks such as FlashAbacus [96], the SSD controller typically has even more idle time to spare for non-essential workloads.

2.3 Alternative Approaches

A number of alternatives have been suggested to address the data-I/O bottleneck for general-purpose applications and the dataset-preparation requirements for approximate-computing applications. However, none of the alternatives addresses the demands of approximate computing in modern heterogeneous computers. Rather, each alternative only addresses a subset of the challenges of presenting datasets in different resolutions.

Increasing I/O bandwidth The most direct approach to improving data-transfer performance between the storage device and the host computer is to increase the I/O bandwidth of the storage device. However, this approach is difficult and expensive in modern architectures. Figure 3(b) shows the topology of attaching peripheral devices, host processors, and other accelerators in the most popular system interconnect for a PCI Express (PCIe). Most modern SSDs attach to a PCIe using 4$\times$ PCIe Gen3 lanes that provide up to 4 GB/sec of bandwidth. As modern CPUs incorporate their memory controllers on-chip and use an exclusive processor-memory bus, the bandwidth that the host application can use to communicate with other devices (including GPUs, NICs, hardware accelerators, and SSDs) is limited by the total PCIe bandwidth to which the CPU connects. As a result, the actual outgoing bandwidth that the SSD can use is narrower than the theoretical bandwidth, as it is usually the case that multiple devices are competing for the bandwidth going into the CPU/memory controller. In this modern system-interconnect architecture, increasing the bandwidth is very challenging since it requires the CPU to make more PCIe lanes available (i.e., increase

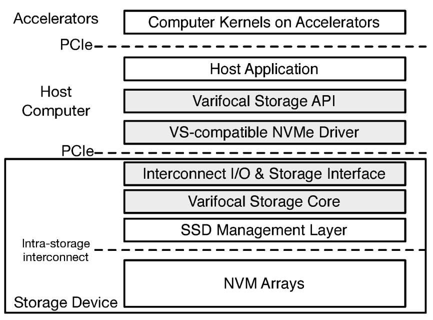

Figure 4: The VS system architecture.

the pin count of the processor) or reduce the number of peripheral devices that the system can connect.

Data compression Although lossy and lossless data-compression algorithms [8, 11, 64, 90] both help to reduce data size and save I/O bandwidth, the overhead of decompressing data on the destination computing device can easily cancel the benefit of reducing data-transfer time; without appropriate hardware support, data compression may lead to performance degradation [45]. This is precisely what we observed (see Section 8.2.6). If the storage device stores data using lossy algorithms, the system sacrifices support for exact-computing.

In-storage processing (ISP) General-purpose intelligent storage frameworks such as Willow [76], Samsung's SmartSSD [16], Morpheus [85], Biscuit [23], Summarizer [40] and FlashAbacus [96] allow applications to use the processing power inside SSDs. These platforms fall short of approximate computing for the following reasons. (1) These platforms aim at offloading computation from exact computing and can lead to suboptimal performance for approximate computing. For example, Summarizer's filter operation picks pages that render datasets distorted from the raw dataset, which increases data exchanges and computation. (2) These platforms require the programmer to customize near-storage computation (e.g., resolution adjustments in approximate computing) on per application basis, increasing the burden of programmers and creating security concerns. (3) Even language support makes programming easier, the programmer can easily overestimate the capability of controllers and hurt performance.

Approximate storage Approximate storage systems store data using unreliable memory cells. Since these cells do not faithfully store raw data , systems that use them can neither support exact computation nor dynamically generate data in different resolutions; such approximate-storage systems simply sacrifice flexibility [21, 28, 43, 73].

Quality control Without revisiting storage-interface design, existing quality-control mechanisms must request full-size, raw data from the storage device [25, 37, 41, 49, 68, 72, 81]. Most frameworks control output quality by comparing subsets of results for exact and approximate computation, missing the opportunity to capture low-quality input that failed the requirement before computation begins. Section 8.2.5 presents the superiority of VS over conventional approaches in this respect.

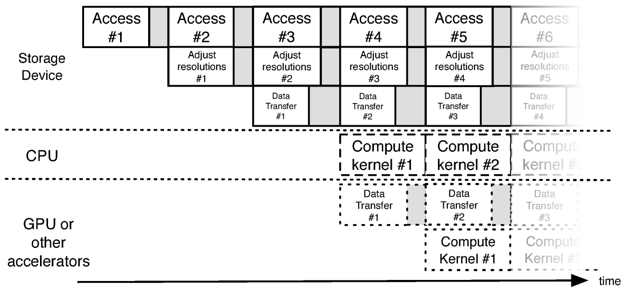

Figure 5: The data-processing pipeline of VS.

3 OVERVIEW

Figure 4 shows VS in a heterogeneous computer system. VS revisits the storage-system stack to allow the device to dynamically produce data with different resolutions on demand. The VS core layer resides inside the storage device to change data resolutions presented to applications. The VS layer interacts with existing system I/O interfaces and provides an extended interface for resolution adjustments. The VS layer also works together with the SSD management layer (i.e., the flash translation layer in flash-based, solid-state drives) to locate the requested data. The host system needs an extended kernel driver and API functions in order for the applications to send requests, exchange data, and receive feedback from the VS core layer. The host application interacts with the API and sends commands specifying the raw data types and operators that VS should work on.

The VS core layer supports a set of operators that are especially effective for applications that contain high data-level parallelism but are able to tolerate inaccuracies in datasets. The VS layer is also where the Autofocus and iFilter perform mechanisms that automatically determine the most appropriate data resolution for quality control. The host application can optionally enable Autofocus and iFilter through VS's API and kernel driver.

Figure 5 illustrates the data-processing pipeline that VS enables to tackle the challenges of performance, quality, flexibility, and cost. By using operators and quality-control mechanisms inside the storage device, VS exploits the richer internal bandwidth and idle processing power to efficiently adjust/prepare datasets in lower resolutions for approximate computing applications. Instead of always sending raw data, VS allows the storage device to send adjusted datasets to the host, reducing the total latency of transferring data over the system interconnect. In this way, VS mitigates the idle time in compute units and frees up CPU resources to tackle more useful workloads, leading to performance gains for approximate applications on the host side.

4 THE VARIFOCAL STORAGE PROGRAMMING MODEL

To prepare an application to take advantage of the VS model, the programmer uses the VS library to specify data resolutions and retrieve adjusted data for the application. The application also needs access to compute kernels that work with lower-resolution data. The details of the VS programming model are given below.

VS provides a set of library functions for applications. The programmer uses these functions to set up (1) the operators required to read data and whether Autofocus or iFilter is enabled, and (2) the

Synopsis	Description
`int vs_setup(int fildes, struct vs_operator** op_list, const char *restrict format)`	This function sets up the VS operator to apply on a file stream that is associated with a file descriptor, `fildes`. The `op_list` describes the desired operators for data associated with the open file descriptor. This function collects data formats within the file through the format string and VS will apply each operator to each type of data in the string accordingly. If the list contains a nop operator, VS will not apply any approximation of the corresponding data.
`int vs_read(int fildes, void *buf, size_t nbyte, struct vs_feedback *fb)`	The function reads data from the storage device using the previously set operators for the open file descriptor and provide the feedback through the struct vs_feedback data structure.
`int vs_release(int fildes)`	The function disables the VS operators on the given data stream that `fildes` represents and releases the resources that these operators use.

Table 1: Sample functions from the VS API.

```
int setup(int argc, char **argv)  {
    // Skip – the rest of code ...
    int infile;
    // VS: Declare VS variables
    struct vs_operator op[1];
    struct vs_feedback fb[1];
    // Open a file descriptor
    infile = open(filename, O_RDONLY, "0600");
    // Read precise data from the file descriptor
    read(infile, &npoints,   sizeof(int));
    read(infile, &nfeatures, sizeof(int));
    // Skip – some other initialization code ...
    // VS: set parameters for desired operator
    // PACKING(default)/PACKING_AF(autofocus)/VS_IF(iFilter)
    op[0].op = PACKING;
    op[0].resolution = HALF;
    // Skip – some other initialization code ...
    // VS: apply the desired VS operator for the file
    vs_setup(infile, &op, "%f");
    // VS: read data processed by the VS operator
    vs_read(infile, buf, npoints*nfeatures*sizeof(float), &fb);
    // VS: disable the usage of VS operator for the file
    vs_release(infile);
    // Skip – the rest of code ...
    // VS: use approximate kernel if the operator succeed
    if(fb[0].resolution == op[0].resolution)
        cluster_approximate(…);
    else
        cluster(…);
    // Skip – the rest of code ...
}
```

Figure 6: A KMeans code sample with inserted VS function calls.

parameters that allows the underlying storage device to adjust data as well as control variables that Autofocus and iFilter use to control the adjusted data. Table 1 lists three representative API functions; the functions are used when an application calls open to create a file descriptor. If the offset of an open file descriptor needs to be manipulated, the application simply uses conventional file system functions like `lseek` or `fseek`.

Figure 6 shows KMeans code (Rodinia benchmark suite [69]) with VS function calls inserted. In the example, KMeans uses conventional system-library functions (e.g., open and `close`) to manage the file descriptor. If the program reads data using standard I/O functions (as in the two `read` function calls in the code), VS does not change the resolution of the accessed data. The modified KMeans code initiates VS for the `infile` file descriptor by calling `vs_setup`. This version of the code sets the desired operator and resolution. The `vs_setup` function also accepts an argument that describes the data formats. In the KMeans code sample, VS will interpret the file content as floating point numbers.

VS starts adjusting data only if the application calls the `vs_read` function. This function resembles the existing Linux `read` function except that (1) the resulting data size may be different from the requested data size, since operators will trim data sizes in most cases, and (2) the function will provide feedback regarding the resolution that VS selects. If the program calls a regular `read` function to replace the `vs_read` in Figure 6, VS will not change the data resolution (even if the program previously initiated VS using `vs_setup`). These API functions (e.g., `vs_read`) can interact with the underlying file system cache to further improve performance if another application is requesting the same dataset with the same resolution.

If VS successfully adjusts the data, the application can use a compute kernel that supports lower-resolution input (e.g., `cluster_approximate`) to further reduce the total execution time of the program. If the kernel is elastic to changes in dataset size (like machine learning algorithms), then no need to change the compute kernels. In many cases, the programmer can compose approximate versions of compute kernels by slightly modifying the original kernel functions to operate on less precise data types or summarized input datasets [70, 71]. The application can also use library functions (e.g., Mixed-Precision CUDA libraries and FANN library for NPU [18]) leveraging approximate hardware accelerators to perform the approximation in compute kernels.

Depending on the approximate compute kernels that the application uses, the programmer can choose different VS operators for data adjustments when calling the `vs_setup` function. To determine the desired resolution, the programmer can leverage existing language frameworks and profiling tools [7, 70–72]. In addition to traditional approaches for determining resolutions, VS provides the Autofocus mechanism to automatically decide the resolution using a set of control variables that the programmer can optionally pass as parameters. The resolution-reduction choices Autofocus makes are usually more conservative than those of a programmer, but Autofocus can nonetheless help applications adapt to datasets. To ensure the quality of the execution result, VS may leverage existing approximate frameworks [37, 41, 49, 68, 72, 81].

If a given application can apply multiple versions of approximate kernels for different VS operators, the programmer can use the iFilter mechanism to let the storage device choose the most appropriate operators and resolutions for each dataset. The programmer can pass "VS_IF" as the operator to trigger the iFilter mechanism and optionally describe the available set of operators and the control variables. Using the feedback data structure (`vs_feedback`), the application can then execute the corresponding approximate kernel.

5 THE CORE VARIFOCAL STORAGE LAYER

The core of VS provides a set of operators to adjust data resolutions. VS exposes these operators to applications through an extended storage interface. The VS layer also implements two mechanisms to determine appropriate data resolutions and provide quality control over the adjusted data.

5.1 VS Operators

VS provides several operators to adjust data resolutions before shipping the data to host applications. To achieve the best performance using the VS model, operators are selected in accordance with the following criteria: (1) The computation overhead must match the processing power inside the storage device. Thus, VS can minimize the impact on access latency and power consumption and avoid extra hardware costs. (2) A wide range of applications must be able to apply the operator, thereby allowing for more efficient use of valuable device resources (VS identifies the most useful operators from previous efforts [70, 71]). (3) The operator must allow VS to take advantage of mismatches between external and internal bandwidths and downsize the outgoing data—the VS model is most effective when the data adjustment can reduce the demand of interconnecting bandwidth.

The current VS framework supports the following categories of operators for diverse data types.

Data Packing The data-packing operator trims the dataset size by using fewer bytes to express each item and by condensing the layout in memory. A data-packing operator is suitable for applications that only use a small range within the number space of the original data type and for applications that can tolerate some inaccuracies in the input data. Since the data-packing operator translates raw data into a less-precise data type, it can potentially decrease accuracy (e.g., double→float→half or int64→int32→short→char).

Quantization The quantization operator rescales the raw values into a smaller value space as well as preserves the relative order of values. The quantization operator is applicable to the application requires large value sapce.

Reduction/Tiling The reduction operator applies a function (e.g., average) to a group of input values and yields a single output value. After applying a reduction operator, VS sends only the resulting value of each group in order to reduce the amount of data passing through the system interconnect. This operator is especially useful for machine learning and statistics applications when the input data is uniformly distributed [70].

Sampling The sampling operator chooses a subset of items from the raw data and sends the selected items to the host computer. Operators in this category can perform uniform/random data selection or report only the most representative data. The sampling operator helps to filter out repetitive/similar inputs that make no contribution to the final application result. If the compute kernel is elastic with respect to the number of records within the dataset, the sampling operator can achieve the same effect as that of loop perforation [57, 61, 78] but without any code modification (without the VS sampling operator, conventional loop perforation needs the raw data to be present in system memory).

Besides, by providing the preceding types of operators, VS gives system designers the chance to extend the number of operator types using the mechanisms described in Section 6.3.

5.2 Autofocus and iFilter

The Autofocus and iFilter mechanisms provide quality control and reduce the amount of programmer effort required to adjust data resolutions. Autofocus and iFilter are inspired by two previously observed phenomena: (1) The quality of the input data affects the quality of the result in approximate computing [37, 41, 42]. (2) A small subset of input data is representative of the rest of the input data in approximate-computing applications that tolerate inaccuracies [41] . Building upon these observations, Autofocus and iFilter can select the resolution/operator using only a small portion of the raw input data from a requested dataset and then monitor the quality of the adjusted input data.

Algorithm 1 Autofocus

```
    Input: $op$, $CVs$                          ▷ $CVs$ are optional
 1: for each $r \in R$ do                      ▷ $r$ is sorted in ascending order
 2:     $D \leftarrow RawData$
 3:     for each $d \in D$ do
 4:         $d' \leftarrow adjust\_data(d, op, r)$
 5:         $\Delta \leftarrow compute\_CVs(d, d', op)$
 6:         if $\Delta$ satisfy $CVs$ then
 7:             remove $d$ from $D$
 8:             if $D \in \emptyset$ then
 9:                 return r
10:         else
11:             go to 1
```

5.2.1 *Autofocus.*

Autofocus allows the programmer to simply specify the desired VS-operator, letting VS decide the most appropriate resolution that guarantees quality while improving performance. Autofocus also makes applications more adaptive to different datasets, as the most appropriate resolution varies from dataset to dataset.

Algorithm 1 shows how the Autofocus mechanism works. Autofocus makes decisions using the programmer-selected operator (op) and the quality-control variables specified in (CVs), with values being determined by either the programmer or the default settings. Autofocus then adjusts each data subset (d) using the specified operator (op) with the least precise resolution (r) that Autofocus has not examined from the available operator resolutions (R).

Autofocus will check the quality of adjusted data (d') by comparing the adjusted data with the raw data (Line 5) and generate the comparison result (Δ). Take the data-packing operator as an example; Autofocus will compare the precision loss between the original data type (e.g., FP32) and the adjusted data type (e.g., FP16) and check to see whether the difference is smaller than the value from the control variable. To reduce overhead of operators that need more complex logic (e.g., sampling) to generate Δ or when the controller's load is high, Autofocus only applies the quality-control function $compute\_CVs$ to each byte of data in the first few pages (8 in our experiments) and then randomly checks the remaining adjusted data. Table 2 summarizes how we compute the control variables for each VS operator.

If every checked piece of the adjusted data successfully passes through the $compute\_CVs$, VS will report the current resolution to the host application and transfer the adjusted data (Line 9 of Algorithm 1) through the system interconnect. If the quality of the adjusted data (d') fails on the control variables, Autofocus will fall back to the next resolution (Line 11 of Algorithm 1).

5.2.2 *iFilter.*

iFilter can work without programmer input and is more effective than Autofocus for applications having compute

VS Operator	Function $compute\_CVs$	Description
`Data Packing`	$abs(data_{raw}, data_{adjusted})$ and $min_{new\_data\_format} \leq data_{new} \leq max_{new\_data\_format}$	For data packing, VS calculates and check if (1) the absolute difference between the original data and adjusted data is smaller than the given threshold and (2) adjusted data falls in the range of the target data type.
`Quantization`	$abs(data_{raw}, data_{adjusted} * scale\_factor)$, where $scale\_factor = \frac{max(data_{old\_data\_format}) - min(data_{old\_data\_format})}{max(data_{new\_data\_format}) - min(data_{new\_data\_format})}$	For quantization, VS controls the quality by rescaling the adjusted data back to the raw data format and measuring the absolute difference. VS drops the adjustment if the difference is greater than the given threshold.
`Reduction/Tiling`	$abs(data_{raw}, data_{adjusted})$	For reduction/tiling, VS computes the absolute difference between raw data and adjusted data. VS compares if the absolute difference is smaller than the given threshold.
`Sampling`	$binary\_distance(data_{raw}, data_{adjusted})$ [65]	For sampling, VS calculates the Hamming distance between raw data and adjusted data and drops the current decision if the distance is larger than the given distance.

Table 2: Summary of function $compute\_CVs$**.**

Algorithm 2 iFilter

```
Input: $OP$, $CVs$                                    ▷ $OP$, $CVs$ are optional
 1: for each $op \in OP$ do
 2:     for each $r \in R[op]$ do                     ▷ $r$ is sorted in ascending order
 3:         $D \leftarrow FirstFewChunksOfRawData$
 4:         $min\_size[op] \leftarrow 0$
 5:         $min\_res[op] \leftarrow r$
 6:         for each $d \in D$ do
 7:             $d' \leftarrow adjust\_data(op, d, r)$
 8:             $\Delta \leftarrow compute\_CVs(d, d', op)$
 9:             if $\Delta$ satisfy $CVs[op]$ then
10:                 remove $d$ from $D$
11:                 $min\_size[op] \leftarrow min\_size[op] + size(d')$
12:                 if $D \in \emptyset$ then
13:                     go to 1
14:             else
15:                 go to 2
16: $op \leftarrow select\_op(OP, size, res)$
17: $D \leftarrow RawData$
18: for each $d \in D$ do
19:     $d' \leftarrow adjust\_data(op, d, res[op])$
20:     $\Delta \leftarrow compute\_CVs(d, d', op)$
21:     if $\Delta$ satisfy $CVs[op]$ then
22:         remove $d$ from $D$
23:         if $D \in \emptyset$ then
24:             return $op$, $r$
25:     else
26:         remove $r$ from $R\_op$
27:         go to 1
```

kernels that are compatible with multiple VS-operators. Algorithm 2 shows how the iFilter mechanism works.

The iFilter algorithm includes a *decision-making phase* (Line 1–Line 15) and a *monitoring phase* (Line 16–Line 27). In the decision-making phase, iFilter will try out all available VS operators (OP) that can be applied to the input data type for the first few pages (8 in our experiments) of the requested data. The iFilter algorithm is similar to the Autofocus algorithm in that it selects the most appropriate resolution for each operator, except that iFilter will keep track of the resolution ($min\_res[op]$) and the resulting data size ($min\_size[op]$) for each operator (Line 5 & Line 11).

After the decision-making phase, iFilter will enter the monitoring phase and select the operator that yields the smallest data size (Line 16). iFilter uses the selected operator (op) to adjust every piece of raw data. If iFilter successfully reaches the end of the request, iFilter will report the selected operator and resolution (Line 24) and send the adjusted data to the host. If iFilter fails to reach the end of the request, it will remove the current resolution from the available set of resolutions ($R[op]$) and restart the decision-making phase to choose the next appropriate operator and resolution (Line 26 & Line 27). The computation overhead for iFilter is thus higher than that of Autofocus since iFilter examines more operators to choose the one with the minimum amount of data going through the system interconnect. However, the additional overhead is negligible with large datasets because the relevant VS operators need only be applied to the first few chunks of the dataset.

6 BUILDING A STORAGE DEVICE COMPLIANT WITH VARIFOCAL STORAGE

Building a VS-compliant storage device means tackling challenges associated with (1) providing a hardware/software interface that allows applications to describe the resolutions and quality of the target data, and (2) minimizing the computational overhead/cost of adjusting data resolutions. VS overcomes the former challenge by extending the NVMe interface; this requires the fewest modifications to the system stack and applications. VS addresses the latter challenge by exploiting the idle cycles available in modern SSD controllers. This section describes the NVMe extensions and the use of existing architectural components in an SSD that are needed to ensure VS compliance. This section also describes how to add new operators to the VS architecture.

6.1 NVMe Extensions for VS

Conventional storage interfaces such as the popular NVMe protocol only support read/write commands for data access. Therefore, the NVMe extensions for VS need to provide commands to set up VS operators and apply those operators on datasets. The extended NVMe interface aligns with the programming model in Section 4 to simplify the complexity of software implementation.

Setting up VS operators The NVMe extension for VS provides a new command to set up I/O stream and file descriptors—the `vs_setup` command. This command carries the descriptor number using the 8-byte reserved area in the standard NVMe command format. The descriptor usually corresponds to a file or I/O stream in high-level programming language/system abstractions.

VS uses an abstraction similar to an instruction-set architecture that allows the API to map the demanding operators to each stream. Each operator starts with a 4-byte opcode followed by a 4-byte integer for the number of arguments, which is then followed by the arguments (e.g., target data resolutions, quality control variables). For each category of operator, VS provides a different opcode for

Host System	
CPU	Intel Core i7-7700K [26] @ 4.2 GHz
GPU	NVIDIA Tesla T4 [60]
OS & file system	Linux Kernel 4.15 & EXT4
baseline/VS-compliant SSD	
Controller	Microsemi flashtec controller with 32 channels [66]
DRAM	2GB DDR4 DRAM
Capacity	768 GB with 10% overprovisioning
Flash Chip	MLC NAND/8 KB page size [55]
I/O interface	NVMe through PCIe 3.0×4

Table 3: The platform configuration used for evaluation.

different data types. The API generates a sequence of operators and works with the driver to store the sequence in a host DMA page for the SSD to access.

Upon receiving the `vs_setup` command, the SSD will add the page specifying the operators into its internal data structure, which usually resides in the DRAM space of the SSD. Later commands can use the descriptor number to indicate the operators that a `vs_setup` command previously set and look up the corresponding operators from the internal data structure. When the application does not need the setup operators for the I/O stream, the `vs_release` command will signal the SSD to release the descriptor, allowing a later `vs_setup` command to reuse the descriptor number.

Applying VS operators VS only adjusts data resolutions on data requested by the `vs_read` command. The `vs_read` command is similar to a typical `read` command with the following exceptions: (1) The `vs_read` command contains a flow number in its 8-byte reserved area. (2) The `vs_read` command reports the resulting data size to the host, as most operators will change the data size or a negative value if an error occurs. (3) The `vs_read` command reports the selected operator and the degree of data adjustment to the host software stack if necessary.

Since the regular `read` does not provide any feedback to the host computer other than the error code, `vs_read` requires the driver to always allocate an additional DMA page on the host for each command that receives the feedback. As NVMe's Physical Region Page (PRP) list uses a type of linked-list data structure that allows the `vs_read` command to specify an almost unlimited number of DMA pages, accommodating feedback information does not require any change in the NVMe command format. Rather, only minor modifications to the device driver are required.

The current NVMe standard only allows each NVMe command to transfer at most 32 MB of data. Consequently, firmware programs will keep the offset of processed data within the data stream associated with a given descriptor. If Autofocus or iFilter revises a decision while processing a large (e.g., greater than 32 MB) file transaction, the API is allowed to generate commands to restart the entire transaction with the revised decision.

6.2 Architecting a VS-compliant SSD

To minimize extra hardware costs, VS makes efficient use of existing architectural components in modern SSDs.

With modern flash memory technologies, the critical path of the data-access pipeline is determined by either the access time of flash chips or the latency of the DMA stage (i.e., depending on the outgoing bandwidth) of the SSD. In either case, data transfer through the critical path in the pipeline usually takes a few microseconds. As even the humblest modern processor cores can execute thousands of instructions within the latency period of the critical stage in the SSD data-access pipeline, such cores are idle most of the time and leave slack that can be taken up by VS to apply operators without the need for additional accelerators. An SSD will not experience any performance degradation in accessing its own data array if the applied operator does not create more than the average data-access latency in the pipeline.

VS extends firmware programs to reclaim these idle computing resources for VS operators. When a chunk of the requested data (e.g., a flash page) arrives in the SSD DRAM, the extended firmware programs will signal an underutilized or idle processor core to fetch data from the data location in the SSD DRAM and apply the desired operator(s). Since VS operators reduce dataset size, the programs using VS operators can reuse the existing data buffers and thus do not require additional space to buffer their processing results; the firmware programs can keep their runtime states in the SSD DRAM or in the data caches of the processor cores.

6.3 Adding New Operators

In our SSD, VS operators are implemented as overlay functions in the firmware programs. With the extended NVMe protocol providing a mechanism to exchange information for adjusting data resolutions, the overlay functions receive the same set of arguments (including the resolution and the pointer to the SSD DRAM data-buffer location) and report the data size and resolutions through a data structure defined in our framework. To add a new operator, our current tool chain requires the designer to first write C functions. The designer also needs to update a header file where the firmware program identifies and locates the new operator. The designer can then use a cross-compiler to generate machine code for the controller's microarchitecture. Finally, the system deploys the compiled firmware program to the SSD through the standard firmware update command in the NVMe protocol [4].

7 EXPERIMENTAL METHODOLOGY

We developed VS by extending a datacenter-class SSD. We then measured the performance of the resulting system with several workloads that span a wide range of applications. This section describes the setup of the experimental platform and the benchmarks that we used.

7.1 Experimental Platform

We built a VS-compliant SSD by extending a commercialized, datacenter-class SSD. We attached the VS-compliant SSD to a high-end heterogeneous machine with a GPU. The host operating system contains the extended NVMe driver to support additional VS NVMe commands. Table 3 lists the key specifications of the host computer and the SSD. The VS-compliant SSD runs our modified firmware programs. The firmware is also compatible with a standard NVMe. Since we did not modify the code that handles regular NVMe commands, the firmware achieves the same performance as a regular NVMe SSD with the same hardware configuration. Throughout our tests, the baseline SSD achieved a 3.2 GB/s bandwidth when communicating with the host systems, but the theoretical internal bandwidth is twice of that.

Workload Name	Application Category	Operator	Resolution	Raw Data Size	Relative Error Rate
Breadth-First Search (BFS) [69]	Graph Traversal	Packing	62.5%	3.5 GB [69]	0%
Black-Scholes [91]	Financial	Packing	50%	3 GB [91]	< -0.25%
HotSpot [69]	Physics Simulation	Reduction	25%	2 GB [69]	< -0.15%
2D Discrete Wavelet Transform (DWT2D) [69]	Image/Video Compression	Reduction	50%	1.6 GB [69]	< 0.1%
Inversek2j [91]	Robotics	Packing	50%	2 GB [91]	< -0.01%
Jmeint [91]	3D gaming	Packing	50%	2 GB [91]	< -0.02%
KMeans [69]	Data Mining	Quantization	25%	1.36 GB [69]	< -0.97%
k-Nearest Neighbors (kNN) [52]	Data Mining	Packing	50%		< -0.01%
streamcluster (SC) [69]	Data Mining	Packing	50%		< -0.01%
ThunderSVM–Train (SVM-Train) [87]	Machine learning	Sampling	75%	2.6 GB [79]	+ 0.5%
ThunderXGB (XGB) [88]	Machine learning	Packing	50%		< -0.10%
CNN–Pred [1]	Machine learning	Quantization	12.5%	0.95 GB [79]	< -0.6%
ThunderSVM–Pred (SVM-Pred) [87]	Machine learning	Packing	50%		< -0.01%
ThunderXGB–Pred (XGB-Pred) [10]	Machine learning	Packing	50%		< -0.10%

Table 4: Workloads, default VS operators, input data sizes, and error rates.

We performed all experiments with 90% utilization of SSD capacity. Because SSDs over-provision internal data arrays (typically by 7%) in order to minimize garbage collection, wear-leveling, and read-intensive workloads like those we created, we did not observe any interference between VS operations and the regular SSD workloads.

7.2 Benchmarks

The workloads we used for VS-performance assessment are shown in Table 4. We used these workloads on both the baseline configuration and the VS-enabled configurations. We selected the given set of applications based on the following criteria: (1) the application had to be representative of approximate computing workloads found in a publicly available repository, and (2) the application had to accept large, publicly available datasets or provide a data generator capable of producing large, arbitrary datasets that could serve as meaningful input. Table 4 lists the dataset sizes that we used in experiments; these are also the largest dataset size that our GPU can accommodate but do not represent a limitation of our SSD or the VS programming model. We followed examples found in previous work in modifying the compute kernels of Black-Scholes [91], Hotspot, DWT2D, and KMeans [70, 71]. We also implemented approximate-computing versions of kNN, SC, SVM, and XGBoost by leveraging the native mixed-precision support in NVIDIA's latest Turing architecture. When running these workloads, we used the default parameters that each workload or its demo script suggested. For each application, we also tried our best to exploit pipeline parallelism that overlaps I/O, resolution adjustment and compute kernels to hide latencies.

Table 4 also lists the lowest data resolutions and the corresponding operators that these approximate-computing applications can accept. For each workload, we carefully profiled and chose the operators and their parameters to limit the relative error rate to less than 1% compared to the exact version of the same application.

In our experiments, three groups of benchmark applications were chosen to use the same datasets: (1) KMeans, kNN, and SC, (2) SVM-Train and XGB-Train, and (3) CNN-Pred, SVM-Pred and XGB-Pred. With respect to evaluating VS, the key difference between KMeans and both kNN and SC is that KMeans uses an aggressive packing operation that reduces input size by 25%. The key difference between SVM-Train and XGBoost is that SVM-Train encourages the programmer to set aside 25% of the raw data for training. For predictors on machine learning (ML) models (e.g., SVM-Pred), we trained the models using precise datasets and reduced the resolutions of the datasets to be predicted. Using these predictors, CNN allows an aggressive quantization that reduces 87.5% of the data size, while other models would lead to errors larger than 1%.

For the basic/programmer-directed VS version, we applied operators and target resolutions as shown in Table 4. When the Autofocus and iFilter mechanisms are enabled, our implementations check the feedback from the VS API. If VS decides to adjust data resolutions, our code can choose to apply appropriate compute kernels to process data. Otherwise, our code uses the baseline compute kernels. When using Autofocus and iFilter, we selected a set of default control variables that were relatively conservative across all applications. For control variables, we used a delta value of 1% for packing, reduction, and quantization as well as 1% binary difference [65] for sampling, since we are targeting at less than 1% error rate.

8 RESULTS

This section presents the performance results for VS on our prototype system and the potential impact of VS on approximate computing.

8.1 The Overhead of VS Operators and Mechanisms

Throughout our experiments, most VS operators required less time than the critical stage of the original data-accessing pipeline of an SSD with limited processor cores, suggesting that the operators can take full advantage of processing inside the storage device. In the most complex case, the *packing* operator takes 1.3 μs to convert a whole page of double-precision numbers into single-precision floating-point numbers, and the *quantization* operator takes 2 μs to rescale a double-precision number into an integer, both of these times are shorter than the critical-stage latency of our SSD. The *reduction* operator takes 0.76 μs to evaluate the average of every pair of double-precision floating-point numbers within a flash page. The *sampling* operator generally takes 0.4 μs to randomly select from binary data.

The Autofocus and iFilter mechanisms also use the SSD general-purpose cores to execute their algorithms. For the Autofocus mechanism, VS takes at most 25 μs to stabilize the resolution for an operator working on binary numbers. For iFilter, the decision-making phase takes about 150 μs to make its first decision because we need

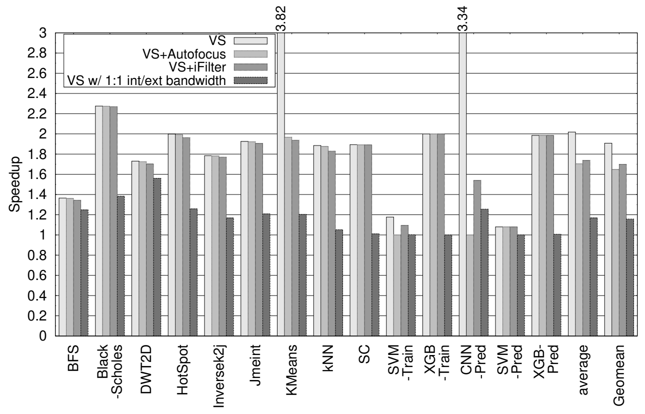

Figure 7: The speedup of reading inputs and adjusting data resolutions using VS.

to perform sampling for all operators. Once Autofocus and iFilter have determined the required resolution, both mechanisms simply compute on values for control variables, so the overhead is negligible and the throughput unaffected.

8.2 The Performance of Data-Resolution Adjustments

Figure 7 shows the speedup in reading input datasets and adjusting data resolutions for each workload using different VS modes; VS is compared with the conventional approximate programming model that relies on the host to adjust data resolutions.

8.2.1 Programmer-Directed VS. Choosing the default VS settings and specifying the desired operator and resolution can speed up the performance of data adjustment by 2.02×. For KMeans and CNN-Pred, which tolerate very low-resolution inputs, the speedup of data adjustment can reach up to 3.82× since the storage device only needs to send out 25% of the raw data size to the host. For Black-Scholes, adjusting raw data on the host is more time-consuming than data transfer. Therefore, VS can achieve more than 2× speedup since VS also takes the advantage from the ISP model for data adjustment. Even with the geometric mean that discounts outliers, VS still exhibits a 1.91× speedup (Figure 7).

8.2.2 Autofocus and iFilter. Without any programmer input on the desired resolution or even on the operator, Autofocus and iFilter accelerate the process of preparing datasets for approximate kernels by about 1.70× and 1.74×, respectively.

For most workloads, the Autofocus mechanism effectively selects the same resolutions as those obtained using exhaustive profiling. For KMeans, the programmer's decision to condense the dataset into 25% of the original space by quantizing, but Autofocus only quantizes the dataset in half of the original space, producing a result indistinguishable from the result achieved using the raw dataset. For CNN-Pred, Autofocus conservatively decides to not quantize inputs; however, if the programmer uses exhaustive profiling, the quantization operator can shrink the input data size by 87.5%.

For SVM-Train, Autofocus does not perform any adjustment, but ships the raw data for kernel computation. As the kernel computes on raw data, SVM-Train skips the data-preprocessing stage on the host, so we still see a slight performance gain in data adjustments.

In the fully automatic mode, iFilter achieves a speedup of 1.74× for data preparation. Though the overhead of iFilter in its decision-making phase is larger than that of Autofocus (as iFilter may need to test more operators/resolutions), this overhead is relatively insignificant as inputs get larger. For most cases, iFilter makes the same decisions of the operator and target resolution as does Autofocus, except that for KMeans, SVM-Train and CNN-Pred, iFilter selects packing instead of the programmer's decision.

In our experiments, the relative error rate of computation observed when using Autofocus and iFilter never exceeded the values in Table 4 because Autofocus and iFilter always made more conservative choices than the programmer.

8.2.3 Internal/external bandwidth. VS is most useful when the SSD has limited external bandwidth. Nonetheless, because VS adjusts data resolutions within the data-access pipeline and avoids the operating system overhead, the VS model is still beneficial when internal bandwidth matches external bandwidth. To quantify this benefit, we modified the SSD firmware to only allow the controller to use half of the SSD channels, so the internal bandwidth matched the external bandwidth while preventing the application from taking advantage of the reduced demand for outgoing bandwidth.

The "VS w/ 1:1 int/ext bandwidth" bar in Figure 7 shows the speedup from using this modified version of our prototype SSD. Without being able to rely on the host CPU for data adjustment, the basic VS still speeds up the total latency of preparing datasets by 1.17×. Additionally, VS reduces the size of data going through the system interconnect, making applications more adaptive when many devices have to compete for the same set of limited PCIe links.

8.2.4 Case study: shared datasets. VS can reduce space overhead by storing only one copy of each dataset but dynamically changing resolutions to accommodate the demands for diverse applications. As noted above, we allowed three groups of applications, KMeans/kNN/SC, SVM-Train/XGB-Train, and CNN-Pred/SVM-Pred/XGB-Pred to share raw input.

In our study, the basic VS allowed the programmer to use the quantization operator and a resolution that reduces data size to 25% for KMeans while using packing operator for kNN and SC to reduce data size to 50% with the shared dataset. When Autofocus and iFilter were enabled to select resolutions by previewing the input dataset without having the compute kernels running, all mechanisms chose a resolution of 50% for these applications. Note that without an architecture like VS, the storage system must store multiple versions of a shared dataset or provide raw data to the host for preprocessing, hurting either space-efficiency or performance.

For SVM-Train/XGB-Train, our experiments also showed that the programmer was able to pick different operators for the shared dataset. When iFilter is enabled, it selects packing for SVM-Train instead of sampling with the same resolution as that chosen by iFilter for XGB-Train. As SVM-Train's compute kernel is elastic to different input dataset sizes, iFilter allows an application to take advantage of SVM-Trains's elasticity to discard some data and achieve an effect similar to the effect of loop perforation in an unmodified compute kernel. Similarly, in CNN-Pred/SVM-Pred/XGB-Pred, the programmer can quantize input data using VS to achieve better performance than the performance achieved by simply using the same operator for the same dataset.

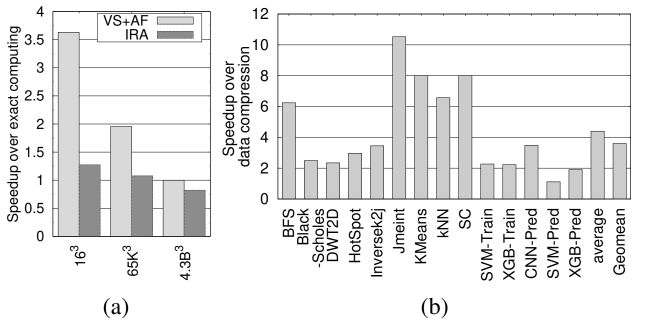

Figure 8: (a) The speedup of end-to-end latency using VS and conventional approximate-computing framework. (b) The speedup of data preparation using VS, compared with data compression.

8.2.5 Case study: diverse datasets. Since Autofocus determines the most appropriate resolution by examining the characteristics of datasets, Autofocus makes VS more adaptive to changes in input datasets for each approximate-computing application. In addition, Autofocus does not rely on feedback from kernel computation results and does not require the storage device to send raw data in the beginning, so Autofocus is more efficient than conventional approaches tackling the same problem. To illustrate this strength of VS, we modified the data generator of Jmeint from AXBench to generate random points in various sizes of 3D spaces. We next present the results when using Autofocus with datasets from three different dimensions: 32^3, 65536^3 ($65K^3$) and 4294967296^3 ($4.3B^3$).

Figure 8(a) shows that VS with Autofocus exhibits significantly shorter end-to-end latency for all datasets compared to the conventional approximate-computing approach using IRA [41]. We used the unmodified exact-computing version of Jmeint as the baseline. Since Autofocus does not need to send raw datasets to the host, VS outperforms IRA by more than 2.86× in the case of the 32^3 dataset, with VS only sending data encoded in 8-byte integers. For the $65K^3$ dataset, Autofocus down-samples the datasets to short data type, leading to a 1.80× speedup over IRA.

In the case of the $4.3B^3$ dataset, the distribution of point coordinates expands the number space to 32-bit floating point, so approximate computing kernels cannot take advantage of using less-precise values without exceeding the 1% error rate limit—both VS and IRA will apply exact computing to generate results. As Autofocus detects no potential in changing data resolutions, the slight slowdown of VS comes from the overhead that Autofocus needs to make a decision. In contrast, IRA slows down by 18% when approximate computing cannot generate meaningful results.

8.2.6 Data Compression Comparisons. Since the most significant VS performance gain comes from reducing data-movement overhead, we also compared VS with several high-performance lossy/lossless compression algorithms: FPC [8], C-Pack [11], BDI [64], and ZSTD [90]. We clocked the time of reading compressed data, of decoding data, and of adjusting resolutions. We excluded the overhead of compressing data. We use the best-performing compression algorithm as our baseline in Figure 8(b), showing the speedup of using VS comparing against data compression.

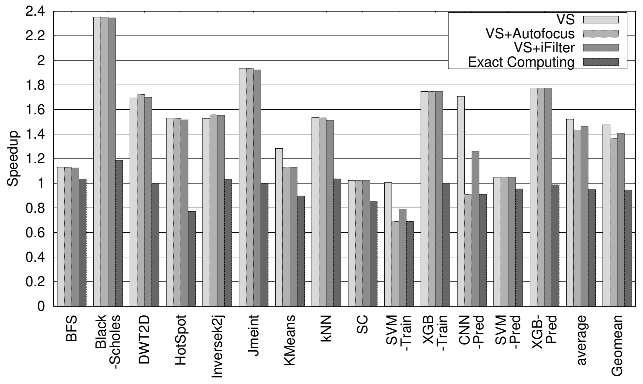

Figure 9: The speedup of the end-to-end latency.

On average, VS outperforms the best compression algorithm for each dataset by 4.40×. This is because the overhead of decompression consumes considerable overhead on the host even though decompression saves bandwidth. In addition, VS generates data that compute kernels can directly process, but the application can never bypass the decompression overhead if we use data compression. Without hardware-accelerated compression/decompression (which adds costs), data compression cannot compete with VS.

8.3 The Impact of VS on Total Application Latency

Figure 9 shows VS's impact on the relative end-to-end latency of running a complete workload using workloads with the conventional approximate computing approach with GPU-accelerated kernels as the baseline. Since VS efficiently prepares input datasets in storage devices for approximate computing kernels running the GPU, the basic programmer-directed VS leads to a speedup of 1.52× for these applications. Using Autofocus to dynamically select data resolutions, these applications achieve an average speedup of 1.43×. As Autofocus adjusts data resolutions under the constraints of the control variables that generally lead to more conservative decisions than the programmer, Autofocus gives up resolution adjustments in SVM-Train and CNN-Pred and applies exact computing kernels so as not to distort the result. Without any programmer intervention, iFilter can improve performance by 1.46× because iFilter has more flexibility in choosing the appropriate combinations of VS operators and resolutions compared to Autofocus. However, without using VS, the conventional approximate-computing approach can only speed up exact computing by 1.07×.

8.4 Power and Energy

To quantify the effect of reducing the CPU workload, total power, and energy consumption, we first examined the CPU frequency when performing data packing on the VS-compliant SSD using the baseline host-version implementation. We sampled the CPU frequency every 500 ms. Even though packing is a very lightweight operation, adding this computational burden to the host program still forces the CPU frequency to go beyond 3 GHz most of the time. For VS, which requires that the CPU handle DMA or issue NVMe commands, the peak CPU frequency during the data I/O is only 1274 MHz. Using a Watts Up meter to measure the power

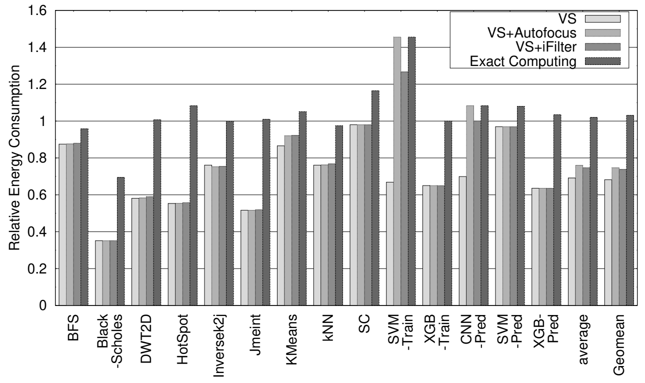

Figure 10: The total system energy consumption.

consumption, the total system consumes 64.7 W for this frequency. Without VS, the system consumes an average of 70.8 W during the whole data I/O process.

Since VS reduces both the power consumption during I/O and the total application latency, VS also reduces the energy consumption. To measure power consumption, we used Watts Up to measure the power draw every 200 ms. Figure 10 shows that the basic VS achieved an average energy savings of 32% for these applications compared to the conventional approximate-computing approach. Even without a programmer's aggressive decision in adjusting data resolutions, VS's Autofocus and iFilter still achieve the same level of energy savings in most applications, except for SVM-Train and CNN-Pred due to their increased end-to-end latency as Section 8.2.2 explains. Autofocus and iFilter provide energy savings of 25% and 27%, respectively. In contrast to this, the conventional architecture with aggressive data adjustments and approximate-computing kernels could only improve energy consumption over exact computing by 5%.

9 OTHER RELATED WORK

Approximate computing has a significant presence among solutions that tackle the limitations of modern hardware design. Using simplified algorithms, smaller ALUs/FPUs, or faster operators, approximate computing maximizes the area-efficiency of silicon chips [24, 29, 32, 34, 46, 54, 77, 83, 86, 93, 97]. By designing simpler, faster approximate circuits (e.g., circuits that use neural-network accelerators [59], load value approximation [56], or approximate memoization [3]), approximate computing also avoids intensive usage of slower but precise circuits for better performance or energy efficiency. In addition, approximate computing allows hardware designers to use unreliable transistors that are commonly found in advanced process technologies [13, 15, 36]. Yet all of the approximate-computing research cited above still follows the single-point design principle, creating the resolution-adjustment problem that this work tries to address. VS is complementary to these projects and can work together with them to address the issues they raise.

To reduce the overhead of applying approximate hardware or software-based approximate-computing solutions, current research projects provide support and analysis through programming language extensions and compilers [5, 7, 13, 15, 36, 37, 41, 70–72]. Since VS simply exposes its features to applications through an API and proposes extensions in the I/O protocol and firmware programs, applications can adapt VS without programming language extensions or compilers. Further, the Autofocus and iFilter mechanisms control input quality after applying VS operators within storage devices, so VS can react before the compute-intensive kernel starts. VS and existing projects are also orthogonal; the system can incorporate VS with existing approximate-computing programming frameworks to use VS operators and mechanisms more efficiently.

Even though VS shares the benefits from recent advances in ISP [6, 9, 14, 23, 31, 33, 35, 40, 67, 75, 76, 82, 85, 89, 96] and near-data processing [2, 17, 20, 39, 48, 50, 63, 80, 84], these frameworks need the mechanisms that VS offers in order to execute approximate computing applications efficiently. And while using approximate computing in channel encoding [38, 62] and memory controller [30] can achieve an effect similar to that of VS in terms of reducing data-movement overhead, VS is independent of these projects and requires no changes in hardware.

10 CONCLUSION

This paper presents VS architecture that supports arbitrary data resolutions for both exact and approximate computing. VS adjusts the resolution of the input data within source-storage devices giving applications a simple way to access the features of VS and programmers a simple interface to do the same. VS significantly reduces overhead and speeds up latency by leveraging underutilized processor resources. This paper also describes the Autofocus and iFilter mechanisms that automatically select the most appropriate parameters for data adjustment that reduces programmer burden while enforcing quality-control measures for outgoing data.

Through experiments conducted with a VS-compliant SSD and the experience gained from tailoring applications on the platform, this paper also demonstrates that a VS-compliant architecture requires very few modifications to hardware or software. A clear indication of VS's efficiency relative to conventional approximate-computing architectures may be found in the 2.02× speedup observed for VS-based data-resolution adjustments and the 1.52× speedup observed for total end-to-end latency, with both improvements producing a change in results of less than 1% . In summary, VS improves performance, maintains flexibility, guarantees quality, and incurs no storage-space overhead for adjusting data resolutions—all at a low cost.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments. We also owe a debt of gratitude to Christopher Fraser for his excellent copyediting skills. We would like to thank the AI infrastructure group and academic relations group from Facebook in providing insights on datacenter workloads and research award funding. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Quadro P5000 GPU used in the early phase for this research. We also appreciate Professor Steven Swanson from University of California, San Diego in supporting the development of our prototype SSD. This work was sponsored by the two National Science Foundation (NSF) awards, 1940046 and 1940048. This work was also supported by new faculty start-up funds from North Carolina State University and University of California, Riverside.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

[2] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. 2017. Compute Caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 481–492. https://doi.org/10.1109/HPCA.2017.21

[3] C. Alvarez, J. Corbal, and M. Valero. 2005. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.* 54, 7 (July 2005), 922–927. https://doi.org/10.1109/TC.2005.119

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

[5] Woongki Baek and Trishul M Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, Vol. 45. ACM, 198–209.

[6] S. Boboila, Youngjae Kim, S.S. Vazhkudai, P. Desnoyers, and G.M. Shipman. 2012. Active Flash: Out-of-core data analytics on flash storage. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. 1–12. https://doi.org/10.1109/MSST.2012.6232366

[7] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. 2015. Probability Type Inference for Flexible Approximate Programming. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 470–487. https://doi.org/10.1145/2814270.2814301

[8] M. Burtscher and P. Ratanaworabhan. 2009. FPC: A High-Speed Compressor for Double-Precision Floating-Point Data. *IEEE Trans. Comput.* 58, 1 (Jan 2009), 18–31. https://doi.org/10.1109/TC.2008.131

[9] 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

[10] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. *CoRR* abs/1603.02754 (2016). arXiv:1603.02754 http://arxiv.org/abs/1603.02754

[11] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas. 2010. C-Pack: A High-Performance Microprocessor Cache Compression Algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18, 8 (Aug 2010), 1196–1208. https://doi.org/10.1109/TVLSI.2009.2020989

[12] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. 2013. Analysis and characterization of inherent application resilience for approximate computing. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–9. https://doi.org/10.1145/2463209.2488873

[13] H. Cho, L. Leem, and S. Mitra. 2012. ERSA: Error Resilient System Architecture for Probabilistic Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 4 (April 2012), 546–558. https://doi.org/10.1109/TCAD.2011.2179038

[14] I. Stephen Choi and Yang-Suk Kee. 2015. Energy Efficient Scale-In Clusters with In-Storage Processing for Big-Data Analytics. In *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS '15)*. ACM, New York, NY, USA, 265–273. https://doi.org/10.1145/2818950.2818983

[15] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. 2010. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 497–508. https://doi.org/10.1145/1815961.1816026

[16] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1221–1230. https://doi.org/10.1145/2463676.2465295

[17] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. 2002. The Architecture of the DIVA Processing-in-memory Chip. In *Proceedings of the 16th International Conference on Supercomputing (ICS '02)*. 14–25.

[18] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 449–460. https://doi.org/10.1109/MICRO.2012.48

[19] Y. Fang, H. Li, and X. Li. 2011. A Fault Criticality Evaluation Framework of Digital Systems for Error Tolerant Video Applications. In *2011 Asian Test Symposium*. 329–334. https://doi.org/10.1109/ATS.2011.72

[20] Marc E. Fiuczynski, Richard P. Martin, Tsutomu Owa, and Brian N. Bershad. 1998. SPINE: A Safe Programmable and Integrated Network Environment. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications (EW 8)*. 7–12.

[21] S. Ganapathy, A. Teman, R. Giterman, A. Burg, and G. Karakonstantis. 2015. Approximate computing with unreliable dynamic memories. In *2015 IEEE 13th International New Circuits and Systems Conference (NEWCAS)*. 1–4. https://doi.org/10.1109/NEWCAS.2015.7182027

[22] L.M. Grupp, A.M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf. 2009. Characterizing flash memory: Anomalies, observations, and applications. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*. 24 –33.

[23] 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

[24] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy. 2011. IMPACT: IMPrecise adders for low-power approximate computing. In *IEEE/ACM International Symposium on Low Power Electronics and Design*. 409–414. https://doi.org/10.1109/ISLPED.2011.5993675

[25] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic Knobs for Responsive Power-aware Computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, 199–212. https://doi.org/10.1145/1950365.1950390

[26] Intel Corporation. 2018. INTEL(R) CORE(TM) i7-7700K PROCESSOR. https://www.intel.com/content/www/us/en/products/processors/core/i7-processors/i7-7700k.html.

[27] Intel Corporation. 2018. Intel(R) Optane(TM) Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[28] Djordje Jevdjic, Karin Strauss, Luis Ceze, and Henrique S. Malvar. 2017. Approximate Storage of Compressed and Encrypted Videos. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 361–373. https://doi.org/10.1145/3037697.3037718

[29] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[30] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Proteus: Exploiting Numerical Precision Variability in Deep Neural Networks. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 23, 12 pages. https://doi.org/10.1145/2925426.2926294

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

[32] A. B. Kahng and S. Kang. 2012. Accuracy-configurable adder for approximate arithmetic designs. In *DAC Design Automation Conference 2012*. 820–825. https://doi.org/10.1145/2228360.2228509

[33] 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)*.

[34] Z. M. Kedem, V. J. Mooney, K. K. Muntimadugu, and K. V. Palem. 2011. An approach to energy-error tradeoffs in approximate ripple carry adders. In

IEEE/ACM International Symposium on Low Power Electronics and Design. 211–216. https://doi.org/10.1109/ISLPED.2011.5993638

[35] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. 1998. A Case for Intelligent Disks (IDISKs). *SIGMOD Rec.* 27, 3 (Sept. 1998), 42–52. https://doi.org/10.1145/290593.290602

[36] D. S. Khudia and S. Mahlke. 2014. Harnessing Soft Computations for Low-Budget Fault Tolerance. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 319–330. https://doi.org/10.1109/MICRO.2014.33

[37] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. 2015. Rumba: An online quality management system for approximate computing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 554–566. https://doi.org/10.1145/2749469.2750371

[38] Y. Kim, S. Behroozi, V. Raghunathan, and A. Raghunathan. 2017. AXSERBUS: A quality-configurable approximate serial bus for energy-efficient sensing. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 1–6. https://doi.org/10.1109/ISLPED.2017.8009172

[39] P.M. Kogge. 1994. EXECUBE-A New Architecture for Scaleable MPPs. In *Parallel Processing, 1994. Vol. 1. ICPP 1994. International Conference on*, Vol. 1. 77–84.

[40] 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)*.

[41] Michael A. Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. 2016. Input Responsiveness: Using Canary Inputs to Dynamically Steer Approximation. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 161–176. https://doi.org/10.1145/2908080.2908087

[42] I. Lazaridis and S. Mehrotra. 2004. Approximate selection queries over imprecise data. In *Proceedings. 20th International Conference on Data Engineering*. 140–151. https://doi.org/10.1109/ICDE.2004.1319991

[43] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

[44] Xuanhua Li and Donald Yeung. 2007. Application-Level Correctness and Its Impact on Fault Tolerance. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*. IEEE Computer Society, Washington, DC, USA, 181–192. https://doi.org/10.1109/HPCA.2007.346196

[45] Youjie Li, Jongse Park, Mohammad Alian, Yifan Yuan, Zheng Qu, Peitian Pan, Ren Wang, Alexander Gerhard Schwing, Hadi Esmaeilzadeh, and Nam Sung Kim. 2018. A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks. In *51th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2018)*.

[46] J. Liang, J. Han, and F. Lombardi. 2013. New Metrics for the Reliability of Approximate and Probabilistic Adders. *IEEE Trans. Comput.* 62, 9 (Sept 2013), 1760–1771. https://doi.org/10.1109/TC.2012.146

[47] Jiaheng Lu, Chunbin Lin, Wei Wang, Chen Li, and Haiyong Wang. 2013. String similarity measures and joins with synonyms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 373–384.

[48] A.B. Maccabe, W. Zhu, J. Otto, and R. Riesen. 2002. Experience in offloading protocol processing to a programmable NIC. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*. 67–74.

[49] D. Mahajan, A. Yazdanbaksh, J. Park, B. Thwaites, and H. Esmaeilzadeh. 2016. Towards Statistical Guarantees in Controlling Quality Tradeoffs for Approximate Acceleration. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 66–77. https://doi.org/10.1109/ISCA.2016.16

[50] Ken Mai, T. Paaske, N. Jayasena, R. Ho, W.J. Dally, and M. Horowitz. 2000. Smart Memories: a modular reconfigurable architecture. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*. 161–171.

[51] Gurmeet Singh Manku and Rajeev Motwani. 2002. Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 346–357.

[52] Vadim Markovtsev and Máximo Cuadros. 2017. src-d/kmcuda: 6.0.0-1. https://doi.org/10.5281/zenodo.286944

[53] Jiayuan Meng, S. Chakradhar, and A. Raghunathan. 2009. Best-effort parallel execution framework for Recognition and mining applications. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–12. https://doi.org/10.1109/IPDPS.2009.5160991

[54] Jin Miao, Ku He, Andreas Gerstlauer, and Michael Orshansky. 2012. Modeling and Synthesis of Quality-energy Optimal Approximate Adders. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '12)*. ACM, New York, NY, USA, 728–735. https://doi.org/10.1145/2429384.2429542

[55] Micron Technology, Inc. 2010. MT29F256G08 Datasheet. https://www.micron.com/products/nand-flash/mlc-nand/part-catalog.

[56] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load value approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 127–139.

[57] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. 2010. Quality of service profiling. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 25–34. https://doi.org/10.1145/1806799.1806808

[58] T. Moreau, A. Sampson, and L. Ceze. 2015. Approximate Computing: Making Mobile Systems More Efficient. *IEEE Pervasive Computing* 14, 2 (Apr 2015), 9–13. https://doi.org/10.1109/MPRV.2015.25

[59] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. 2015. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture, HPCA 2015*. Institute of Electrical and Electronics Engineers Inc., 603–614. https://doi.org/10.1109/HPCA.2015.7056066

[60] NVIDIA Corporation. 2019. NVIDIA T4 TENSOR CORE GPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf.

[61] H. Omar, M. Ahmad, and O. Khan. 2017. GraphTuner: An Input Dependence Aware Loop Perforation Scheme for Efficient Execution of Approximated Graph Algorithms. In *2017 IEEE International Conference on Computer Design (ICCD)*. 201–208. https://doi.org/10.1109/ICCD.2017.38

[62] Daniele Jahier Pagliari, Enrico Macii, and Massimo Poncino. 2017. Approximate Energy-Efficient Encoding for Serial Interfaces. *ACM Trans. Des. Autom. Electron. Syst.* 22, 4, Article 64 (May 2017), 25 pages. https://doi.org/10.1145/3041220

[63] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. 1997. Intelligent RAM (IRAM): chips that remember and compute. In *Solid-State Circuits Conference, 1997. Digest of Technical Papers. 43rd ISSCC., 1997 IEEE International*. 224–225. https://doi.org/10.1109/ISSCC.1997.585348

[64] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 377–388.

[65] Colin Perciva. 2006. Matching with Mismatches and Assorted Applications. (2006).

[66] PMC-Sierra. 2014. Flashtec NVMe Controllers. http://pmcs.com/products/storage/flashtec_nvme_controllers/.

[67] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. 2001. Active Disks for Large-Scale Data Processing. *Computer* 34, 6 (June 2001), 68–74. https://doi.org/10.1109/2.928624

[68] Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. 2015. Monitoring and Debugging the Quality of Results in Approximate Programs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 399–411. https://doi.org/10.1145/2694344.2694365

[69] 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.

[70] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based Approximation for Data Parallel Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 35–50. https://doi.org/10.1145/2541940.2541948

[71] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. Sage: Self-tuning approximation for graphics engines. In *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*. IEEE, 13–24.

[72] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-power Computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 164–174. https://doi.org/10.1145/1993498.1993518

[73] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. 2013. Approximate Storage in Solid-state Memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2540708.2540712

[74] Samsung Electronics, Co. Ltd. 2017. Ultra-Low Latency with Samsung Z-NAND SSD. https://www.samsung.com/semiconductor/global.semi.static/Ultra-Low_Latency_with_Samsung_Z-NAND_SSD-0.pdf.

[75] Mohit Saxena, Michael M. Swift, and Yiying Zhang. 2012. FlashTier: A Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 267–280. https://doi.org/10.1145/2168836.2168863

[76] 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
[77] Muhammad Shafique, Waqas Ahmad, Rehan Hafiz, and Jörg Henkel. 2015. A Low Latency Generic Accuracy Configurable Adder. In *Proceedings of the 52Nd Annual Design Automation Conference (DAC '15)*. ACM, New York, NY, USA, Article 86, 6 pages. https://doi.org/10.1145/2744769.2744778
[78] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 124–134. https://doi.org/10.1145/2025113.2025133
[79] Patrice Simard, Bernard Victorri, Yann LeCun, and John Denker. 1992. Tangent Prop - A formalism for specifying selected invariances in an adaptive network. In *Advances in Neural Information Processing Systems 4*, J. E. Moody, S. J. Hanson, and R. P. Lippmann (Eds.). Morgan-Kaufmann, 895–903. http://papers.nips.cc/paper/536-tangent-prop-a-formalism-for-specifying-selected-invariances-in-an-adaptive-network.pdf
[80] Arun Subramaniyan and Reetuparna Das. 2017. Parallel Automata Processor. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 600–612. https://doi.org/10.1145/3079856.3080207
[81] Xin Sui, Andrew Lenharth, Donald S. Fussell, and Keshav Pingali. 2016. Proactive Control of Approximate Programs. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 607–621. https://doi.org/10.1145/2872362.2872402
[82] Devesh Tiwari, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Simona Boboila, and Peter J. Desnoyers. 2012. Reducing Data Movement Costs Using Energy Efficient, Active Computation on SSD. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems (HotPower'12)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=2387869.2387873
[83] Jonathan Ying Fai Tong, David Nagle, and Rob. A. Rutenbar. 2000. Reducing Power by Optimizing the Necessary Precision/Range of Floating-point Arithmetic. *IEEE Trans. Very Large Scale Integr. Syst.* 8, 3 (June 2000), 273–285. https://doi.org/10.1109/92.845894
[84] J. Torrellas. 2012. FlexRAM: Toward an advanced Intelligent Memory system: A retrospective paper. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*. 3–4.
[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] A. K. Verma, P. Brisk, and P. Ienne. 2008. Variable Latency Speculative Addition: A New Paradigm for Arithmetic Circuit Design. In *2008 Design, Automation and Test in Europe*. 1250–1255. https://doi.org/10.1109/DATE.2008.4484850
[87] Zeyi Wen, Jiashuai Shi, Bingsheng He, Qinbin Li, and Jian Chen. 2018. ThunderSVM: A Fast SVM Library on GPUs and CPUs. *To appear in arxiv* (2018).
[88] Zeyi Wen, Jiashuai Shi, Bingsheng He, Qinbin Li, and Jian Chen. 2019. ThunderGBM: Fast GBDTs and Random Forests on GPUs. *To appear in arXiv* (2019).
[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] Yann Collet. 2018. Zstandard - Fast real-time compression algorithm. https://github.com/facebook/zstd/releases/.
[91] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. 2017. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design Test* 34, 2 (April 2017), 60–68. https://doi.org/10.1109/MDAT.2016.2630270
[92] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. 2015. Neural Acceleration for GPU Throughput Processors. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 482–493. https://doi.org/10.1145/2830772.2830810
[93] Rong Ye, Ting Wang, Feng Yuan, Rakesh Kumar, and Qiang Xu. 2013. On Reconfiguration-oriented Approximate Adder Design and Its Application. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '13)*. IEEE Press, Piscataway, NJ, USA, 48–54. http://dl.acm.org/citation.cfm?id=2561828.2561838
[94] Thomas Y. Yeh, Glenn Reinman, Sanjay J. Patel, and Petros Faloutsos. 2009. Fool Me Twice: Exploring and Exploiting Error Tolerance in Physics-based Animation. *ACM Trans. Graph.* 29, 1, Article 5 (Dec. 2009), 11 pages. https://doi.org/10.1145/1640443.1640448
[95] Yong Ho Song. 2017. The OpenSSD Project. http://www.openssd-project.org/wiki/The_OpenSSD_Project.
[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 (EuroSys '18)*. ACM, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/3190508.3190544
[97] Ning Zhu, W. L. Goh, and K. S. Yeo. 2009. An enhanced low-power high-speed Adder For Error-Tolerant application. In *Proceedings of the 2009 12th International Symposium on Integrated Circuits*. 69–72.