UNIVERSITY OF CALIFORNIA
RIVERSIDE

Rethinking the Programming Interface in Future Heterogeneous Computers

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Yu-Chia Liu

December 2022

Dissertation Committee:

Dr. Hung-Wei Tseng, Chairperson
Dr. Rajiv Gupta
Dr. Zhijia Zhao
Dr. Elaheh Sadredini

The Dissertation of Yu-Chia Liu is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to acknowledge Professor Hung-Wei Tseng for his support as the chairman of my committee. I would not have been here without his help and advice.

I would also thank Professor Rajiv Gupta, Professor Zhijia Zhao and Professor Elaheh Sadredini for their valuable comments and feedback as my committee members.

I thank my collaborators and labmates in Hung-Wei's group and all my friends for the ups and downs we spent together.

Finally, I want to thank my parents, especially my mother, for financially and mentally supporting me in pursuing two Master's degrees in Chemical Engineering and Computer Science and a Ph.D. degree in Computer Science.

This degree cannot be completed by myself. This degree is done by everyone I have met and known in this long journey. This journey will move on to a new chapter, and I hope to see you all there.

Chapter 2 contains the material from "NDS: N-dimensional Storage" by Yu-Chia Liu and Hung-Wei Tseng, which is published in the 54th IEEE/ACM International Symposium on Microarchitecture (MICRO 2021). The dissertation author was the first investigator and author of this paper.

Chapter 3 contains the material from "Rethinking Programming Frameworks for In-Storage Processing" by Yu-Chia Liu and Hung-Wei Tseng, which is still in submission. The dissertation author was the first investigator and author of this paper.

The material in Chapter 4 is still under development and planning to be submitted by the end of 2022. The dissertation author is the first investigator of this paper.

To my parents, family and friends for all the support.

ABSTRACT OF THE DISSERTATION

Rethinking the Programming Interface in Future Heterogeneous Computers

by

Yu-Chia Liu

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2022
Dr. Hung-Wei Tseng, Chairperson

Computer systems have become more heterogeneous due to the breakdown of Dennard Scaling and the rapid growth of application demands. In addition to just having general-purpose processors, both factors have pushed modern computers to embrace hardware accelerators that are specialized for such as graphics and AI/ML domains. Besides hardware accelerators, because of the limited bandwidth provided by interconnection among the hardware components, we have seen the development of in-memory processing units and computational storage that also help with performance and thus diminish the boundary between processing units and memory in heterogeneous systems. Even though emerging hardware components in heterogeneous computers provide rich opportunities for performance improvement, programming frameworks that lack flexible programmability and proper interfaces limit the power of heterogeneous systems.

In this dissertation, we envision an efficient and effective programming framework for future heterogeneous computers, and we propose the framework should contain the following characteristics. First, the interface for the heterogeneous systems must fulfill the demand of applications while maintaining the generality for a broad spectrum of applications to minimize the overhead of data representations in different system modules. Second, the programming framework for heterogeneous systems should intelligently identify the opportunities of using available hardware resources to deliver better performance and provide easy programmability. Finally, the programming interface must make applications easily adopt future accelerators or processing units.

I have proposed three different works based on the envision. First, I have proposed NDS, an efficient storage interface that fulfills the various application demands of data objects and gauges the underlying memory-device architectures from application demands to minimize the overhead of transforming data representations. Second, I have proposed ActivePy, a programming framework that automatically identifies the potential code regions for computational storage, generates efficient code, and distributes tasks for the best performance without any programmer's intervention. Lastly, I proposed UDSL, a potential programming paradigm that allows a program to scale easily with the advance of hardware accelerators or any future hardware.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Due to the breakdown of Dennard Scaling and the rapid growth of application demands, the conventional von Neumann architecture has incorporated heterogeneous processing units to fulfill emerging demands of high-performance computing in applications and compute kernels. In addition to only having general-purpose processors in a computer, we have vector processing units, such as GPUs, matrix processing units, such as TPUs, and reconfigurable hardware accelerators, such as FPGA. Those heterogeneous processing units have made complicated computations faster. Besides hardware accelerators, we have included computational storage devices (CSDs) in heterogeneous computers to reduce the data volume going through the system interconnect.

Although emerging hardware components in heterogeneous computers provide rich opportunities for performance improvement, programming interfaces that lack flexible programmability and efficiency limit the power of heterogeneous computers. Figure 1.1 shows the current programming paradigms in a heterogeneous computer. Suppose an application

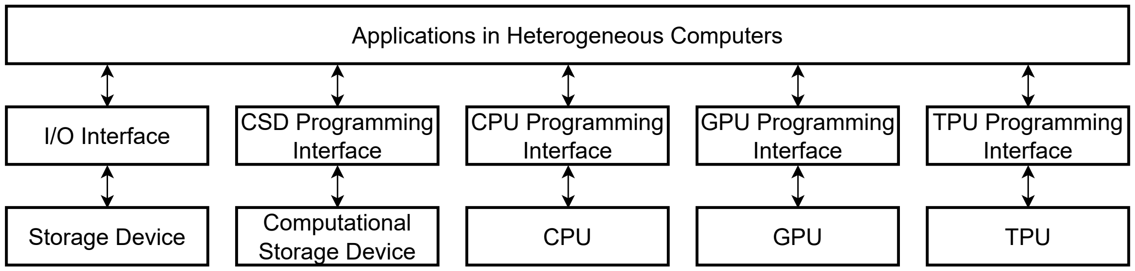

Applications in Heterogeneous Computers
I/O Interface
CSD Programming Interface
CPU Programming Interface
GPU Programming Interface
TPU Programming Interface
Storage Device
Computational Storage Device
CPU
GPU
TPU

Figure 1.1: Current programming paradigm in a heterogeneous computer.

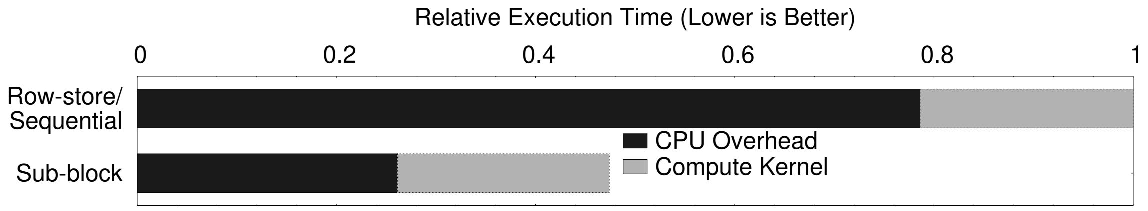


Figure 1.2: The relative execution time of matrix multiplications using row-store format (sequential) and sub-block format, with data already in main memory.

wants to utilize the full power of a heterogeneous computer. In that case, it has to be implemented in different architecture-specific programming interfaces that allow applications to access the underlying devices. We reviewed prior works regarding the hardware/software interfaces in heterogeneous computers and observed three things that limit the power of heterogeneous computers:

Inefficient hardware/software interfaces

The multi-dimensional demands of ML/AI and scientific computing have pushed the compute units from scalar processing to matrix processing. Computer architects have worked intensively to make compute units fulfill the application demands of processing multi-dimensional datasets efficiently. Nowadays, we have not only various ML accelerators that natively support matrix processing but also modern GPU architectures and even the latest CPU architectures incorporated with matrix processing units.

However, the linear memory access interface used for data transfer fails to fulfill the demands of modern applications and processing units. The traditional memory access interfaces, such as C programming language invented 50 years ago, newer programming languages like Python and Rust and the latest NVMe storage interface, require a parameter to describe the starting address and another parameter to describe the requested data length.

Figures 1.2 shows the overhead of marshaling datasets caused by the mismatch between entrenched 1-dimensional data access interface and application demands. In Figures 1.2's example, the application fetches 8K×8K submatrices from a 32K×32K matrix to compute matrix multiplication for the maximum computation throughput. However, due to the constraint of the 1-D interface, all datasets must be stored in either row or column orientation. A programmer must create multiple data access commands requiring a significant amount of CPU instructions to construct requested submatrices. As shown in Figures 1.2, without data marshaling overheads, creating a submatrix from a dataset can be 2.11× faster than storing datasets in linear address spaces.

Inflexible programming interfaces

As shown in Figure 1.1, different architectures have their architecture-specific programming interface. Current programming interfaces for a heterogenous computer require programmers to implement functionalities for each hardware component and schedule the interactions between components in a heterogeneous computer to make resulting applications work correctly. However, this programming model not only makes programmers spend extra efforts in implementing different functions for targeted devices, but also the programming model is inflexible to system dynamics that might cause performance degradation during the execution.

For example, to fetch and preprocess the data from the CSD to the system main memory, copy data from system memory to device memory and invoke compute kernels on the hardware accelerator. Programmers have to do following steps: (1) programmers have to implement compute kernels on the hardware accelerator for better computation throughput. (2) programmers have to implement offloaded tasks to a computational storage device to fetch and preprocess the data to improve the I/O bandwidth. (3) programmers have to write a CPU host application that controls the workflow for computational storage device and hardware accelerators. These implementations are written in different architecture-specific programming interfaces. Furthermore, since the CPU host application is statically compiled in the current programming model, the workflow of the application cannot be adjusted during the runtime, which might cause severe performance degradation when system dynamics change.

Figure 1.3 demonstrates that the inflexible programming model can cause performance issues when the condition of targeted devices changes during the runtime. We have two implementations running under different CSD computing resources in this example. The first version is the "oracle" version, which makes the perfect offloaded task allocation to the CSD that minimize the end-to-end latency of each application. The baseline does

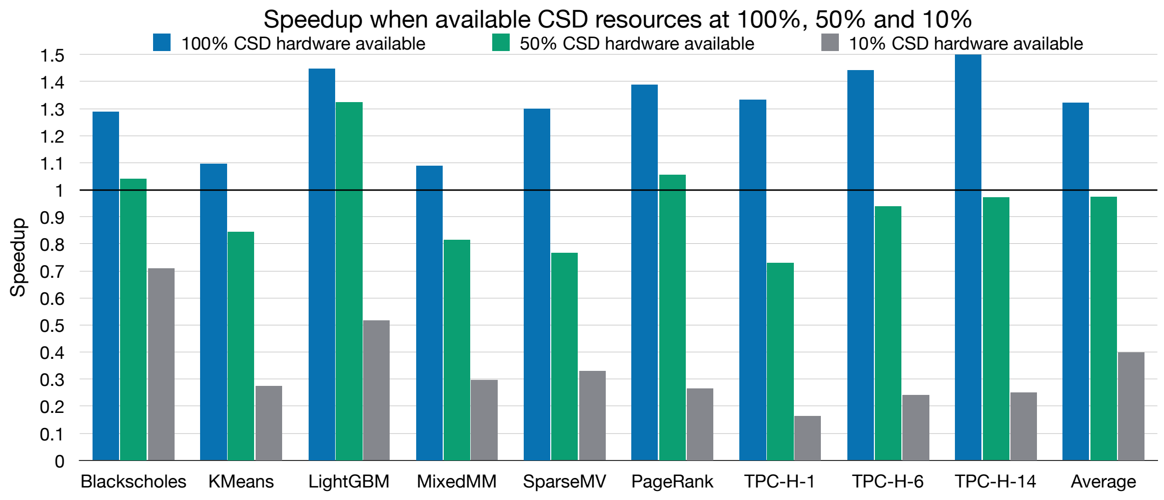


Figure 1.3: The performance of end-to-end latency speedup under different available CSD computing resources.

not offload any task to the CSD, and we use this baseline to compare with the "oracle" version. The horizontal axis shows the set of applications, and the vertical axis shows the speedup of the oracle version compared to the baseline with no task executed on the CSD. The result shows that under 50% available CSD computing resources, some applications in the oracle version have worse performance than the baseline. Not to mention when the CSD only has 10% CSD computing resources available: the oracle version only achieves 0.4× speedup compared to the baseline version.

Unsustainable architecture-specific programming interface

As mentioned in earlier paragraphs, the demands of high-performance computing have pushed computer architects to invent different hardware architectures for different types of computations. Each hardware architecture has its architecture-specific programming interface that supports the features and programming models of the devices. For example, since the CPU is a scalar processor, its programming interface only requires programmers to think about the operation of each element at each step. However, GPU is a vector processor that can spawn a bunch of threads simultaneously. Hence, programmers have to consider the task for each thread and the interactions between threads on each step when using GPU's programming interface. On the other hand, TPU is a specialized hardware accelerator for processing matrices, and its functionality is exposed through an API from a domain-specific framework. In this case, programmers only need to write down one line to compute matrix multiplication on TPU.

It seems architecture-specific programming interfaces are able to support their corresponding architectures well. However, those architecture-specific programming interfaces make applications unsustainable, since programmers must almost rewrite the whole compute kernels for applications in a different programming interface to adopt another hardware architecture. Even though programmers fully understand the architecture-specific programming interfaces and the architectural details, there might be a chance that highly optimized compute kernels cannot outperform standard libraries' APIs implemented with the same algorithm, since some optimizations and internal configurations are not exposed to the outside.

Envision for the ideal programming interface for heterogeneous computers

Inefficient hardware/software interface, inflexible programming interface and unsustainable architecture-specific programming interface limit the power of heterogeneous computers. Figure 1.4 shows an ideal programming interface in a programming paradigm

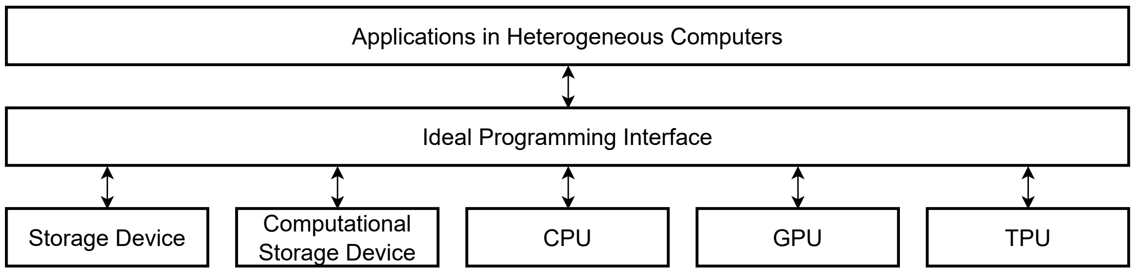

Applications in Heterogeneous Computers
Ideal Programming Interface
Storage Device
Computational Storage Device
CPU
GPU
TPU

Figure 1.4: The ideal programming paradigm in a heterogeneous computers.

that we envision to address problems introduced by those hardware/software interfaces. First, the programming interface should fulfill demands of applications, processing units and memory devices to deliver the full power of heterogenous computers.

Second, the interface should have the intelligence to identify the opportunities of performance gains in the heterogeneous computer, schedule and distribute tasks to components, and dynamically adjust loadings on each component for the perfect balance. Those features should be done by the framework itself, without any programmers' hints and interventions.

Last but no least, the programming interface should be architecture-independent and provide easy programmability and sustainability, so that programmers can concentrate on application side without worrying about architectural details of underlying devices.

This dissertation describes and evaluates all the aspects mentioned above of the envisioned ideal programming interface for heterogeneous computers. Chapter 2- 4 are written in the conference paper format: each chapter has its own introduction, background, problem statements, proposed solution, implementations, methodology, results, related works and conclusion. This dissertation is organized in the following way.

Chapter 2 presents NDS: N-dimensional Storage, which can provide a multi-dimensional data accessing interface that allows high-dimensional applications to naturally describe their desired view of datasets in storage.

Chapter 3 presents ActivePy: Intelligent Programming Framework for Computational Storage Device, which addresses the problems caused by inflexible programming frameworks by automatically identifying the opportunities of using computational storage devices without programmers' interventions.

Chapter 4 presents UDSL: Universal Domain-Specific Languages, which provides an application-specific programming interface that makes applications easily adopt and fully utilize different processing units.

Chapter 5 concludes this dissertation.

Chapter 2

NDS: N-dimensional Storage

2.1 Abstract

Emerging demands of high-performance computing in multi-dimensional applications and compute kernels have pushed processing units to collaborate with matrix or higher-dimensional hardware accelerators. However, the data access interface interacting with those processors and accelerators is inefficient because of the overheads caused by the entrenched linear-space abstraction. Besides the demand mismatch between the data access interface and applications, the data access interface often ignores the different types of parallel access of modern memory/storage systems that support multi-dimensionality. NDS provides a novel and multi-dimensional memory/storage system that fulfills the different demands of different applications and modern hardware accelerators. NDS offers an abstraction of underlying memory arrays that allows applications to naturally describe data objects by using coordinates and shapes in application-defined multi-dimensional spaces, and therefore helps to avoid the overheads caused by data-object transformations. NDS

also gauges the application demands from underlying memory-device architectures through proposed building blocks and the space translation layer that intelligently translate multi-dimensional coordinates to the physical data layout to maximize memory access bandwidth and minimize the overheads of presenting objects to arbitrary applications. We evaluate NDS with a set of workloads in the fields of linear/tensor algebra, graph and data mining on a custom-built system, and our result shows a 5.73× speedup with appropriate architectural support.

2.2 Introduction

Because of the brokenness of Dennard scaling and the demand for multi-dimensional applications and compute kernels, traditional von Neunman computers have started to collaborate with heterogeneous processing units that support matrix or higher-dimensional processing models. The internal structures of modern memory and non-volatile storage devices in heterogeneous computers are also multi-dimensional. Modern memory technologies typically allow data to be accessed in parallel through several independent aspects, such as banks and channels, to utilize parallelism for higher access bandwidth. Therefore, each chunk of datasets has a unique combination of row, column, bank, rank, channel and etc to describe their locations in the memory array.

Applications, hardware accelerators and memory architectures are multi-dimensional, but the memory/storage systems are still exposed through an entrenched, one-dimensional addressing mode. The one-dimensional addressing mode requires applications to serialize high-dimensional data along a select dimension, row or column, for example, for storing data in a memory device. Another application must deserialize the raw data from the memory device to data objects in the desired shape for compute kernels to retrieve data. Such abstraction results in the processing overhead of serialization and deserialization and the inefficient use of interconnect and memory-device bandwidths, and those overheads lead to low utilization of high-dimensional, data-intensive compute kernels in hardware accelerators.

Prior works have proposed application-defined, efficient data storage format [153, 12] and optimized algorithms [184, 1] to address the mismatch between memory devices, memory/storage interface and applications. However, finding the most efficient data format is challenging. First, the optimal layout may vary from different memory architectures, and those internal characteristics of memory devices are hidden from applications. Second, the optimal layout may be different among computer kernels in applications. Third, even though we know the optimal layout for the targeted compute kernel and the target memory device, the desired structures of data objects for maximizing the computation throughput and maximizing the memory access bandwidth may not match each other.

We proposed NDS, N-Dimensional Storage, to address the aforementioned mismatch between modern memory architecture, linear memory abstraction and the demand for modern hardware-accelerated, high-dimensional compute kernels/applications. NDS provides an interface that allows applications to define their desired abstractions of data objects. Inside NDS, the space-translation layer (STL) gauges application demands and memory-device characteristics by breaking down datasets into building blocks that match the granularity for optimal data access bandwidth in memory devices. The STL records each dataset's dimensionality and the mapping of the dataset's building blocks. When an application requests a data object through NDS, the STL translates application-defined abstractions to physical locations of building blocks through the data structures maintained by STL and dynamically composes/decomposes data from/into building blocks.

NDS resolves the aforementioned overheads and offers several benefits. First, NDS migrates the marshaling overhead because NDS does not require applications to serialize/deserialize the dataset through conventional linear memory abstraction. Second, NDS maximizes the utilization of processing units by allowing applications to have the optimal data structures for compute kernels. Third, NDS fully utilizes the system-interconnect bandwidth and memory-device internal bandwidth by accessing data through building blocks to take advantage of device-level parallelism.

We built different prototype NDS systems to investigate the trade-offs of NDS features within different system components. The software-only NDS implementation demonstrates the effectiveness of building blocks that lowers the overhead of constructing multi-dimensional objects for application demands, and it achieves 5.07× speedup among a broad range of applications, including large-scale, dense matrix/tensor algebra applications, graph traversal applications, and high-dimensional data-mining applications. With appropriately architectural support, we are able to implement NDS features on an SSD controller to further achieve 5.73× speedup for the same set of applications, since the hardware-supported NDS implementation efficiently utilizes internal parallelism in memory devices and reduces the number of communications crossing the I/O interface that lowers the overhead on a host computer.

NDS makes the following contributions:

(1) It is the first work to present an application-defined, multi-dimensional memory/storage abstraction as an alternative to the entrenched linear memory abstraction.

(2) It demonstrates that granularities and dimensionalities of data accesses are different among devices and that simply optimizing application-based file-storage formats is insufficient.

(3) It presents an efficient data-allocation strategy that gives programmers and applications an agnostic memory/storage data layout while allowing arbitrary data-access patterns to fully utilize the interconnect/device bandwidth and efficiently construct multi-dimensional application objects.

(4) It evaluates different NDS system architectures and shows the performance gains from each system architecture.

2.3 Background

This section explains the effects of mismatching dimensionalities among applications, memory devices and memory abstractions on application performance, the challenges of addressing these mismatches and the limitations of previously proposed solutions.

2.3.1 Modern heterogeneous architectures

Figure 2.1 shows the key hardware components for processing high-dimensional datasets in a modern heterogeneous computer. Such a computer typically uses hardware accelerators (e.g., GPUs in Figure 2.1) to perform compute kernels on high-dimensional

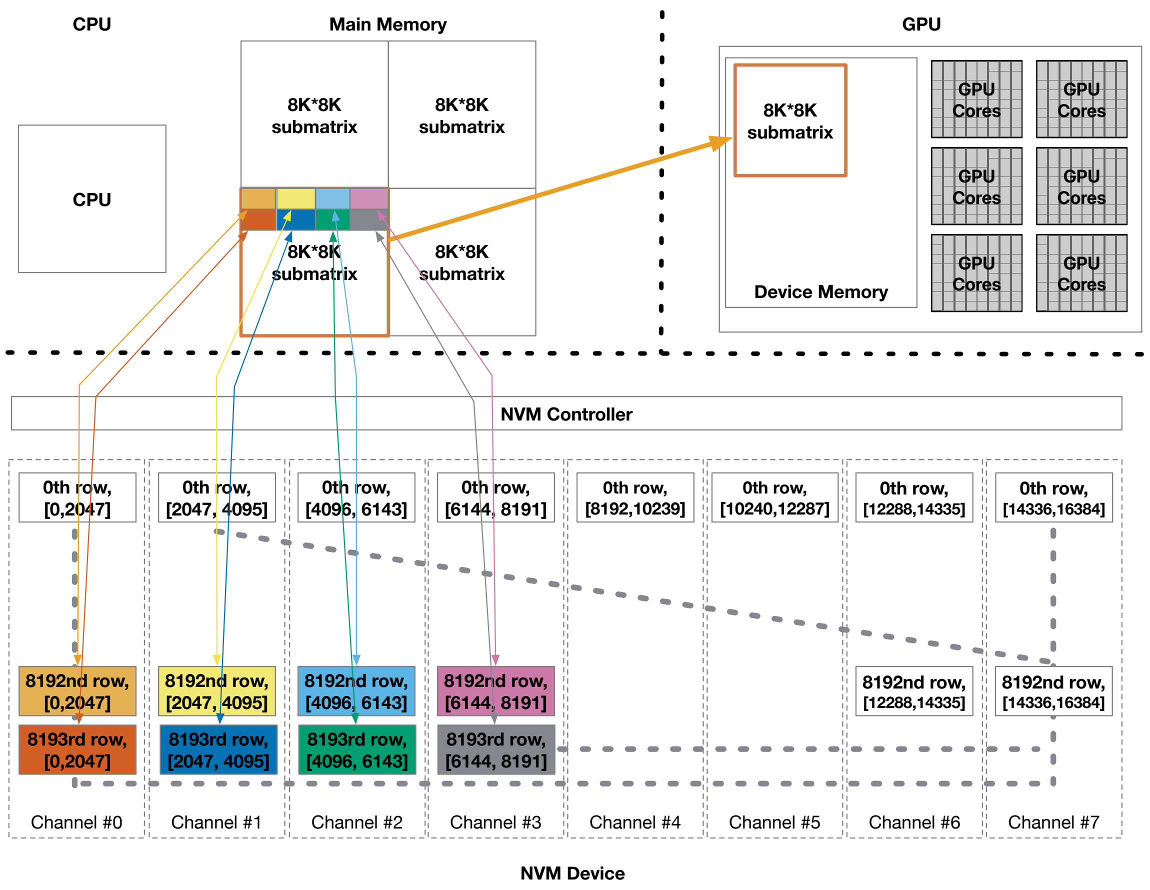


Figure 2.1: The blocked-matrix-multiplication datapath in a hardware-accelerated computing system having a conventional storage-system hierarchy with non-volatile memory (NVM)

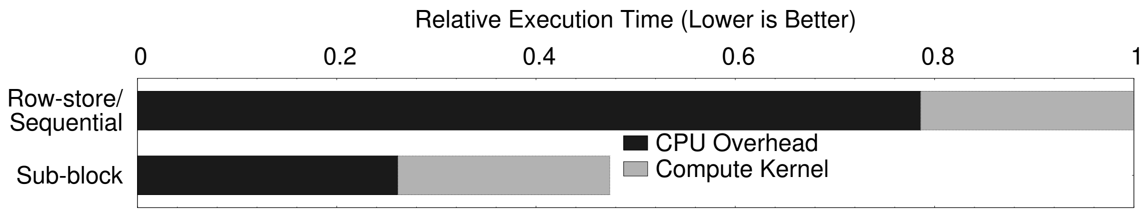
datasets efficiently, since modern hardware accelerators provide two types of processing elements: (1) vector processing units (e.g., GPU cores) and (2) matrix processing units (e.g., Tensor Cores (TCUs) or Tensor Processing Units (TPUs)) to more efficiently process high-dimensional datasets.

On the data storage side, a modern computer has a high-performance solid-state drive (SSD) that stores data in non-volatile memory technologies, such as NAND flash memory and phase-change memory (PCM) [164], and they have different basic access granularities (e.g., page in flash memory) because of different memory architectures. Similar to DRAM technologies, modern SSDs organize their memory arrays into channels, with all parallel channels capable of accepting unique requests simultaneously, and the SSD controllers exploit channel-level and bank-level parallelism to improve data access bandwidth. However, unlike DRAM controllers, SSD controllers must carefully manage mapping between the logical addresses that software uses and physical addresses in memory, since commercialized non-volatile memory technologies have limited program-erase cycles.

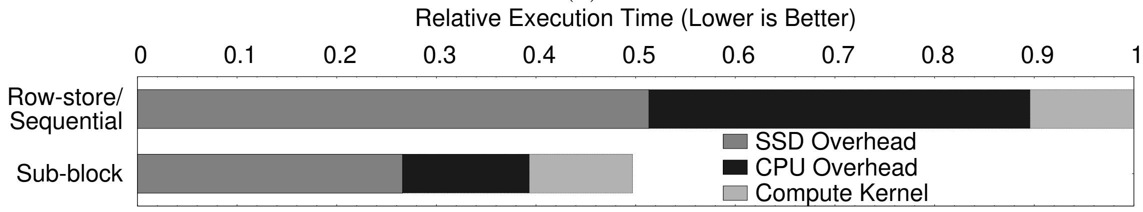
However, modern storage devices are exposed through linear address spaces such as logical block addresses (LBAs) for data access. As a result, producers of high-dimensional datasets must reduce data dimensionality to 1-dimensional representations, typically with row-oriented or column-oriented storage formats. If a hardware-accelerated compute kernel needs to compute on datasets, an application invoking the compute kernel must first fetch each row or column of requested data from the storage system through the 1-dimensional addressing interface. Then, the application assembles each received data chunk into memory objects with a shape that satisfies the demands of the targeted compute kernel. Finally, the application copies the created memory object from the host main memory to the accelerator's device memory so the compute kernel can use the object.

Figure 2.1 also shows matrix multiplication (MM) performed using a modern GPU. Each raw input matrix is 16K$\times$16K and presented in a row-oriented storage format, which is generally considered the most efficient for CPU-based compute kernels. The GPU kernel delivers maximum computation throughput if the MM kernel is performed on 8K$\times$8K submatrices. The SSD has 8 parallel channels and 8 KB pages, where each SSD page stores 2K elements encoded in IEEE-754 32-bit floating-point format, so each matrix row occupies logically consecutive 8 pages in the SSD's LBA. In conventional SSD design, these consecutive pages are typically striped across different channels to enable sequential parallel accesses in LBAs, since most file systems and applications assume that when the devices access data sequentially, the underlying storage devices perform more efficiently. As a result, each of the aforementioned, logically consecutive 8 pages will be physically located in a distinct channel respectively.

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



(a)



(b)

Figure 2.2: The relative execution time of matrix multiplications using row-store format (sequential) and sub-block format, with (a) data already in main memory and (b) from the SSD.

[P1]: The overhead of marshalling input data.

An application must use multiple CPU instructions to calculate the mapping between the raw-data offset and the target memory locations to re-constructure stored data from row-/column-orientation into the dimensionality that hardware accelerators need. Then the application must use CPU instructions again to issue I/O requests through the system software stack to fetch pieces of raw data and place (and potentially convert) the received data chunks in their designated memory locations. Figure 2.1 shows that if the application needs to fetch an 8K×8K submatrix, the row-store format will require the program to calculate offsets 8,192 times and issue 8,192 I/O requests to fetch 8,192 rows that contain the desired submatrix.

Figure 2.2(a) illustrates the marshaling overhead on the CPU, with the raw data already in the system main memory before the compute kernel launches. The system uses an AMD RyZen 3700X CPU and Nvidia's RTX 2080 GPU. The baseline comprises pipelined matrix multiplications— that is, the multiplication of two 32K$\times$32K matrices using sub-blocks of 8K$\times$8K matrices in the aforementioned example. As the source dataset uses a sequential row-store format, the baseline needs extra CPU instructions to reshape and form 8Kx8K matrices before the compute kernel starts. In contrast, the alternate configuration with sub-block format already has 8Kx8K submatrices stored in main memory and does not require the CPU to prepare data for compute kernels. As a result, the sub-block configuration can be 2.11x faster than the sequential baseline configuration.

[P2]: Underutilization of interconnect bandwidth.

Besides the overhead of marshaling input data, the mismatching demands between the storage abstraction and compute kernels lead to the underutilized interconnection bandwidth on the I/O side, since I/O requests to fetch data chunks are typically smaller than the sizes that can amortize the overhead of each I/O transaction. In the modern NVMe [6] interface, interconnect bandwidth saturates if each request is larger than 2 MB. However, in the Figure 2.1's example that fetches a row containing 8K elements for an 8K$\times$8K sub-matrix, each I/O request is only for 32 KB of data, so the interconnect can only achieve 66% of the peak bandwidth. An application may be designed to firstly fetch consecutive chunks of storage data into a large memory buffer, and then gradually copy the buffered data into designated memory locations to address the aforementioned issue. However, this approach creates three performance issues: (1) it generates traffic from copying small data blocks on the CPU-memory bus, (2) it wastes precious main-memory capacity for memory buffers, and (3) it fetches data elements that the application might not use immediately, so they might be fetched again due to limited buffer memory capacity and therefore wastes I/O bandwidth.

[P3]: Underutilization of device bandwidth.

One-dimensional LBA space requires an application to serialize the original multi-dimensional data structure when storing the data in storage. However, if the data dimensionality or the access pattern does not fit the storage device's internal structure, the application consuming the serialized data is likely to underutilize the device's internal bandwidth. As in Figure 2.1, when fetching 8K$\times$8K submatrices, a program can only utilize 50% of the available parallel channels since the requested data only reside in 4 of the 8 channels.

Figure 2.2(b) shows the impact of such underutilized device bandwidth in a MM application by extending the situation in Figure 2.2(a) for data placed in an SSD with 32 parallel channels. Besides the CPU overhead of transforming rows into submatrices, the baseline spends 1.92$\times$ more time fetching data compared to an SSD configuration with optimal data layout for the workload by storing 8K$\times$8K submatrices consecutively due to the underutilized device bandwidth.

2.3.2 Challenges

In Section 2.3.1, we observe that the locality and dimensionality mismatches of the data layout between storage devices, compute kernels and storage interface need to be addressed by fulfilling the following requirements: (1) The layout must minimize the

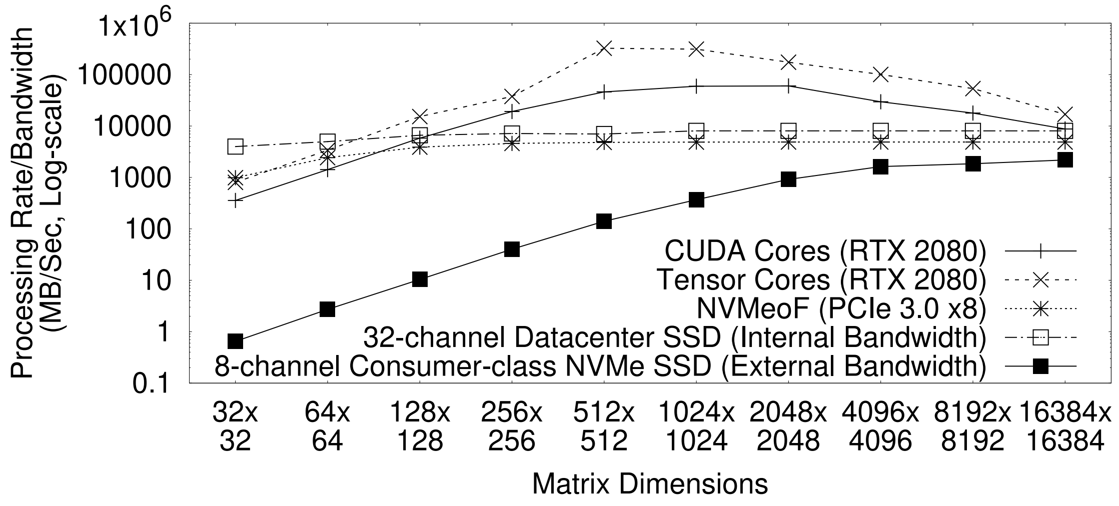


Figure 2.3: The effective data processing rates or I/O bandwidth of system components with various matrix sizes

overhead of reassembling data for compute kernels and hardware accelerators. (2) The layout must saturate the interconnect bandwidth and reduce the number of communications because of I/O requests. (3) The layout must allow the application to fully utilize the internal bandwidth of the storage device.

However, designing an optimal data layout that satisfies the above three requirements under the traditional one-dimensional addressing mode is challenging. The followings are the reasons:

[C1]: Unavailability of internal memory-device architecture to applications.

Physical memory locations in a modern storage device are only exposed through LBAs, so that applications have no information about the device's architecture. Although existing NVMe commands allow an application to query the parameters of an underlying device and to optimize the LBA layout for an individual device, mechanisms, such as garbage collection and wear-level functions in the SSD management layer (e.g., the flash-translation layer (FTL) and PCM translation layers [181, 135, 58]), can lead to data-location shuffling and thus suboptimal performance.

In addition, because internal designs and hardware structures are different among devices, and that information is hidden from the application, the application is unable to optimize the LBA layout for all storage devices. Figure 2.3 shows each hardware component's data supply rate or consumption rate. We fetch different shapes of submatrices in a 4 GB matrix sequentially from the SSD's LBA space from two flash-based SSDs, one with 32 channels and the other with only 8 channels, and compare their maximum bandwidth. Figure 2.3 shows that the 32-channel SSD can utilize the maximum bandwidth with the 512×512 matrices fetched by applications, but 8-channel SSD only achieves its maximum bandwidth when the matrix size is larger than 4K×4K. This comparison underscores the difference in optimal granularity due to different internal storage-device architectures.

[C2]: Unpredictability of optimal dimensionality in compute kernels.

There exists no optimal layout that can maximize the efficiency of all applications or accelerators, since any application can potentially share a dataset, and a compute kernel can execute on various computing resources. For example, a row-/column-oriented pair-wise matrix can maximize GPU computation, but cannot be efficient for matrix-multiplication kernels. Figure 2.3 also shows the comparison of the data-processing rates and input data sizes for processors/accelerators between different processing models—the vector processing model (used with general matrix multiply, GEMM, from Nvidia's cuBLAS [106] library) on conventional GPU cores (CUDA cores), and the 2-D matrix-processing model used with

Nvidia's Tensor Cores. To measure pure compute-kernel performance, we made all matrices available on the GPU device memory before the GEMM functions were invoked. Setting aside the significant performance lead in Tensor Cores, the optimal submatrix size that maximizes performance on the CUDA cores was found to be 2048×2048, whereas the optimal submatrix size for the Tensor Cores was 512×512.

[C3]: Demand mismatch between storage devices and compute kernels.

Even though a user can narrow down the use of a dataset to a specific type of compute kernel, hardware accelerator, and dedicated storage device, there is still no guarantee that the optimal input data for the hardware accelerator will match the layout that can fully utilize the internal bandwidth of the storage device. For the situation covered in Figure 2.3, our matrix-multiplication kernel running on TCUs worked best on consuming 512×512 submatrices, but the I/O bandwidth for the 8-channel consumer SSD is maximized by providing 16K×16K submatrices.

2.3.3 Alternatives

Prior work has proposed solutions to address mismatching demands of dimensionalities in modern heterogeneous computers through (1) more efficient data storage formats, (2) libraries supporting high-dimensional addressing, and (3) offloading of I/O operations to intelligent storage devices. Unfortunately, none of the three approaches tackles the challenges mentioned in Section 2.3.2.

File formats

Many works have focused on the optimal data layouts for the dense data storage. Apache's Parquet [11] and ORC [12] offer efficient columnar storage formats, and Arvo [10] offers row stores for Hadoop. Besides efficient row-/column-storage, Google's protocol buffers [157] and JSON's binary representation (BSON) [120] propose binary-encoded internal data representations to reduce the CPU processing overhead of deserializations. Albis [153] fuses features of columnar/row-major stores with binary-encoded formats to exploit both spatial locality and low-overhead deserialization. File formats for specific domains are also available to address the demand of popular compute kernels in application subsets, such as aggregate genomic data (AGD) [29] for biological applications, G-Store [79] for graph applications, and JSOI [127] and ONNX [149] for machine learning models. Frameworks that automatically generate data layout for GPU kernels [86, 94] also exist. However, none of these formats can address challenges **[C1]** and **[C3]**, leading to problems **[P2]** and **[P3]**.

High-dimensional software I/O libraries

Cloud storage systems, such as N5 [132] and Zarr [7], use n-dimensional addressing modes and chunk raw data to better utilize I/O bandwidth. To reduce the overhead in applications that create high-dimensional data objects, besides n-dimensional addressing modes, Zarr [7] also uses data compression to maximize the effective I/O bandwidth. However, these software libraries still access data through linear memory/storage address spaces, which leads to problems **[P2]** and **[P3]**.

In-storage processing (ISP)

Near-data processing (NDP) or ISP models can use a device's internal bandwidth by leveraging tasks on the controller within the device to improve the bandwidth/memory demand and reduce CPU overhead of present data objects that fulfills compute kernels' demands. Existing ISP frameworks can transform raw data into application objects for general-purpose compute kernels [155], mixed-precision/approximate computing workloads [62] or specialized applications such as graph applications [97]. However, the above ISP approaches fail to address [**C3**], since the in-device data layout does not fit the access patterns of ISP code. As a result, proposed solutions cannot fully utilize interconnect bandwidth to efficiently produce objects for applications.

2.4 Overview of NDS

NDS follows three key design principles to address the issues and challenges outlined in Section 2.3.1 and Section 2.3.2.

Provide commands in multi-dimensional addressing modes.

As noted, traditional 1-dimensional address mode forces applications to serialize multi-dimensional data into one-dimensional data and leads to [**P1**], [**P2**] and [**P3**]. To address those problems, NDS offers a set of multi-dimensional storage and I/O commands that enable a single I/O request to manage data in arbitrary dimensions. NDS thus minimizes the number of I/O requests while maximizing the data volume in each I/O request.

Fulfill compute-kernel demands through application-defined, multi-dimensional address spaces.

To address [**C2**], each application can define its own view (e.g., shape or dimensionality) of a data object in NDS, regardless of (1) the original data layout and dimensionality in data storage and (2) the view of the data object defined by other applications. NDS automatically and implicitly transforms a data object into an application's required dimensionality. As an application accesses data objects using its natural view of the address space, NDS further reduces the overhead caused by data-dimensionality conversion ([**P1**]), and because NDS presents data in an optimal layout for the targeted compute kernel, the compute kernel can work more efficiently.

Make software agnostic to storage-device characteristics.

NDS decouples an application's view from storage-device granularity and storage data layout and therefore reduces programming complexity and broadens data-layout applicability, which addresses [**C1**], [**C2**] and [**C3**]. NDS uses the dimensionality from a dataset producer to determine storage space and intelligently restructure storage data into *building blocks*—collections of basic access units (e.g., pages) serving as internal structures of the memory device. These building blocks maximize internal access bandwidth and allow low-overhead conversion among different dimensionalities.

Figure 2.4 shows how NDS presents an architecture to support multi-dimensional data accesses on a conceptual level. NDS receives optimal access granularities and serves as an intermediary to fill the demands of both application and device by working between the

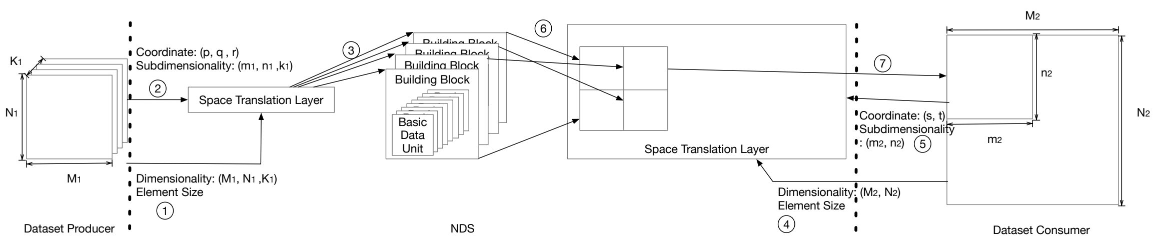


Figure 2.4: An overview of NDS data-access operations

software stack and storage hardware, In NDS, each in-storage and in-application address space is determined by three properties:

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

Following the numbering system in Figure 2.4, NDS's dataset producer creates a multi-dimensional space using the three essential properties describing the address space (①). The STL then parses the data structure, determines the dimensionality of building blocks in the address space, and returns the space identifier to the software. When transferring data into NDS (②), an application passes the source memory location and the address-space data position using two parameters:

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

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

We use Figure 2.5 to demonstrate the revised request by using NDS, where the original request is shown in Figure 2.1. In Figure 2.5, the producer application creates a three-dimensional space with a size of 8,192$\times$8,192$\times$4. After gauging the device capabilities and the dimensionality of the space, NDS uses 16,384 building blocks, where each building block is sized as 128$\times$128. In this way, NDS maximizes the access bandwidth during the store operation, since 8 logically consecutive pages in a building block are filled in 8 different channels in the same bank.

For an application that treats the dataset in the space as four 8,192$\times$8,192 sub-matrices and requests the [1,0] sub-matrice (elements [8192, 0] through [16383, 8191] in a

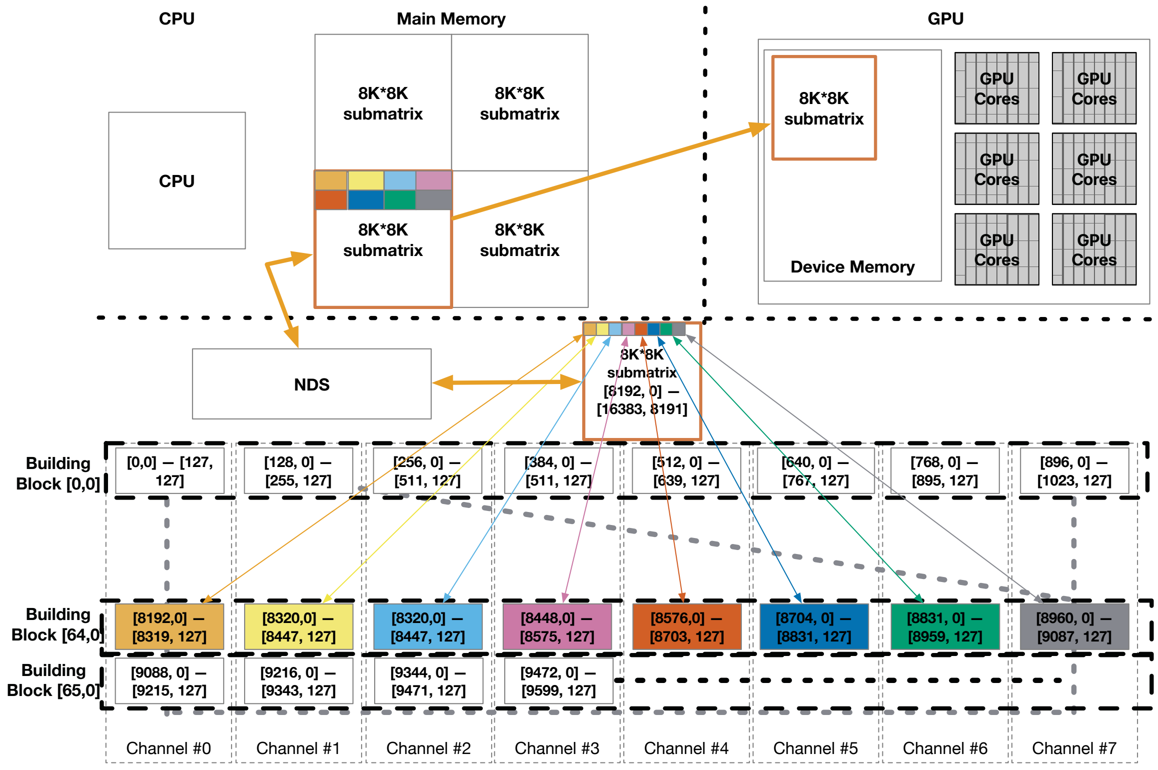


Figure 2.5: Accessing a submatrix from NDS

16,384×16,384 matrix), the consumer simply needs to send one I/O command describing the location of the sub-matrix in the space and NDS translates this single request into locations to 4,096 (64×64) building blocks. Each access to the building block can fully utilize the internal bandwidth of the device, as each building block consists of pages from different channels in the same bank. In this way, NDS can issue an access request to a building block in another bank in the next cycle to pipeline building-block accesses. In contrast, the non-NDS approach in Figure 2.1 consistently wastes approximately 50% of the internal parallelism even when accesses are fully pipelined.

Upon receiving each building block, NDS dynamically assembles the datasets into 8,192×8,192 sub-matrices that the compute kernel requires and delivers each consecutive data chunk to the host computer as soon as the chunk is assembled. The application does not need CPU instructions to restructure the object, because the data are already presented in the optimal layout for the compute kernel. By using NDS, the complete process requires only one I/O command from the host computer. In contrast, the process in Figure 2.1 requires the host computer to either intensively access the CPU memory bus and use CPU instructions to relocate data in the memory buffer, or requires issuing an excessive number of I/O requests.

2.5 The Space-Translation Layer (STL)

The STL is the core of NDS that creates data structures for maintaining multi-dimensional address spaces and determines how building blocks are used. The STL receives access requests containing coordinates in arbitrary dimensionalities and translates these requests into hardware commands to access the physical data locations of building blocks. After address translations, the STL dynamically transforms and presents the address space in the desired application view by assembling building blocks. Compared to the STL, conventional FTLs only present data in 1-dimensional addressing modes and rely on applications to adjust the data's dimensionality.

2.5.1 Building blocks

A building block is a fixed-size logical chunk of data storage in NDS, and it is considered as the basic unit of data-storage elements that are colocated in their original address space. A building block contains a set of physical memory-access units (e.g., pages in NAND flash-based storage). The STL determines the size of a building block and assigns memory units to a building block by considering the data access granularity that maximizes the device's internal bandwidth and minimizes the access latency. To achieve the optimal performance of data access, a complete building block stores its data in units available through all parallel channels. Therefore, the STL determines the minimum building-block size (BB_Size_{min}) allowable for the underlying storage device as

$$\begin{aligned} BB\_Size_{min} = Num\_Parallel\_Requests_{MAX} \times \\ Granularity_{Basic\_Access} \end{aligned} \tag{2.1}$$

where $Num\_Parallel\_Requests_{MAX}$ represents the maximum number of parallel requests a memory device can perform (e.g., the number of parallel device channels),

and $Granularity_{Basic\_Access}$ represents the fine-grained structure of the aforementioned basic-access units. For example, if an SSD contains flash chips having 4 KB pages and 8 parallel channels, the minimum size of building blocks will be defined as 32 KB (4 KB $\times$

8) by using Equation 2.1. A minimum building block will consist of 8 pages, each from a different parallel channel in the device. BB_Size_{min} is identical for devices with the same internal architecture, but it can vary among devices with different memory architectures.

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

$$BB = N \times (2^{\lceil \frac{log_2 \frac{BB\_Size_{min}}{N}}{2} \rceil})^2 \tag{2.2}$$

where N is the size of each element in the space, and with each dimension in the building block storing $2^{\lceil \frac{log_2 \frac{BB\_Size_{min}}{N}}{2} \rceil}$ elements. For example, when BB_Size_{min} is 32 KB (4 KB $\times$ 8 channels), and the application creates a 2-D space to store 4-byte elements, the STL will use 64 KB as the size of each building block. Each building block will contain 2 pages from each channel and store 128 elements in each dimension.

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

$$3D\_BB\_Size_{min} = BB\_Size_{min} \times Num\_Banks \tag{2.3}$$

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

$$3D\_BB = N \times (2^{\lceil \frac{log_2 \frac{BB\_Size_{min}}{N}}{3} \rceil})^3 \tag{2.4}$$

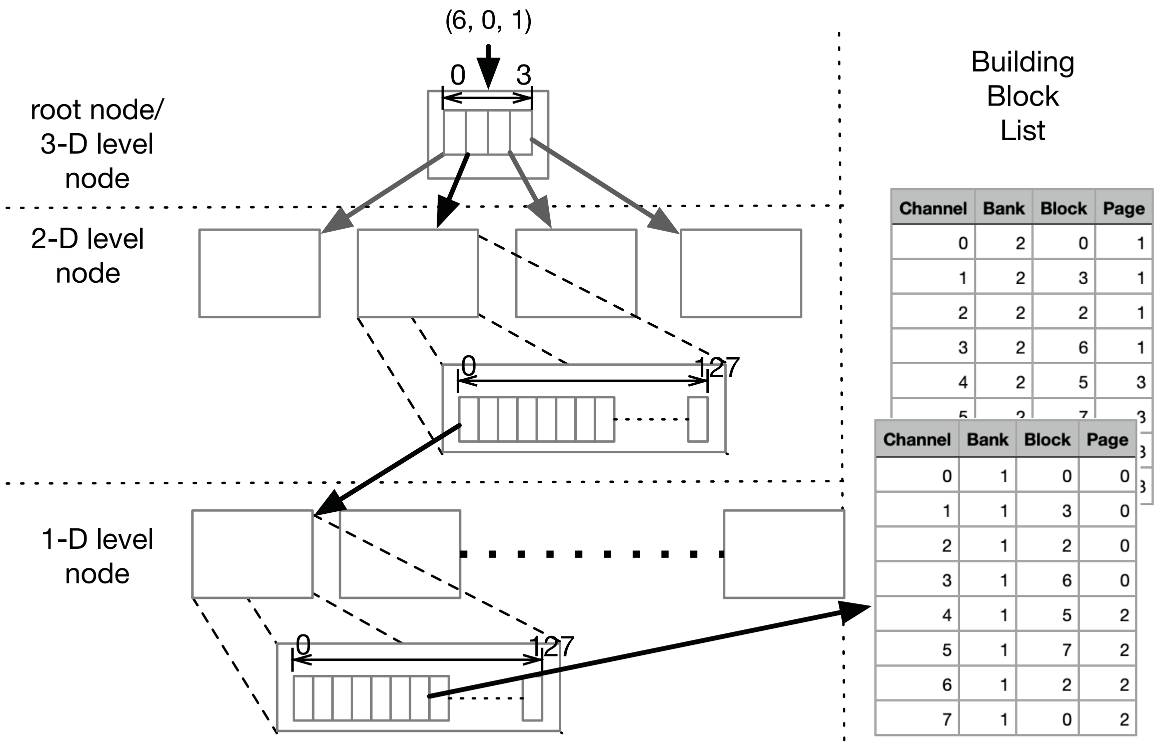
where each dimension stores $2^{\lceil \frac{log_2 \frac{BB\_Size_{min}}{N}}{3} \rceil}$ elements.

If the memory device provides another level of parallelism, NDS's STL can further extend Equation 2.3 to make a building block. However, because modern volatile/non-volatile devices only exploit bank-level and channel-level parallelism, and hardware accelerators only support 1-D or 2-D operations (e.g., TPUs [66] and Tensor Cores), NDS as yet supports only 1-D, 2-D, or 3-D building blocks.

In the rest of this chapter, we use (bb_1, bb_2, ..., bb_n) to represent the dimensionality of a building block in an n-dimensional space, where bb_i represents the size of the ith-order dimension in the space. Since NDS currently supports 1-D to 3-D building blocks, NDS sets the bb_i value to 1 when $i > 3$. We use (d_1, d_2, ..., d_n) to represent the size of an n-dimensional (N-D) space where d_i stands for the size of the ith-order dimension.

2.5.2 Locating and allocating building blocks

Building blocks are the basic granularity in NDS, while the STL is the core of NDS that helps to maintain the mapping building blocks to their physical locations. The STL maintains an N-level B-tree data structure for an N-D space that locates building blocks. In each B-tree, the root node corresponds to the highest order in the N-D space, the second level from the root node corresponds to the second-highest order in the space, and the leaf node corresponds to the lowest order of the space. In each level of B-tree nodes, the node



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

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

Figure 2.6: An exemplary B-tree structure of STL

degree is $\frac{d_i}{bb_i}$, where d_i represents the size of the ith order spatial dimension. For a non-leaf node, each entry is a pointer to a next-level leaf node in the STL's memory space. For a leaf node, each entry points to a list of physical memory locations for basic access units (e.g., pages in an SSD) that belong to the corresponding building block. The list of access-unit locations is sorted according to the sequential order of the units in the building block.

Figure 2.6 illustrates such a B-tree structure for a 3-D space with 2-D building blocks: The multidimensional space is an (8192, 8192, 4) space that uses (128, 128) building blocks, the example shown in Figure 2.5. Since the space has three dimensions, the B-tree has three levels. If a request goes to a building block with coordinate (6, 0, 1), the tree will visit the 1st entry in the root node to reach the next level and use the 0th entry in the 2-D node to reach the 1-D/leaf node. The leaf node points to a list of pages in storage. Then, the STL will issue requests to fetch all pages in the list in parallel to access a building block.

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

In a typical NVM storage device that exploits bank-level and channel-level parallelism, the STL can take several different approaches to access-unit selection, as follows: (1) If the STL has not created the building block, the STL randomly chooses an access unit from a channel and a bank. Alternatively, (2) if the building block exists, the STL picks an access unit from a parallel channel that the building block uses the least (a *least-used* channel); in this case, the unit is from the same bank as the most recently allocated unit in the building block. (3) The STL can also select an access unit from an unused or least-used bank if the building block has used a unit from every channel in the bank. (4) If the STL cannot find a page through the above rules because the building block has used a unit from every channel and bank, the STL chooses one of the least-used banks and repeats (1) through (3).

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

2.5.3 The space translator

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

As Section 2.4 describes, an application can work with its own multi-dimensional space (e.g., m-dimension space) of size δ_1, δ_2, ..., δ_m regardless of that space's representation in storage. With this m-dimensional space, an application can access NDS using a coordinate (x_1, x_2, ..., x_m) and the sub-dimensionality (β_1, β_2, ..., β_m) that represents the partition of the requested data.

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

$$\begin{aligned} Y_i = \{a \in \mathbb{Z} : a \geq \lfloor \frac{\sum_{j=2}^{m}[(x_j \times \beta_j)\prod_{k=1}^{j-1}\delta_k] + x_1 \times \beta_1}{\prod_{k=1}^{i-1} bb_i} \rfloor \\ and\ a \leq \lfloor \frac{\sum_{j=2}^{m}\{[(x_j+1) \times \beta_j)]\prod_{k=1}^{j-1}\delta_k\} + (x_1+1) \times \beta_1}{\prod_{k=1}^{i-1} bb_i} \rfloor\} \end{aligned} \tag{2.5}$$

2.5.4 Data accesses, assembly, and composition

When the space translator decomposes a request into building-block coordinate accesses, the STL walks through the B-tree structure to issue requests (and allocate space if necessary) to each access unit, as described in Section 2.5.2.

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

For a read request, the STL will access a set of building blocks and creates a buffer in the STL's memory space to place the received data into logical locations, from the application's perspective, with object assembly determined by the translation process described in Section 2.5.3. As soon as a segment of the assembled object reaches the optimal data-exchange volume for the system interconnect, NDS starts to move the assembled data. Once the STL has assembled all data for an application request and has moved the assembled multi-dimensional object to the application's memory space, NDS designates the request as complete and returns to processing other application requests.

2.6 The NDS Prototype

This section describes a proof-of-concept storage system that demonstrates the validity and value of NDS. This prototypical NDS system includes an application programming interface (API) and implements the STL functions, the core of NDS. We implemented a hardware-assisted NDS and a software-only design for comparison by extending a baseline SSD to fundamentally address the issues and challenges mentioned in Section 2.3.

2.6.1 APIs

Because NDS accepts high-dimensional coordinates instead of conventional 1-D offsets, NDS needs new system-level API functions that allow applications, file systems, and programming-language libraries to access user-space and kernel-space I/O functions. Our resulting APIs fall into the three categories below.

Space creation/management

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

Open/close

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

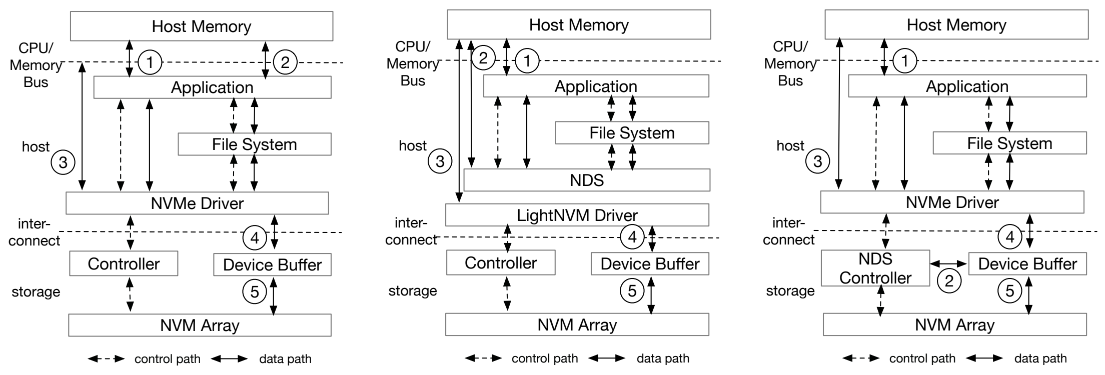
Read/write

API Functions in this category receive the following arguments:

1. a space identifier describing the source/target NDS space
2. coordinates and sub-dimensionalities describing the source/target data locations and shapes from the application's perspective
3. a memory buffer in the application space for the source data or target

If a function writes to NDS, NDS triggers the STL to allocate building blocks and transfer the source data from the specified coordinates in the application to the designated locations of building blocks in NDS. If a function reads from NDS, the STL translates the coordinates from the application's perspective to the corresponding locations of building blocks in the storage device. The read/write API functions can also work with multi-dimensional data-movement API functions (e.g., `cudaMemcpy2D` in CUDA) to move data more efficiently between high-dimensional accelerators and NDS.

Without NDS, the programmer needs to manually optimize applications for retrieving and storing data chunks using an optimal data layout that maximizes the storage



(a) (b) (c)

Figure 2.7: (a) The baseline, conventional SSD, (b) the software-only NDS implementation, and (c) the hardware-assisted, NVMe-based NDS implementation

bandwidth for the specific storage device. Most of time, such chunk size for storage does not match the chunk size for accelerators that maximizes the GPU utilization as challenges presented in Section 2.3.2. Therefore, the programmer needs to add code to construct application objects in chuck sizes that maximize the performance of compute kernels. And finally, the compute kernel that receives these objects can work on parts of the constructed data objects. In addition to the efforts of modifying applications, programmers need to go through exhaustive design space explorations in search for the optimal size for both the storage frontend and the compute kernel backend. In contrast, the applications using NDS allow the programmer to just focus on describing the desired objects from the compute kernel's perspective, since NDS can transparently handle the performance optimizations in the storage frontend. Although the programmer may still need to search for optimal parameters of the compute kernel, Programming with NDS should at least save half of the development time, since the programmer does not need to take care of the optimal object layouts and parameters for storage.

2.6.2 System implementations

We implemented 3 different versions of the storage systems to test the NDS concept. We developed a hardware-assisted NDS by extending the baseline SSD's controller and replacing the existing NVM management layer with the STL. In addition, we implemented a software-only NDS by using LightNVM [25], a storage protocol/interface that exposes software to the physical addresses of the underlying NVM device.

Figure 2.7 shows the system architectures, control paths, and data paths for the three implementations: The baseline SSD, the software-only NDS, and the hardware-assisted NDS. In applications involving conventional SSDs, like the configuration shown in Figure 2.7(a), the system stack incurs significant overheads of data accesses to the system main memory in order to exchange data objects for compute kernels (①), serialize/deserialize objects (②), and exchange the serialized objects with the device (③–⑤).

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

2.6.3 The NDS-compliant storage device

Our hardware-assisted, NDS-compliant storage device provides an interface supporting NDS's multi-dimensional address mode and implements the STL functions on a storage-device controller. This section describes the extended command set and the controller architecture in our prototype implementation.

PCIe/NVMe command extension

Our prototype NDS-compliant storage supports an extended NVMe command set while remaining compatible with existing NVMe commands [8]. An extended NVMe command uses a reserved bit in the first 64-bit command word in the NVMe standard to distinguish itself from conventional NVMe commands. Upon receiving a conventional NVMe command, NDS simply treats the command that requests data from a one-dimensional address space.

For read/write operations, the extended NVMe commands are almost identical to conventional read/write commands, except that the NDS version uses the second 64-bit command word. This command word points to a memory page that contains the coordinates and sub-dimensionality from the application's perspective.

The NDS/NVMe command extension has three commands for managing multi-dimensional address space: `open_space`, `close_space`, and `delete_space`. The `open_space` command can create a new space or change the dimensionality of an existing space depending on the flag set in the command header. The second `open_space` command word points to a 4 KB memory page that lists the dimensionality of the space, with up to 32 dimensions and 2^{64} elements in each dimension. The `open_space` command returns a 64-bit identifier and a *dynamic space ID*; the software system can use the space ID to distinguish between different views an application uses for the space. In contrast, the `close_space` command reclaims the dynamic space ID and disables the use of the previously defined space view. The `delete_space` command permanently deletes an address space by invalidating all space building blocks and removing the translation data structures for the space.

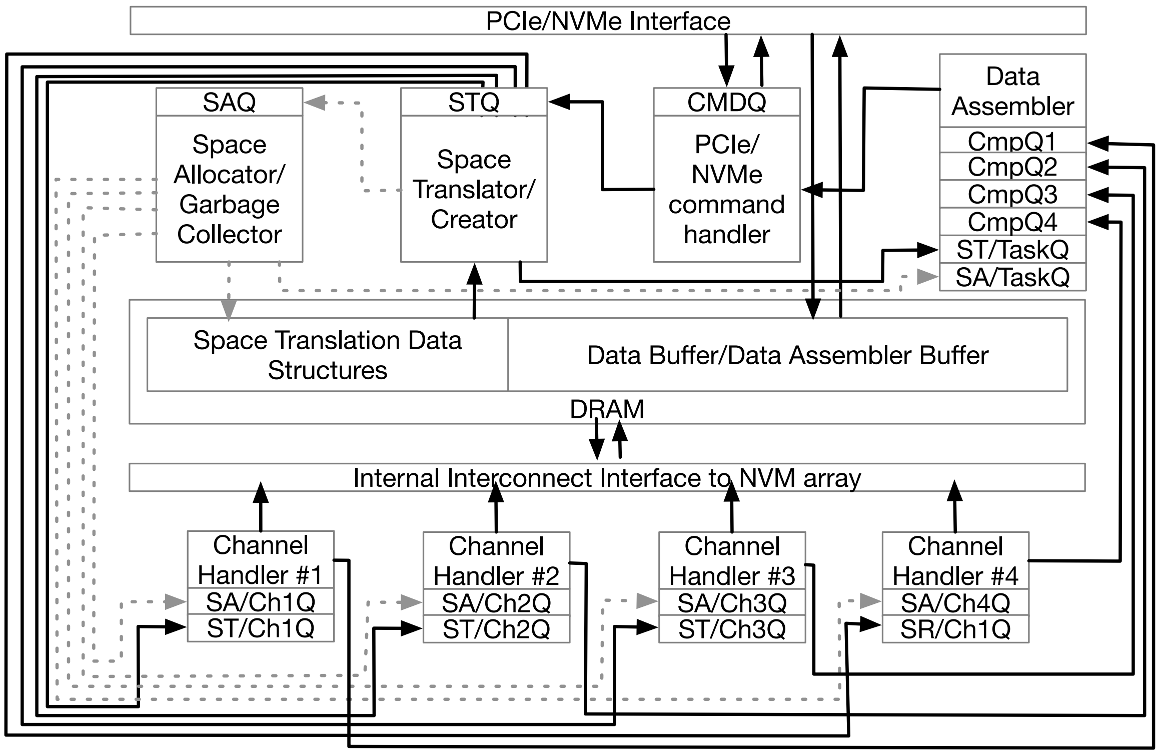


Figure 2.8: The NDS-compliant SSD controller

The controller architecture

As shown in Figure 2.8, our prototype NDS storage device has a controller that extends an existing NVMe/SSD controller to support the STL features. Similar to conventional NVMe controllers, the NDS controller exploits pipeline parallelism to maximize command-handling throughput by containing the following pipeline elements: (1) a PCIe/NVMe command handler, (2) a space translator/manager, (3) a space allocator with a garbage collector, (4) a data assembler, and (5) four channel handlers.

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

We built our prototype NDS controller by extending the firmware code in the baseline SSD controller using 8-core ARM A72 cores, and each pipeline element is statically mapped to one of eight cores. While the NVMe baseline controller also uses eight cores to implement essential functions, the baseline controller replaces the STL creator/translator with an address-lookup function and the data assembler with a command-control manager. Both the baseline and NDS controller have the same amount of channel handlers.

Supporting cryptography

Modern computer systems provide cryptography features in software modules and hardware accelerators to protect sensitive information. Modern datacenter-class SSD controllers [101] typically incorporate intellectual property cores to provide high-throughput data encryption/decryption. The most popular standard block-based advanced encryption mechanisms (AES) [104] work in the way of dividing data into fixed-sized sections and performing pseudorandom permutations within the same section of data. The resulting data size remains the same before and after encryption/decryption.

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

Supporting Data Compression

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

If the storage device transparently performs data compression functions, the hardware-assisted NDS can work with existing data compression features if (1) the data compression/decompression process occurs before the space allocation stage and (2) the data compression/decompression occurs in units of building blocks. As NDS's space allocation policy, explained in Section 2.5.2, randomly chooses access units from channels/banks, NDS can still ensure performance and even wearing, but simply uses fewer access units for each building block.

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

2.7 Experimental methodology

We evaluated the built NDS systems by modifying the I/O functions of a set of applications while keeping the same applications' multi-dimensional compute kernels. This section describes the system configurations and experimental setup we developed.

2.7.1 Experimental platform

We used an 8-core AMD RyZen 3700X processor with a clock rate of up to 4.4 GHz. We installed Ubuntu 16.04 (Linux kernel version 4.15) and implemented the NDS API, NDS subsystem (for the software-only version) and an extended NVMe driver (for the hardware-assisted version) to support the NDS model. We built a TLC-NAND flash-based SSD with the controller described in Section 2.6.

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

2.7.2 Benchmarks

As shown in Table 2.1 and Table reftable:benchmarksource, we used 10 applications to test our NDS system, with the applications falling into 6 categories: graph (traversal), linear algebra, physics simulation, data mining, image processing, and tensor-algebra operations. We selected the applications because each application (1) provides a highly efficient open-source implementation (or one that is easily optimized) for large datasets, (2) works on multi-dimensional datasets, and (3) provides or allows for the generation of datasets that exceed the capacity of GPU device memory. As the data volume of each workload is larger than the device-memory buffer on the GPU (where we execute compute kernels),

Wordload Name	**Category**	**Dimensionality**		**Data Size (Elements)**	**Kernel Sub-dimension Size (Elements)**
		Data	**Kernel**		
Breadth-First Search (BFS)	Graph Traversal	2D	1D	65536×65536	65536
Bellman-Ford (SSSP)	Graph Traversal		2D		65536×4096
Block-GEMM (GEMM)	Linear Algebra	2D	2D	65536×65536	8192×8192
Hotspot (Hotspot)	Physics Simulation	2D	2D	65536×65536	4096×4096
K-Means (KMeans)	Data Mining	2D	1D	65536×65536	65536
K-Nearest Neighbor (KNN)	Data Mining		1D		65536
PageRank (PageRank)	Graph	2D	2D	65536×65536	4096×65536
2D Convolution (Conv2D)	Image Processing	2D	2D	65536×65536	4096×4096
Tensor Times Vector (TTV)	Tensor Algebra	3D	2D/1D	2048×2048×2048	512×512
Tensor Contractions (TC)	Tensor Algebra		2D		512×512

Table 2.1: NDS Workloads, the dimensionality, size of their raw data and the dimensionality of their compute kernels.

Wordload Name	**Category**	**Source of Baseline**
Breadth-First Search (BFS)	Graph Traversal	Rodinia [34]
Bellman-Ford (SSSP)	Graph Traversal	Parallel Implementation of Bellman Ford Algorithm [146]
Block-GEMM (GEMM)	Linear Algebra	MSplitGEMM with cuBLAS using Tensor Cores [174, 106]
Hotspot (Hotspot)	Physics Simulation	Rodinia [34]
K-Means (KMeans)	Data Mining	Rodinia [34], Parallel K-Means Data Clustering [89, 54]
K-Nearest Neighbor (KNN)	Data Mining	Rodinia [34], knn-CUDA [52, 158]
PageRank (PageRank)	Graph	GraphBlast [165], GraphChi [80]
2D Convolution (Conv2D)	Image Processing	CUDA Separable Convolution [119]
Tensor Times Vector (TTV)	Tensor Algebra	NVIDIA [136, 33]
Tensor Contractions (TC)	Tensor Algebra	NVIDIA [136, 33]

Table 2.2: NDS Workloads and the sources of their baseline implementations

the compute kernels execute algorithms in a block fashion and must restructure input data into sub-blocks prior to data processing. Each application is pipelined so that its I/O and data restructuring (if required) overlap with the I/O and data restructuring of the compute kernels.

For the baseline implementation, we carefully partitioned the I/O size and selected the optimal size of the sub-matrix for compute kernels to minimize the end-to-end latency. For both NDS versions of these applications, we choose the parameters of each application's compute kernel by assuming the dataset already presented in the host main memory with the format that is ready for the compute kernel to consume, regardless of what the underlying NDS implementation is. As the datasets of these applications are larger than the host main memory capacity, these applications are very I/O intensive and the I/O part is always the longest pipeline stage in their baseline versions.

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

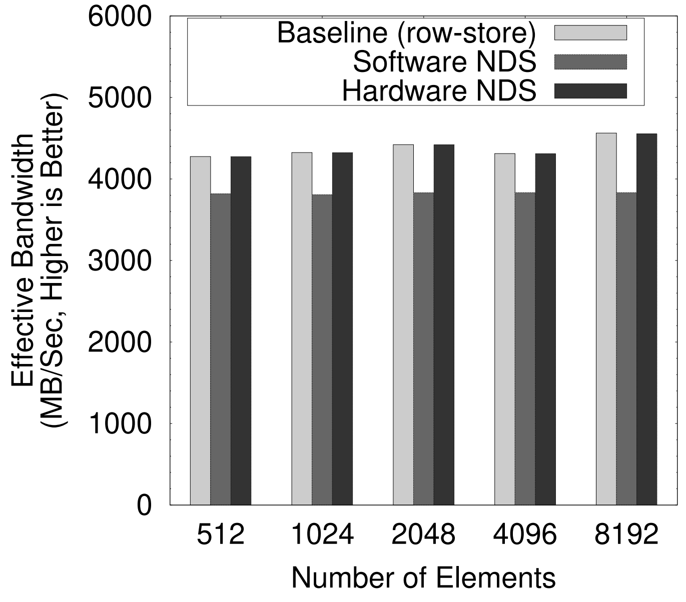


Figure 2.9: The performance of the baseline SSD, the software-only NDS, and the hardware-assisted NDS for fetching data in row direction with different dimensionalities

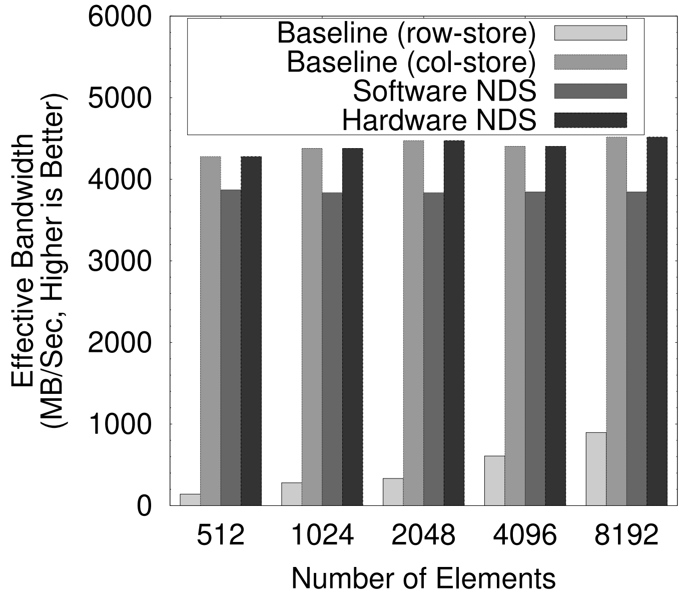


Figure 2.10: The performance of the baseline SSD, the software-only NDS, and the hardware-assisted NDS for fetching data in column direction with different dimensionalities

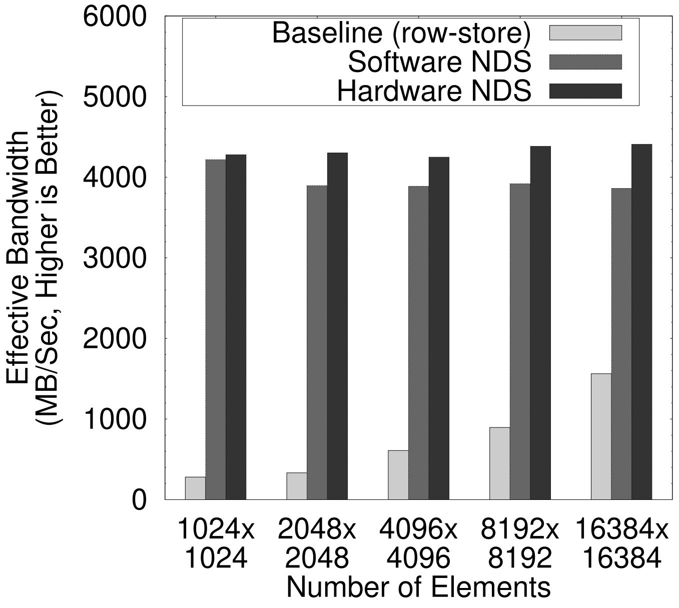


Figure 2.11: The performance of the baseline SSD, the software-only NDS, and the hardware-assisted NDS for fetching submatrices with different dimensionalities

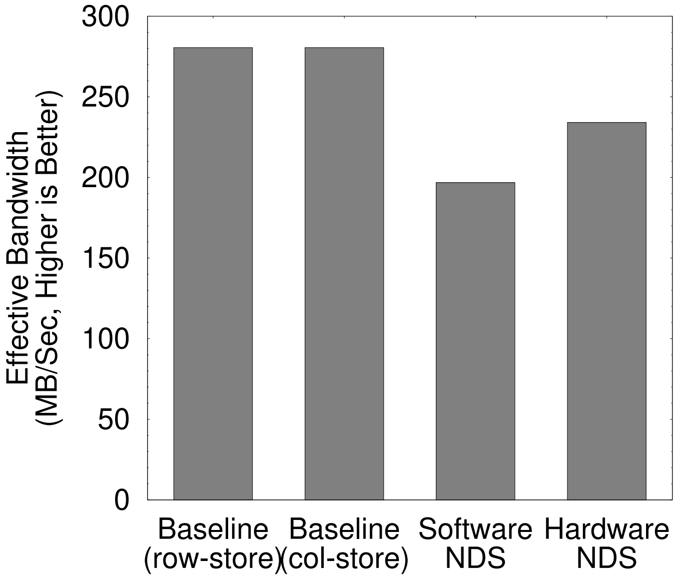


Figure 2.12: The performance of the baseline SSD, the software-only NDS, and the hardware-assisted NDS for writing data

2.8 Results

This section summarizes our evaluation of NDS, and we observed the hardware-assisted NDS implementation achieves 5.73× speedup compared to baseline SSD.

2.8.1 Microbenchmarks

We created a set of microbenchmarks with raw data comprising a 32,768×32,768 2-D matrix and a storage device with 32 channels and 4 KB pages to understand the pure I/O performance of NDS. The prototype NDS system selected 256×256 double floating-point elements as the building-block size for the storage. Figure 2.9 – 2.12 shows the effective bandwidth measured from the application side in fetching and structuring the data.

In Figure 2.9, the application fetches data in the row-direction, and the dimensions of the requested data range from 512×32,768 to 4096×32,768 until the application finishes reading the whole matrix. As the data are already in row order, the microbenchmark with the baseline SSD achieves an effective bandwidth of around 4.3 GB/sec. The hardware-assisted NDS achieves a performance that is almost identical to that of the baseline without optimizing building-block dimensions for the access pattern. These results indicate that the determined building-block structure effectively permits the hardware NDS to fully utilize a device's internal bandwidth and thereby cover the overhead of constructing rows from those building blocks. In contrast, the software-only NDS creates significant overhead by copying 256 elements (2 KB; smaller than a host DRAM page size) for constructing a row from related building blocks. As a result, the software NDS's effective bandwidth was observed to be only 3.8 GB/sec.

In Figure 2.10, the application performs column accesses for sub-matrices with dimensions ranging from 32,768×512 to 32,768×4096. For the baseline/row-store condition, if the dataset is not presented optimally in the baseline SSD, then the effective bandwidth of fetching a column is at most 600 MB/sec, with the largest granularity appearing when the system cache is allowed to serve later requests without visiting the SSD. On the other hand, NDS still works efficiently in constructing a column for each access, and the resulting performance (4.3 GB/sec) is comparable to storing column-ordered data in the baseline SSD (the baseline/column-store condition).

In Figure 2.11, the application fetches submatrices of various shapes. Because NDS's building blocks match the 2-D space the application requests, NDS significantly outperforms the baseline SSD, regardless of NDS implementation type. For an application whose data is stored in the baseline SSD, the application must generate many small I/O requests, with each request fetching a row from the baseline SSD; when accessing content for a submatrix, this process underutilizes both the interconnect and the device's internal bandwidth ([**P1**], [**P2**] and [**P3**] mentioned in Section 2.3).

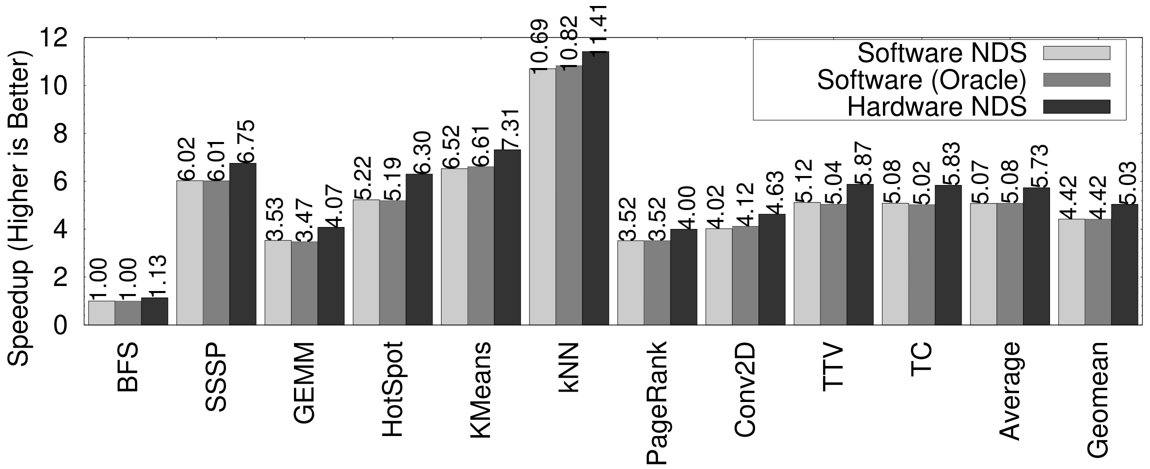
Figure 2.12 shows the performance of writing the microbenchmark 32,768×32,768 2-D data matrix into the baseline SSD and NDS. For the baseline SSD, data is arranged in both row-store and column-store formats before writing. For NDS, the dataset is arranged in row-oriented 2-D dense-matrix format before writing to NDS (using the NDS API). For these experiments, we disable asynchronous writes, so latency is measured until the end of the whole programming process. The baseline SSD has an effective write bandwidth of 281 MB/sec for both row-store and column-store formats.

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

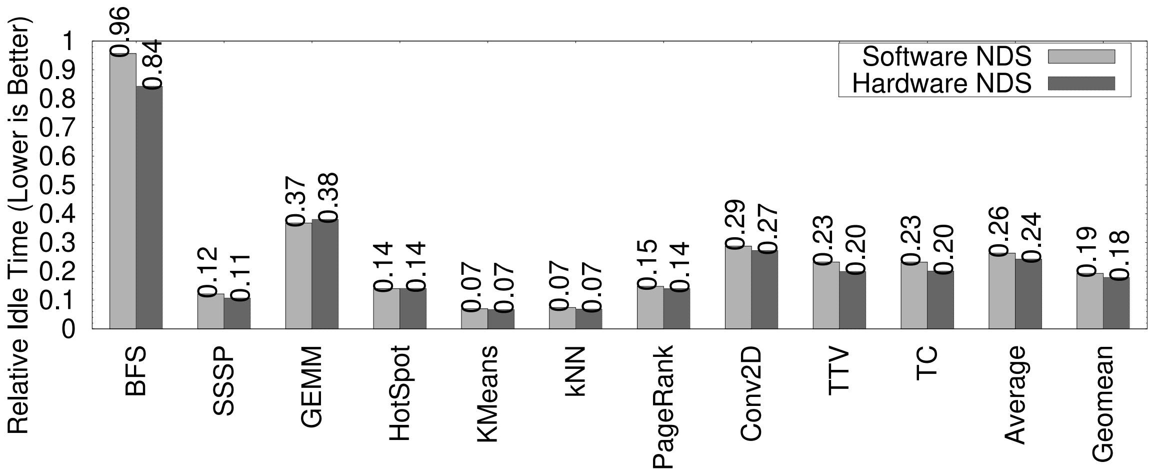
For hardware NDS, the SSD requests host main memory content in 4 KB pages and breaks them up later, allowing the SSD to better utilize the host main memory bus and system interconnect. However, the increased overhead in the storage controller along with the controller's lower performance (compared to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

2.8.2 End-to-end application latency

Figure 2.13(a) shows the speedup of end-to-end latency of running applications. The software-only implementation can achieve a speedup of 5.07×, which demonstrates the benefits of using NDS building blocks to store data and dynamically rebuild objects. By using building blocks, the software-only implementation allows the NDS software to speed up the process of building multi-dimensional objects by 1.52× on average. With building blocks reducing the chances of fetching temporally unnecessary data from the



(a)



(b)

Figure 2.13: (a) The speedup of end-to-end latency and (b) the reduction of idle time in compute kernels of running applications using NDS

SSD, software NDS also reduces 74% of average idle time before each pipelined compute kernel as Figure 2.13(b) shows.

An alternative to software NDS is using a software library and carefully layout data on the storage device. To investigate the maximum potential of all software-based approaches, we created an *oracle* configuration where we exhaustively search for the best storage data layout that incurs zero overhead on the host and minimum end-to-end latency when executing these workloads. In this configuration, we have to store two copies of data in different data layouts for workloads sharing the same datasets (i.e., BFS and SSSP, KMeans and KNN, TTV and TC). Figure 2.13(a) shows that even assuming these software libraries have zero overhead, the performance gain is just about the same as the software NDS.

Compared with the software-only solution, the hardware NDS can further accelerate applications by 5.73×. As Section 2.6.2 explains, the hardware-assisted implementation removes both computation overhead and traffic on the host processor/memory needed to rebuild data objects from building blocks. Hardware NDS completely skips the process of assembling multi-dimensional objects on the host computer. The hardware NDS also allows the STL to access a storage device's internals to fully utilize device's parallelism. Even though the NDS controller is less powerful than the host processor, the hardware NDS still outperforms the software-only solution by 1.13× and reduces the idle time before compute kernels by 76%.

Among these test applications, we found that BFS receives almost no benefit from the software-only NDS; although the original dataset was created and stored in 2-D building blocks, the compute kernel relies on sequential access along each row in the 2-D graph representation, and works efficiently with the stored data in a row-ordered format, which matches the storage format of the baseline SSD version.

The software-only NDS shows that building blocks work well with mismatched access patterns (discussed in Section 2.8.1); accessing matrices in row-order format produces similar results. In contrast, the hardware-only NDS outperforms the baseline version because the hardware-only version (1) passes objects to the application that exactly match compute-kernel demands and (2) accesses building blocks with more bandwidth than any host-side software can provide.

Though hardware NDS's 1.13× speedup over software NDS seems limited, we argue the presence of hardware NDS is meaningful for the following reasons:

(1) Software NDS relies on lightNVM [25] that is currently still not widely adopted by manufacturers.

(2) Software NDS increases the CPU workload and makes it less preferable to heavily loaded servers.

(3) As software NDS always requires the host-side module to perform space translation and address lookup, software NDS makes the use of modern system interconnects' peer-to-peer data exchange between storage devices and hardware accelerators [110, 6, 176, 155, 90] inefficient.

(4) Hardware NDS allows the STL within the storage device to work closer with the NVM array and use the rich internal bandwidth to handle NDS tasks. Our baseline SSD has an internal-to-external bandwidth ratio of 8-to-5. With faster NVM technologies that raise the internal-to-external bandwidth ratio, the advantage of hardware NDS will become more significant.

2.8.3 Overhead of NDS

As the STL requires more complex data structures and arithmetic operations in handling requests, NDS increases the latency and creates space overhead in storage devices, but to a very limited degree. We evaluated the worst-case scenario where a request only asks for a page of data from the baseline SSD and NDS. All requests to the baseline and NDS are carefully designed to avoid any transformation to help to identify the increases in data access latency from B-tree traversal. The result shows 41μs additional latency in software NDS and 17μs in hardware NDS; both are shorter than or in the same order as the average latency of accessing a modern NAND flash page (typically 30μs–100μs) [56, 102]. However, as a leaf node in NDS's B-tree structure can point to up to 512 flash pages, if the request asks for a larger block of data, NDS simply requires one B-tree traversal for all accesses and can easily amortize the additional latency. In the worst-case scenario where every page is in use, the whole STL lookup data structure occupies 0.1% of the storage space.

2.9 Other Related Work

In addition to the related work described in Section 2.3.3, several other lines of NDS-relevant research deserve mention.

Tensor algebra libraries, algorithms, compilers, and accelerators

For decades, tensor algebra has been explored through algorithms [30, 15, 71, 166, 20, 98], libraries [91, 21, 156, 141], code generators [87, 142, 35, 75], and accelerators [183, 59, 84, 123, 53, 3, 144, 143, 55, 178, 115, 182]. Most prior work has focused on improving the efficiency of tensor computations. In contrast, NDS offers a streamlined compute-kernel front-end to address the overheads caused by data transfer/restructuring.

Other in-storage processing approaches

The hardware NDS is similar to in-storage processing in that NDS extends the storage controller to dynamically assemble data from building blocks. Aside from the projects mentioned in Section 2.3.3 that use ISP/NDP to directly present data as applications require, existing general-purpose ISP/NDP platforms can enable object deserialization functions [133, 4, 44, 57, 78, 177, 175]. Specialized ISP/NDP systems can also run compute kernels on storage controllers, thereby accessing the rich internal device bandwidth to accelerate file system operations [32] and data analytics [68, 163, 67]. As noted previously, however, the data layout may not align with in-storage compute-kernel access patterns. In this case, in-storage applications may perform inefficiently even though the internal bandwidth is accessible to code/accelerators in the storage device.

Sparse formats

The Tensor Algebra Compiler (TACO) [76] can generate efficient code based on iteration graphs, merge lattices, and a tensor storage tree for both sparse and dense matrices. One work [37] demonstrates that a sparse tensor-algebra compiler should be agnostic to data layouts [16, 27, 73, 131, 64]. Among data representations, the compressed sparse-block (CSB) format [28] suggests that building blocks may be equally effective for both row-wise and column-wise sparse-matrix processing. NDS focuses more on dense formats because the data-processing throughput of compute kernels on dense datasets is significantly higher and NDS's storage demands are greater. Nonetheless, NDS can store sparse content efficiently through a checking/optimization process that is similar to page-zero optimization in VAX/VMS [85].

Smart main memory controllers

Without a storage system like NDS to present data in a way that aligns with a compute kernel's perspective, the problem `[P2]` will significantly bottleneck application performance for various access patterns. Both Impulse and Gather-Scatter DRAM (GS-DRAM) proposed smart memory controllers or adding additional circuits that add another layer of main memory address translation and dynamically create condensed application objects without redundant elements/values going through the CPU-main memory bus [31, 134]. However, both Impulse and GS-DRAM still lead to `[P3]` as the internal page data layout is still either row or column oriented. RC-NVM [88] further confirms that a dual-addressing mode is unrealistic with DRAM architectures. In contrast, NDS can release the burden of Impulse or GS-DRAM and ultimately address both `[P2]` and `[P3]` without the presence of Impulse or GS-DRAM if the raw data comes from the storage subsystem. NDS also needs zero modifications in NVM chips as RC-NVM.

2.10 Conclusion

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

Chapter 3

ActivePy: Intelligent Programming Interface for In-Storage Processing

3.1 Abstract

In-storage processing (ISP) is the most commercialized implementation of the near-data processing (NDP) model that executes tasks near their data locations on a computational storage device, thereby mitigating the performance bottleneck of exchanging large volumes of data among system components. However, programming NDP platforms is complicated as it requires programmers to work closely with the underlying hardware, and even highly-optimized code can easily lead to suboptimal performance.

This work introduces ActivePy. ActivePy is an intelligent programming framework that can automatically identify the code region and dynamically generate high-performance code for computational storage device (CSD) to balance system-wide trade-offs and maxi-

mizes the benefits of ISP. Unlike conventional ISP frameworks, ActivePy requires no change to the programmer language front-end and no programmer's intervention that makes the programmer completely agnostic to ISP hardware components.

We implemented ActivePy to support Python and evaluated the proposed design using a high-end heterogeneous computer with a CSD. Our results show that ActivePy can use a CSD as efficiently as a conventional C-based framework, achieving 1.33× on the prototype platform. Furthermore, ActivePy has the ability to migrate the offloaded tasks back to the host when it detects slowdowns on CSD, helping ISP programs to avoid severe performance degradation.

3.2 Introduction

The rapid growth of application demands has pushed the conventional von Neumann architecture incorporates hardware accelerators offering orders-of-magnitude performance gains to speed up compute kernels in applications. However, the relatively slow improvement in non-volatile memory technologies has made data supply an emerging overhead. Furthermore, the limited interconnects bandwidth in a heterogeneous computer makes the problem more significant. On the other hand, the sizes of datasets for applications have grown rapidly, increasing the demand for data storage and the traffics among system interconnects.

Due to the mismatching of evolution among hardware components and the rapid growth of application demands, the popularity of in-storage processing (ISP) or computational storage solutions is increasing. ISP addresses both problems by leveraging or extending existing controllers on the data storage. By pre-processing storage data or offloading other host computing resources' workloads on more intelligent controllers, ISP reduces the data volume going through the system interconnect or allows the system to use host computing resources more efficiently.

Unfortunately, programming existing ISP frameworks is challenging for the following reasons. First, identifying the most advantageous use of the ISP model or CSD is not straightforward. It heavily relies on programmers' knowledge of hardware and applications [17]. Second, composing and offloading a function on a CSD heavily depends on firmware programming [62, 96, 169], customized libraries [133, 155, 78, 62], specialized programming models [57, 129], low-level commands attached to storage objects [5], and hardware-description languages [32, 67, 177, 44, 68]. Finally, even though programmers have optimized the ISP program, if computational resources or data sources change during the runtime, the resulting program still leads to suboptimal performance, as these frameworks have zero or minimal capability in dynamically adjusting and migrating offloaded workloads [78, 17].

This work proposes a principled design of ISP programming frameworks to overcome the aforementioned programming challenges. We argue that an ideal ISP programming framework should fulfill the following characteristics. First, the framework should decide on the most valuable CSD functions to release the burden on programmers. Second, the interface of the proposed framework should be an integral part of the host programming interface and not require the programmer to compose the device function explicitly. Finally, the framework should allow flexibility in adjusting host/CSD workloads to adapt to the change in systems. In this way, the proposed programming framework will have flexibility and transparency for the underlying CSD devices.

In summary, this work makes the following contributions:

(1) It is the first work that explores the use of an interpreted language for programming CSDs in heterogeneous computing platforms to demonstrate the deficiency of conventional compiled programming languages on the same platform.

(2) It proposes a set of mechanisms to analyze and automatically generate programs on ISP platforms, making ActivePy the first ISP programming platform that does not rely on any programmer's annotation, pragma, or hint.

(3) It identifies and optimizes the performance bottlenecks of an interpreted-language runtime in heterogeneous computing.

(4) It evaluates ActivePy by building the complete system and experimenting with a set of data-intensive applications.

Our real system evaluation shows that ActivePy can allow ISP programs written in Python without any programmers' hints and CSD code to achieve almost the same performance as the equivalent, fully-optimized ISP programs written in C. The average speedup of the end-to-end latency from these applications is 1.33×, compared with equivalent baseline implementations written in C. ActivePy also makes the program less fragile to system dynamics than conventional ISP programming frameworks.

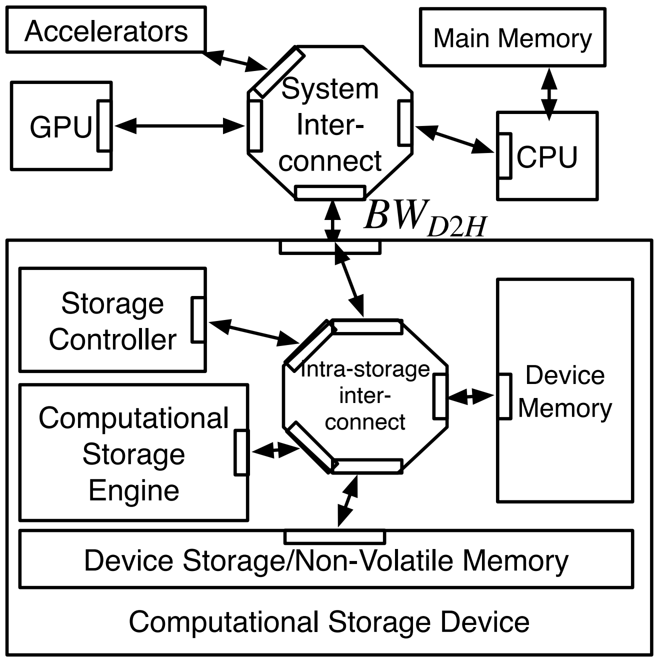

Accelerators
Main Memory
GPU
System Inter-connect
CPU
BW_{D2H}
Storage Controller
Intra-storage inter-connect
Device Memory
Computational Storage Engine
Device Storage/Non-Volatile Memory
Computational Storage Device

Figure 3.1: The architecture of a computational storage device in a heterogeneous computer.

3.3 Background and Motivation

With commercialized products [44, 45, 139] and working groups on standards [140], the ISP model that offloads computation to computational storage drives (CSDs) starts to gain ground in data-intensive computing platforms. This section describes the architecture of modern CSDs and the challenges of exploiting performance using existing system frameworks.

3.3.1 Computational storage devices (CSDs)

Figure 3.1 shows the high-level architecture of a CSD in a modern heterogeneous computer. In modern heterogeneous computers, any peripheral device attached to the host computer must communicate with other devices through a host system interconnect (e.g., PCIe), except the communication between the CPU and main memory. When applications need to move large data volumes around different system components, the system interconnect can become a bottleneck, since, for example, in modern computers using the PCIe 3.0 standard, the storage device can only share 4 GB/sec data bandwidth, but data supply/consumption rates provided by peripheral devices are usually higher than the interconnect bandwidth.

In addition that a storage device typically contains the storage controller, the host interconnect interface, the device memory (volatile), and the device storage (non-volatile), CSDs have a computational storage engine (CSE) that can execute offloaded host functions near the data locations. The CSE communicates with the device memory/storage with relatively richer bandwidth coming from the exclusive intra-device interconnect, typically 8 – 16 GB/sec [118]. In commercialized products, the CSE implementations can (1) leverage existing storage controllers [45], (2) add general-purpose processor cores, or (3) add an FPGA [44]. The processing power of the CSE allows the ISP model to improve program efficiency by (1) reducing the volume of data going through the relatively slow, narrow system interconnect, and (2) permitting tasks running on processors near data locations to receive data with richer internal bandwidth than the available external bandwidth going to the host [45].

$$S = (\frac{DS_{raw}}{BW_{D2H}} + CT_{host}) - (CT_{device} + \frac{DS_{processed}}{BW_{D2H}}) > 0 \tag{3.1}$$

We generalized Equation 3.1 that quantifies the net profit (S) of performing tasks (code regions) using a CSD [26, 150], instead of using the processors/accelerators on the host computer:

In this equation, DS_{raw} represents the raw input data size associated with executing all code regions on the host, CT_{host} represents the latency of executing the code using the host processor with all input data present in the host main memory, CT_{device} represents the latency of executing the equivalent code region on the CSD, $DS_{processed}$ represents the size of the intermediate data the device code produces, and BW_{D2H} represents the interconnect bandwidth between the device and the host. An ISP-assisted program is considered efficient if the program can make $S > 0$ in Equation 3.1.

3.3.2 Challenges

Making $S > 0$ in Equation 3.1 is challenging in existing ISP programming frameworks for the following reasons:

Limited CSE performance.

Unlike GPU or ML/AI accelerators that deliver orders of magnitude speedup over the host CPU, the CSE in a modern CSD has limited processing power, which makes computations on CSD typically slower than the host CPU (without any help from hardware accelerators added to CSD). Previous studies [78, 155] have shown that the computation on the CSE is slower than the host CPU, and a majority of performance gain comes from reduced data volume. Therefore, a programmer must comprehensively understand application behavior and hardware characteristics to allocate application tasks to the most appropriate computational resources.

Ad hoc, difficult-to-use and error-prone programming frameworks

Programming in CSD is further complicated by the programming models. Existing platforms rely on (1) firmware programming in C [62, 96, 169], (2) limited function through customized libraries [133, 155, 78, 62], (3) specialized programming models [57, 129], (4) low-level commands attached to storage objects [5], and (5) hardware-description languages (e.g., Verilog) [32, 67, 177, 44, 68]. As a result, the development of CSD functions is separated from the application, and the conventional CSD programming frameworks require significant changes in the original application. Also, the interface between different languages potentially incurs additional memory and data exchange overhead. On the other hand, the absence of hardware virtualization in these programming platforms makes CSD programming error-prone and non-portable.

Lack of flexibility during execution

Even though the programmer has carefully and exhaustively profiled and optimized the ISP application, there is still no guarantee of performance gain from using CSDs. The parameters in Equation 3.1 can change during the runtime due to (1) resource contention coming from other applications, (2) resource contention coming from the storage management workloads (e.g., garbage collection), and (3) the change of input datasets itself. With the programmer-directed design of CSD functions in modern ISP programming frameworks, there is no flexibility for existing ISP platforms to be adaptive to these system dynamics. The following two examples illustrate why ISP platforms should feature flexibility.

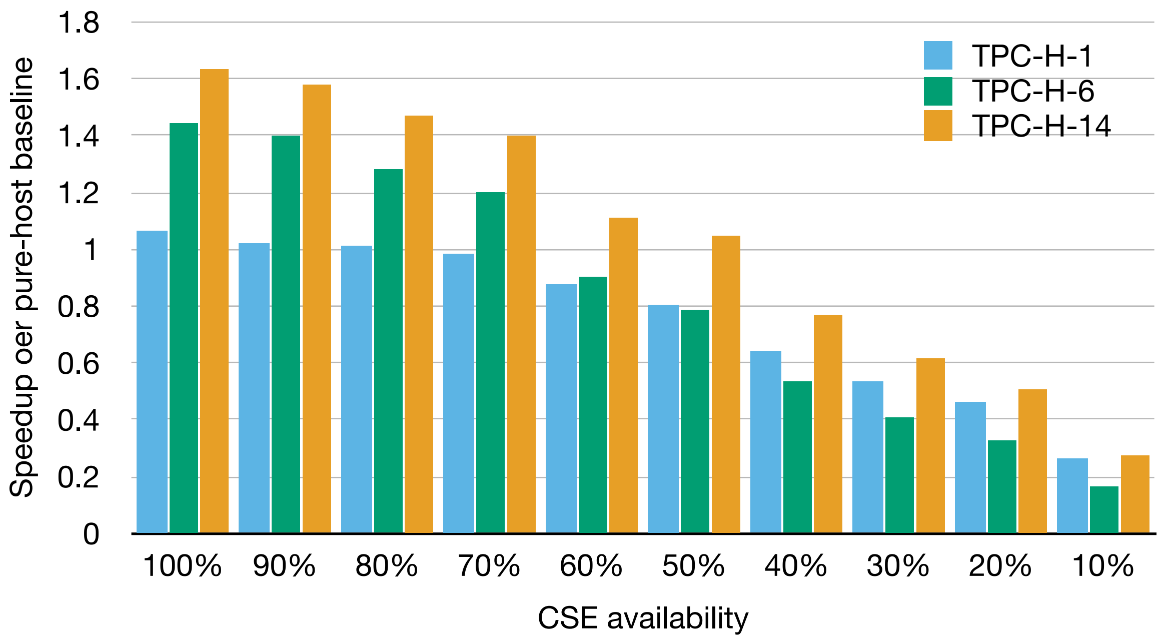


Figure 3.2: The performance of the baseline task assignment compared with the “oracle” (optimized for the case where CSE are free) under different available hardware resources.

In Figure 3.2, we leverage three TPC-H workloads used for evaluating a representative CSD, Summarizer [78]. We write all those workloads in C, implement the same code optimization/distribution, and offload CSD code on our baseline CSD described in Section 3.6.1 later. We change the available CSE resource for code regions running on the CSD to emulate the changes in computing resources. The code optimization from Summarizer is based on the case when the CSE is 100% available to the application. The x-axis in Figure 3.2 represents the portion of CSE available for the program, and the y-axis represents the speedup of the end-to-end latency on the optimized workload under different CSE availabilities. When the CSE fully dedicated its resources to tasks from the CSD-assisted program (i.e., 100% in Figure 3.2), these workloads are 1.25× faster than their baseline (i.e., the same workload but not using CSDs at all). However, the same optimiza-

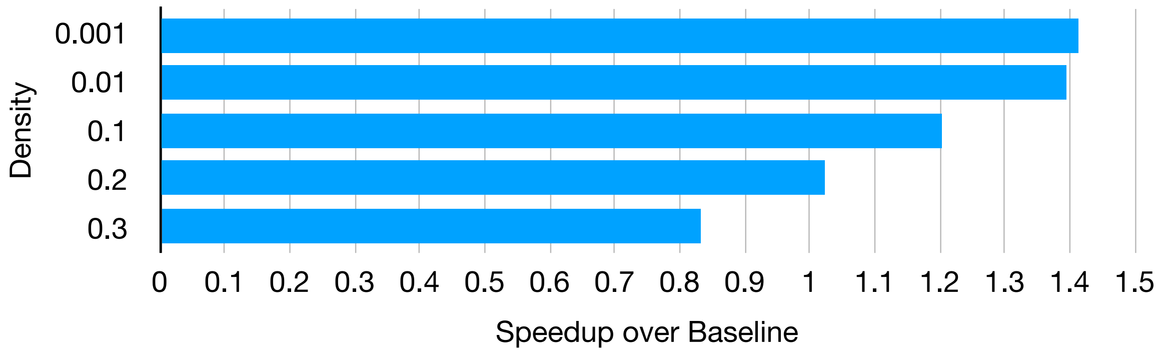


Figure 3.3: The speedup of an optimized ISP program under datasets with various input densities.

tions/distributions to these workloads suffer from performance loss when the CSD has only 60% or less available computation time.

Figure 3.3 shows that changes in datasets can also affect the effectiveness of ISP. In this set of experiments, we used the CSD to generate compressed sparse row (CSR) formatted sparse matrices from the raw data. The baseline is the host program's implementation converting raw data into the CSR format without leveraging the CSD. The result shows that as the matrix becomes denser gradually, the performance gain of using ISP shrinks since the ISP model cannot get benefits from reducing the data volume going through the system interconnect, and the computation on CSE is usually slower than the one on the host CPU. In Figure 3.3, the ISP program can slow down the CSR conversion if the density of the matrix surpasses 20%.

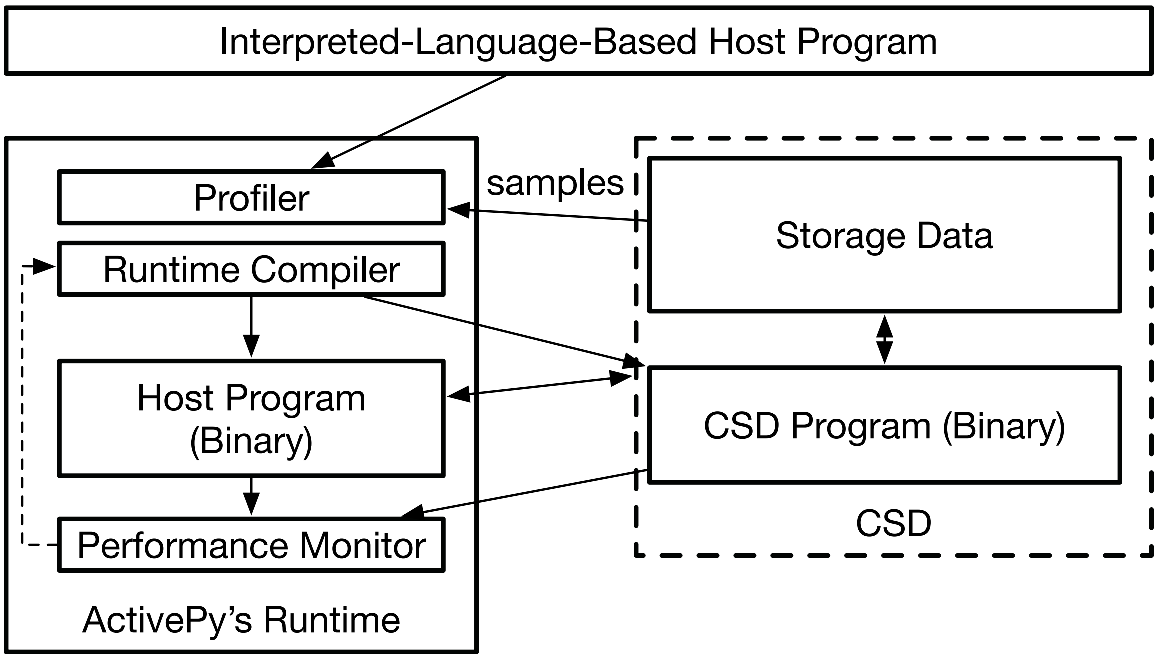


Figure 3.4: The architecture of a computational storage device in a heterogeneous computer.

3.4 ActivePy Overview and Design Principles

We designed ActivePy that addresses the challenges mentioned in Section 3.3.2, so that ActivePy makes programmers easier to exploit the performance of emerging CSDs in heterogeneous computers, compared with using conventional ISP programming frameworks. ActivePy contains three main system components: (1) a front-end programming interface using an interpreted language, (2) a host runtime system, ActivePy's runtime system, on the host computer to execute the main program, and (3) a CSD runtime system with a set of device functions to manage and execute tasks.

Figure 3.4 illustrates the workflow of ActivePy. ActivePy takes a typical interpreted-language-based program (i.e., a Python program in the current implementation) that does not contain any programmer's annotation or hint about the ISP model as the input. The host runtime system collaborates with the CSD runtime to form sample inputs from the raw data files of the running program. Next, the host runtime uses the sample inputs to perform a sampling phase and collect statistics. The host runtime system then estimates the latency of each line of code when executing the code on the host processor and the CSD to develop an initial task-assignment plan. In ActivePy, we define a *task* as a program's dynamic instance of a code region.

Next, ActivePy's host runtime system automatically separates the CSD tasks and rewrites parts of the program code to enable the interaction between the CSD program and the host program. Finally, ActivePy's host runtime system will generate binary running on both the host and the CSD and start executing the program with the raw input.

When the ActivePy program runs, the host runtime system continues monitoring task performance on each computing unit. If a performance degradation occurs during the runtime, ActivePy's runtime system can reassign the tasks between the host and the CSD, repartition and regenerate the code, and migrate tasks between the host processor and CSDs.

The proposed design applies the following principles to address the challenges of modern ISP platforms.

Making programmers agnostic to ISP/CSD to address programmability.

ActivePy distinguishes itself from other conventional ISP platforms as it makes no change to existing programming languages, meaning that programmers do not need to define the CSD function or write any CSD application themselves. Therefore, ActivePy can successfully address the programmability issue that conventional frameworks create.

Performance optimizations through runtime systems for performance, programmability, and flexibility.

Instead of using compilers to generate static ISP programs, ActivePy relies on the runtime system to automatically compose an ISP-accelerated program. Without the runtime system that dynamically generates and optimizes code, the binary generated by a static compiler has no way to evolve itself to adapt to any system dynamics change.

An interpreted-language-based programming interface to address programmability and flexibility.

ActivePy uses interpreted language for the following reasons. First, an interpreted language possesses a line-by-line execution nature and typically relies on the runtime system to dynamically generate machine code. Implementing with an interpreted language makes ActivePy easily create or change tasks for CSDs, which conventional ISP frameworks fail to do. Second, recent successes with compilers for interpreted languages [14, 49, 81, 130, 72], their abilities in emitting intermediate representations [82] for further optimizations, and supporting back-end C/C++ libraries allow ActivePy to adopt an interpreted language front-end but still deliver performance comparable to that of conventional compiled languages. Finally, interpreted languages fulfill the popular trend of improving programmer accessibility [138, 23].

3.5 ActivePy Implementation

This section describes each key component in ActivePy in detail.

3.5.1 Sampling Phase

To find out the sweet spot of slicing code from the original program that maximizes the benefits of using available CSDs, ActivePy needs to determine the parameters in Equation 3.1 for each line of code. ActivePy achieves this goal by running a sampling phase that collects necessary statistics to guide the values when evaluating Equation 3.1.

The design of ActivePy's sampling phase leverages two general observations. (1) A small subset of input data can capture properties of the original input [83]. (2) the latency and resulting data size are relational to the input data size. Therefore, the sampling phase starts by heuristically selecting data from raw inputs and creating sample inputs of different sizes. The current implementation generates inputs using four scaling factors (F): (1) tiny, $2^{-10}\times$, (2) small, $2^{-9}\times$, (3) medium, $2^{-8}\times$, and (4) large, $2^{-7}\times$.

ActivePy will start running the program using these sample inputs and record the execution time, the input data size, and the output data size of each line of code during the execution. If the code contains access to stored data, ActivePy will separate the data access time from the code execution time. This is because the data access time is typically linear to the data size but not necessarily true of the computation time. Although the sample inputs do not guarantee the program to generate meaningful computation results for the workload, however, since the sampling phase is simply collecting information to estimate the benefit of ISP, this method fulfills ActivePy's purpose.

During sample runs, ActivePy generates a set of estimated performance metrics for each line of code (L_i), including (1) $CT_{i,host}$, the estimated computation time on the host, (2) $CT_{i,device}$, the estimated computation time on the CSD, (3) $DS_{i,in}$, the estimated volume of data inputs, and (4) $DS_{i,out}$, the estimated volume of data outputs. Once four sample runs are complete, ActivePy generates a linear model for extrapolation to predict the execution time and data-size changes by selecting the closest fit from one of five curves—$O(1)$, $O(n)$, $O(n\ log\ n)$, $O\left(n^2\right)$, and $O\left(n^3\right)$. Next, ActivePy extrapolates the execution time and data-size changes with the raw data size. ActivePy then estimates the expected execution time on the target CSD by multiplying the predicted computation time on the host with a constant factor (C). ActivePy calculates C by either (1) querying the CSD's performance counters (e.g., retired instructions per cycle), or (2) running a small sample program on both a CSD and the host computer if performance counters are not available.

The accuracy of the sampling phase in ActivePy aims at a "good enough" rather than a highly accurate one for the following reasons: First, ActivePy uses the estimation for initial task allocation and can gradually adjust and optimize the decision during the runtime later. Second, as system dynamics can change anytime, high accuracy in estimation is not very useful in most scenarios. Finally, highly accurate estimation mechanisms usually introduce extra overheads, and such highly accurate mechanisms may not be able to outweigh the gain from a "good enough" but lightweight one.

3.5.2 Identifying CSD code regions

Algorithm 1 summarizes the one ActivePy uses to identify the CSD functions. The design of the algorithm uses one line of Python code as the basic unit of forming code regions, but not finer-grained at the translated code level because: (1) due to the nature of line-by-line interpretation of the interpreted-based language, each line of Python code is a single-entry-single-exit code region that facilitates code generation and optimization. And more importantly, (2) as CSDs must communicate with the rest of the system through bandwidth-constrained, relatively-long-latency system interconnect, the BW_{D2H} factor in Equation 3.1 is usually small and makes the data movement overhead significant. Therefore, ISP models cannot take advantage of fine-grained task allocations that arbitrarily distribute tasks among different components, since the cost of sending data back and forth through the interconnect is expensive.

Algorithm 1 initializes the set of code (i.e., CSD code) to execute on the CSD, P_{csd}, as an empty set and uses the total execution time for all code on the host, T_{host}, as the projected execution time on the CSD, T_{csd}. After having the projected execution time and estimated data-size changes, ActivePy firstly examines every line to determine whether adding the line into the P_{csd} can reduce execution time T_{csd}, and secondly records the assignment that yields the shortest execution time until the algorithm went through every line of code.

3.5.3 Code patching

After ActivePy identifies P_{csd}, the computation and data accesses in the CSD function, ActivePy will patch the program code for the following purposes: (1) memory allocations, (2) CSD function invocations, and (3) elimination of redundant memory operations.

Memory allocations

ActivePy adopts a shared memory address space between the host program and the CSD program. This design reduces the software overhead and redundant memory copies by leveraging existing architectural supports. For CSDs attached to the host computer using PCIe interconnect, the CSD can expose its memory available for ActivePy's CSD functions by declaring them in the PCIe BARs (base address registers). ActivePy's kernel memory module can work with the OS's virtual memory subsystem to map these CSD locations into any ActivePy program's virtual memory address. The host program can later directly access these locations using load/store instructions without the demand for additional memory buffers and calls to I/O library functions. If the CSD attaches to the host computer through storage protocols over the network (e.g., NVMe over fabric (NVMeoF)), the CSD can leverage the RDMA hardware infrastructure NVMe already uses to support the storage function to map the device's internal memory into the host program's virtual address space. Similarly, the host program can use load/store instructions to directly access the CSD's device memory without additional memory and library overhead.

A data structure in ActivePy may belong to one of the following. (1) Local. If only the CSD code or the host code accesses the data structure, ActivePy will allocate the data structure where the task performs. In this case, ActivePy treats the data structure as local to the code's computing resource. (2) Shared-host. If a CSD code region and a host region share the data structure, ActivePy will place the data structure on the host side. ActivePy prefers the data allocation on the host side because the data processing throughputs on current host side computing resources (e.g., CPUs, GPUs) are still larger than those in CSDs. In addition, the bandwidth between the host side computing resources and host local memory locations is still significantly larger than that between the host and the storage. With the support from RDMA or PCIe P2P, for example, allocating data on the host side allows the host side program to resume earlier and more efficient memory accesses when updates from the computing resources perform the compute-intensive kernels, more likely on the host side. (3) Shared-CSD. It is similar to the shared memory allocated on the host side. However, if the target side runs out of physical memory locations, ActivePy will use other available locations.

Although the data access itself uses conventional load/store instructions, ActivePy needs to initialize the mapping of data structures in one of the three different modes by using ActivePy's host or CSD library functions. Then, ActivePy automatically patches these function calls to ensure the correct mapping before the host or CSD code can use them.

CSD function calls

As modern CSDs use fast, non-volatile memory, ActivePy's approach to making CSD function calls mimics the mechanism NVMe uses to communicate with devices for shorter latency [167]. Like the concept of queue-pairs in NVMe [167, 111, 112] and several CSD prototype systems [32, 57], ActivePy maintains a function call queue for each CSD. This call queue resides in a CSD's memory location, and the CSD's CSE fetches a request from the call queue whenever the CSE is free. In addition, the CSD makes the call queue visible to the host, so ActivePy is able to monitor the CSD's current execution status.

Each function call request contains pointers to the entry memory location of the CSD function binary, and the memory page(s) store(s) the function call arguments. ActivePy patches code on the host caller side to fill in necessary pointers and argument values. On the called CSD functions, ActivePy also needs to patch code in retrieving the arguments and initializing the CSD functional's local variables with fetch values.

At the end of each CSD function, ActivePy also patches the CSD function code to fill the completion/response queue with a return value and the return status. ActivePy patches the host code to detect the change in the completion queue. The caller on the host side will continue execution after the host code is aware of the invocation of a CSD function call. ActivePy currently implements blocking CSD function calls to simplify the design of the runtime library.

ActivePy also patches the status update code that reports the current execution status, typically the execution rate, through the completion/response queue to enable feedback and migration when running a CSD function. ActivePy patches status update code only at the end of each line of code at the interpreted language level, typically once every tens of machine instructions. The status update code also checks if the host computer has any request that CSD needs to handle with high priority. The overhead of status update code takes very little overhead in code execution time.

Elimination of redundant memory operations

The design of conventional interpreted-programming-language runtime systems does not assume intensive use of external libraries and creates unnecessary memory copies and object transformations when invoking external functions. ActivePy also patches the code to avoid such overhead.

To remove these redundant memory copies, ActivePy places memory objects exchanged through different function calls directly into mutable memory, potentially a memory location on a CSD. By placing values in mutable memory objects, passing inputs when calling wrapper functions is similar to call-by-reference; the caller and callee share the same memory locations. If ActivePy can determine the target type of memory objects, the relevant library functions in ActivePy can produce memory objects in the target data type (e.g., NumPy) directly to the destination memory locations, further avoiding conversion overhead and bypassing the memory buffer.

3.5.4 Code generation and distribution

ActivePy leverages the code generation functions in Cython [22] to produce high-performance machine code. After ActivePy starts running the program and makes the decisions of task and data allocation, ActivePy invokes Cython functions, compiles the

CSD function into the CSD's machine binary and compiles the rest of the code into the host machine binary for performance.

Next, ActivePy distributes the CSD function using the mapped device locations. As mentioned before, CSDs make their device memory available to the host computer to support ActivePy. In this way, ActivePy can directly copy or emit the generated CSD binary into the target device memory location without additional commands or protocols.

3.5.5 Runtime monitoring and migration

During the runtime, ActivePy monitors the performance of code execution on CSDs and potentially adjusts the workload distribution for situations in which (1) the target device needs to handle high-priority tasks or (2) the device's projected performance does not match its real performance. For the first case, the CSD will signal ActivePy through the command pages to notify ActivePy to take corresponding actions immediately. For the second case, ActivePy will use the measured instructions per cycle (IPC) to re-estimate the time required for the remaining tasks on CSD if any of the following cases occur: (1) the rate of instruction throughput (i.e., IPC) from the CSD code segment is decreasing, or (2) the IPC is significantly below the estimated instruction throughput (i.e., the total amount of estimated instructions divided by the estimated execution time on CSD). If the re-estimated execution time is longer than the total cost of migrating the remaining task to the host computer, including the host computation time and the data movement overhead, ActivePy will initiate task migration to the host.

Once the system decides to migrate the CSD task, ActivePy will stop the CSD execution at the end of the currently executing line of Python code. With the help of a single memory abstraction for all computing units, migrating tasks among different computing units only requires ActivePy to save the local variables and the data in the shared memory space. ActivePy will regenerate the machine code for the new target-computing unit and resume execution of the offloaded code segment at the breakpoint of the Python code. The target-computing unit can use the regenerated machine mode and data/variables presented in the shared memory abstraction to resume execution from the breakpoint of the originally offloaded code segment.

3.5.6 The CSD

Any CSD with hardware components mentioned in Section 3.3.1 can work with ActivePy if firmware/software should implement at least essential supports, including (1) making device memory available to the host computer, and (2) a runtime environment to execute the CSD function. As mentioned earlier, all CSDs that support PCIe or NVMeoF implicitly provide the function of mapping their device memory locations to the host system's virtual memory abstraction. Therefore, this section will focus on the rest two.

The CSD can implement the runtime system support for ActivePy by extending existing FTL firmware or the CSD's embedded OS. The extension requires at least two basic features. First is the ability to handle the function calls and completion/response queues. Second is the ability to set up the CSD runtime to execute CSD functions. During the initialization process, the CSD works with the host computer to agree on shared locations for a doorbell register (on the CSD) and queue locations (on the host). The host computer sets the doorbell register as soon as ActivePy enqueues an event or a request to the CSD. The firmware program or an OS process on the CSD can check the doorbell register whenever the computing resource is free. If the request invokes a CSD function, the CSD firmware/process will set its program counter to the called function's memory address on the CSD to start executing the called function. The processor will dedicate itself to the CSD function either until the end of the execution, or ActivePy decides to change the running workload. The patched CSD function code will reset the firmware/process to the correct state to continue receiving new requests when the running function completes, or ActivePy decides to migrate the current CSD function.

With conventional storage system stacks, programs use file abstractions to identify storage data. So a CSD has two choices if it has no knowledge of the file structure within its own storage array: (1) it can make multiple round trips to query the host file system for data locations, or (2) it can rely on the host program to pass commands in data addresses. Unfortunately, both options limit the granularity and flexibility of ISP because (1) multiple round trips hurt the latency of accessing data, and (2) NVMe [8, 112] only allows at most 32 MB data access in each command.

We implemented an in-device file system that extends DevFS [69] on our CSD. The in-device file system provides library functions for ActivePy to patch code and emit binary to access the in-device file system. The host computer's software stack also supports the DevFS-like file system in our CSD. Using a DevFS-like file system allows the CSD program to access files on the CSD efficiently and thus minimizes the number of I/O commands and round-trips for file/address lookups. In addition, DevFS helps the host program enjoy the benefits of direct user-level access while maintaining conventional file system abstractions and properties, including integrity, concurrency, consistency, and security guarantees.

3.6 Experimental methodology

This section describes both the hardware platform and the applications that we established to evaluate ActivePy.

3.6.1 Experimental platform

The Host Computer

The experimental platform we used has an octa-core AMD Ryzen 7 3700X processor with a base clock rate of 3.6 GHz. We installed Ubuntu 16.04 (Linux kernel version 4.15), extended the language runtime and added user-space drivers to support the ActivePy model. The motherboard contains a PCIe 3.0 I/O hub that connects the processor and other peripherals, including a Mellanox InfiniBand NIC that connects to the CSD. The machine also contains an NVIDIA RTX 2080 GPU to enable compute-intensive kernels.

The CSD

The prototyped CSD includes a system-on-chip (SoC) with 8 ARM Cortex-A72 processor cores and uses 2 TB flash memory arrays for data storage. The SoC and the device DRAM have access to the internal NAND flash-based arrays through an internal interconnect. We measured an effective peak bandwidth at 9GB/sec when accessing the internal NAND array. The CSD uses NVMeoF [112] to communicate with the host computer with up to 5GB/sec bandwidth, almost half of that to the internal device bandwidth. In addition, the CSD's hardware supports RDMA/InfiniBand, which can help to map the CSD's internal memory locations to the host computer.

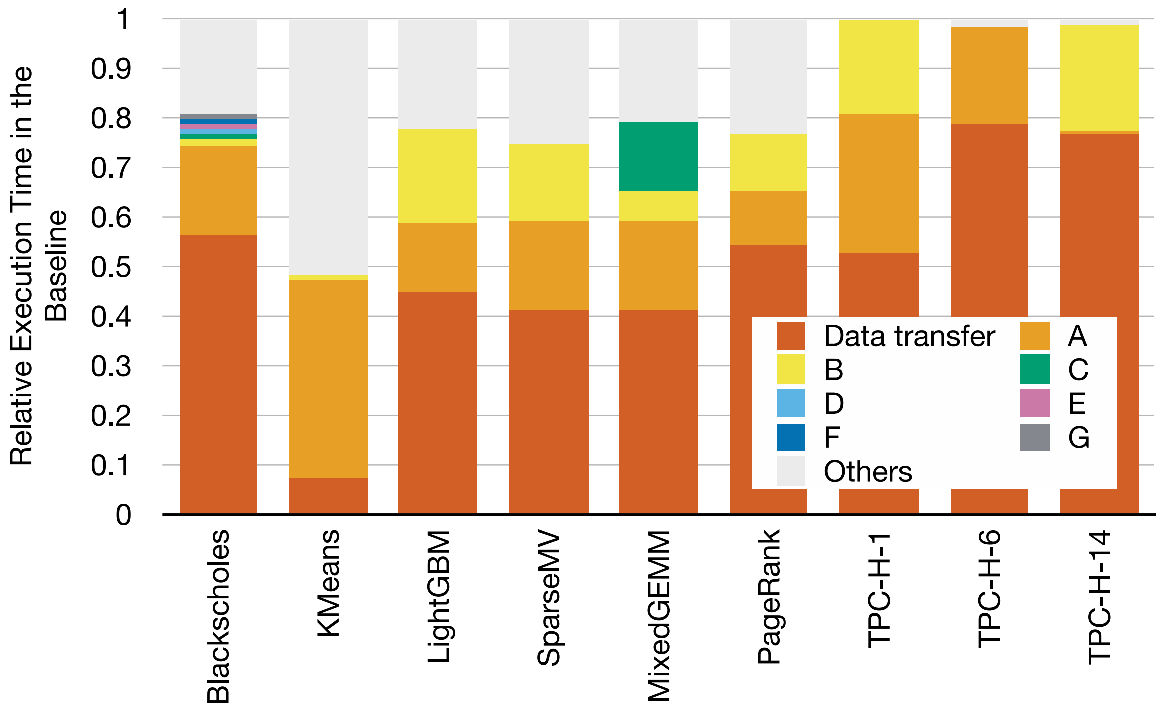


Figure 3.5: The latency breakdown of SSD to host data movements and single-entry-single-exit code regions in unmodified baseline applications.

The resulting hardware assembly and the software stack match the specifications of commercialized products or prototypes [78, 45]. In addition, the resulting CSD also delivers similar performance gain as prior research prototype [78].

3.6.2 Applications

We used nine Python workloads to evaluate the performance of ActivePy. Table 3.1 provides a summary of each application's input data size and code regions. We select these workloads as these applications have both Python and C implementations with optimized compute kernels to allow this work to compare the performance of optimized, programmer-directed versions and with the Python version automatically optimized by ActivePy. These applications are also proven beneficial through using CSDs with general-purpose processors [155, 57, 78, 62]. In order to simplify the discussion on code segments, we partition the meaningful code regions in each benchmark application into code regions labeled from A to G and other host functions.

Figure 3.5 breaks down the latency of unmodified applications into SSD to host data transfers (SSD/Host), code regions (A – G), and other host functions with C implementations (Other). We translated the Python code and optimized these applications in C before measuring their performance. For these applications, moving data between SSD and the host machine accounts for 50% of the latency, and those code regions that ActivePy can potentially leverage the CSD account for another 32% of time in the baseline. These applications only spend 18% of time in code regions with highly optimized C/C++-based or hardware-accelerated backends. This result shows that the code regions and latency issues that ActivePy addresses (i.e., 82% in total) are the most critical part of the performance. We now briefly describe the baseline implementation of these applications.

Black-Scholes

Black-Scholes is a mathematical model for pricing stock options. The baseline Black-Scholes does not require double-precision floating-point numbers [171]; however, GPU kernels perform best with single-precision floating-point applications, so it is advantageous if a baseline application converts raw data into single-precision floating-point values before the compute kernel starts.

For this application, we fed an input file containing 201 million records, each with six attributes, stored in a file containing a matrix with IEEE 754 double-precision binary formatted numbers. The raw data size of 9.1 GB. The baseline code first reads the input. Then, the code splits columns into arrays containing different attributes (A). The single-entry-single-exit code regions B–G then convert each attribute array from 64-bit double-precision numbers into 32-bit floating-point numbers that the GPU kernel can accept.

K-Means

K-Means is a popular clustering algorithm that forms the kernel of many data-mining applications. For a baseline K-Means application, we chose a sample Python program from kmcuda [95]. Kmcuda contains a highly optimized GPU kernel that implements the Yinyang algorithm [173]. The baseline implementation receives 5.3 GB input data in ASCII format and converts the input into binary integers (A). As Kmcuda's kernel code only accepts floating-point inputs, the baseline code needs to additionally convert and create floating-point inputs for the kernel (B).

LightGBM

LightGBM [103] is a gradient-boosting-based machine-learning framework with both Python/C++ front end. Since the compute-intensive training process relies heavily on high-performance CPUs/GPUs, we focused on inference/prediction, which is more data-intensive and latency-sensitive. The program begins by creating memory objects, including a trained model and 7 million instances, each with 256 attributes from a 7.1 GB default raw input file (A). Then, the program feeds the data to the inference model and makes predictions on each. Finally, the program runs root-mean-square-error (RMSE) on predicted results and ground-truth values to check the model's accuracy (C).

Matrix-Vector multiplication

Matrix-vector multiplication (MV) is a fundamental operation from linear algebra that is essential to mathematics, statistics, physics, economics, and engineering applications. The program reads input (A) and checks the sparsity of the matrix (B). Note that the program dynamically adjusts the computation kernel depending on the density of the input data. The density of the input data determines whether the kernel uses dense matrix multiplication or sparse matrix multiplication to maximize the efficiency of matrix multiplication. If a sparse matrix is chosen, the program needs to additional convert the input to CSR-encoded matrices (C). For this application, we used a 6 GB default input with a 40K by 40K binary-encoded sparse matrix with only 0.4% non-zero values, unless otherwise mentioned.

Mixed-Precision Matrix-Matrix multiplication

Mixed precision matrix multiplication (MixedGEMM) improves the computation throughput on the same GPU hardware by performing operations using less-precise half-floating point numbers and slightly trade-off the accuracy of results. To use a mixed-precision matrix multiplication library, the application needs to create matrices from raw input files (A), converts those matrices to half-precision (B), and transfers those matrices to the computation kernel from cuBLAS library that only accepts half-precision inputs. The input data we used is a 9.4 GB binary file that contains two 25K by 25K double-precision matrices.

Pagerank

PageRank is a graph algorithm that estimates the importance of a page by counting the number and quality of links to the page [114]. The graph we used for this application is ak2010 [180] generated from U.S. Census 2010 and Tiger/Line 2010 shapefiles. It is a 7.7 GB binary containing a 45K-by-45K integer adjacent matrix. This application will build the graph from the binary (A), convert the graph to the CSR format (B), and invoke the PageRank compute kernel from the Gunrock library [161].

TPC-H

TPC-H is a set of representative workloads in decision support systems for critical business questions [152]. We leverage the CSD-assisted implementations of Q1, Q6, and Q14 from Summarizer [78] to evaluate whether the same implementations work on our framework and the capability of ActivePy in adapting the variance in prototype systems. For Q1 and Q6, Summarizer's implementation offloads the where condition to CSD and sends the selected records for group-by-and-aggregation operations on the host. For Q14, Summarizer's default CSD-assisted implementation checks the where condition inside CSD and transfers only items of records essential for hash-join-and-aggregation on the host computer.

3.7 Results

ActivePy automatically identified appropriate code regions for the CSD to execute and achieves an average speedup of 1.33× (Section 3.7.1), which is the same level of a speedup as conventional C-based frameworks, but ActivePy requires no programmer's intervention. With ActivePy's ability to dynamically migrate tasks between the host and the CSD when the CSD is overloaded (Section 3.7.2), and ActivePy's sensitivity to input datasets to dynamically adjust the computation kernels (Section 3.7.3), ActivePy avoids the potential slowdown that conventional compiled-language-based ISP frameworks suffer.

3.7.1 ActivePy's automatic ISP code generation

ActivePy's overall performance

Automatically identifying CSD code regions and generating efficient code are essential ActivePy features. We evaluated ActivePy with a set of Python-based applications described in Section 3.6.2 with large datasets and compared their performance with optimal programmer-directed, C-based CSD-assisted programs performing the same tasks.

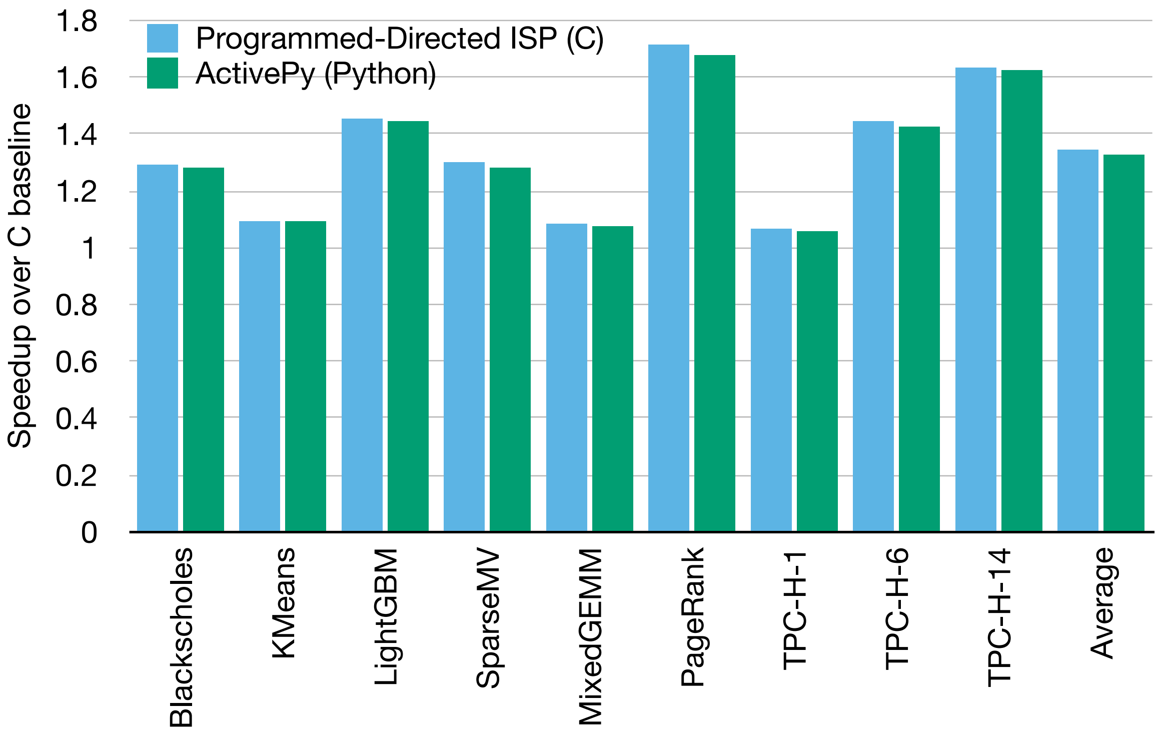


Figure 3.6: The speedup of ActivePy compared to the speedup of a static, C-based ISP platform

In this evaluation, we presented two versions of ISP implementations: the programmer-directed ISP and ActivePy to gauge ActivePy's ability to automatically create efficient CSD code from unmodified code. The programmer-directed ISP is a collection of manually optimized ISP programs written in C, and it runs on the same hardware platform using the system infrastructure as one used in ActivePy. In C-based, programmer-directed implementations, programmers have to manually specify/compose ISP code regions. On the other hand, the proposed ActivePy does not rely on *any* programmer's hint to automatically identify and generate ISP code regions.

To create an optimal programmer-directed code for each C application, we exhaustively tried to offload all reasonable combinations of single-entry-single-exit code regions, mentioned in Table 3.1, to the CSD that entirely dedicates itself for offloaded tasks. Then, we select the combination that delivers the shortest end-to-end latency as the optimal programmer-directed version for each application.

The result in Figure 2.13 shows that ActivePy without any programmer's hint achieves almost the same performance gain as optimal programmer-directed ISP programming. Figure 3.6 shows ActivePy's performance for the total execution time of these applications when the CSD fully dedicated itself to the running application. The execution time of the baseline workloads varies from 11 secs (TPC-H-6) to 73 sec (KMeans) on our machine, and we normalize the speedup to the baseline application (C-based, without ISP). Because ActivePy successfully identified *exactly* the same set of code regions for our CSD to perform as the optimal programmer-directed configuration, ActivePy achieves almost the same performance gain as the programmer-directed ISP (1.33$\times$ vs. 1.34$\times$) The small (1%)

performance difference between ActivePy and the programmer-directed ISP reflects the overheads of the sampling phase, the code-generation phase and the migration mechanism, typically 0.1 sec latency. However, compared to the time efforts in exhaustive profiling and working closely on the CSD, the 1% overhead is negligible.

The performance of our CSD prototype

When evaluating TPC-H Q6 and Q14 queries, ActivePy delivered similar performance gain as Summarizer in their cases of a 1:2 external to internal bandwidth ratio, showing that the prototype we built resembles the performance of conventional research CSD prototypes. However, as the host CPU in our machine is more powerful than the one used in Summarizer's paper, for TPC-H Q1 that I/O is less significant than Q6 and Q14, and the end-to-end latency improvement on our platform is less significant.

Although the same code regions would underperform the high-end host processor by 21%, due to the capability of processors in our CSD, making these code regions into CSD functions helps to reduce the volume of data transfers and achieve 506× speedup in data movements. The overall benefit of offloading these code regions leads to a 1.52× speedup compared with finishing the same set of tasks on the host machine, mainly from the improvement in data exchange overhead.

ActivePy's capability in identifying and composing CSD code

The data volume change in each code region is the main factor affecting ISP program performance, and ActivePy's mechanism usually makes very accurate predictions on this change. As the data volume reduction after processing on our CSD is the main factor

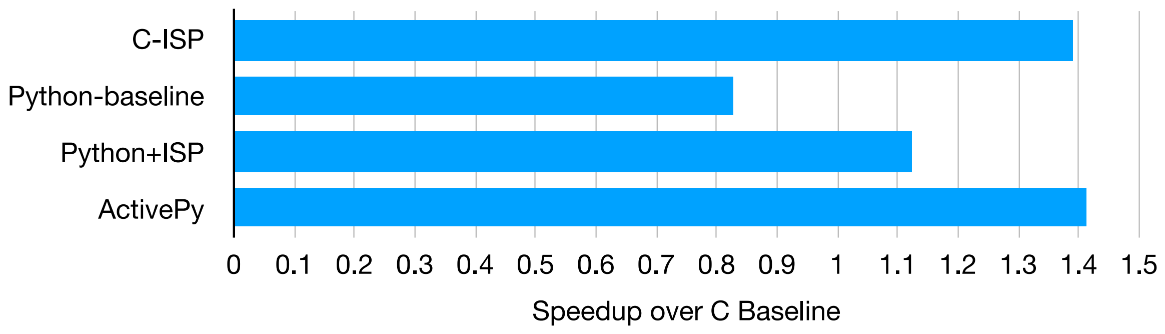


Figure 3.7: The speedup of different ActivePy configurations compared to the speedup of a static, C-based ISP platform

that leads to the performance gain, the accuracy in predicting volume changes compensates for the errors in estimating computation time. The only exception is the conversion to CSR format in PageRank and SparseMV – ActivePy can over-estimate the data volume that our CSD produced after generating data in CSR format by up to 2.41×. The geometric mean of our error rate that discounts the outliers (e.g., CSR format) is only 9%. ActivePy does not predict accurately for CSR format, because the sparsity is challenging to estimate with the limited number of samples we created in our algorithms. However, our experiments on different input matrices show that ActivePy always over-estimates the data volume after generating CSR on our CSD, meaning that our algorithm underestimates the potential of our CSD. Therefore, ActivePy at least does no harm to performance if ActivePy conservatively schedules tasks on the host machine due to the under-estimated data reduction benefit from our CSD.

ActivePy's optimizations in its language runtime

It is worth mentioning the performance of code optimizations of ActivePy in Section 3.5.4. Figure 3.7 shows the average speedup of running our workloads with various ActivePy runtime optimizations enabled, compared with the baseline C-based CSD framework. Without any optimization, the baseline Python code without using CSDs is 41% slower than the C baseline due to the overhead of the original Python runtime. Using a Python-to-C compiler (e.g., Cython) to generate code helps close the performance gap and shrink the slowdown to 20%. By eliminating unnecessary memory copies, the end-to-end latency of running the baseline Python program (ActivePy program without using ISP) makes almost no difference as the C baseline, excluding the 1% compilation overhead.

ActivePy's estimation of execution time

In terms of the estimated execution time on running ISP code regions, the average error rate of ActivePy's prediction on the host performance is 12.6%, with a standard deviation of 5.9%. Our average accuracy is significantly better than XAPP, a state-of-the-art machine-learning-based approach [13]. XAPP's error rate is 22.4%, 1.78× of ActivePy. ActivePy's geometric mean of the error rate that discounts the outliers is 11.1%, very close to XAPP.

3.7.2 ActivePy with dynamic task migration

Unlike traditional compiled-language-based platforms that cannot easily migrate tasks once assigned and statically compiled, ActivePy can reassign task locations and gen-

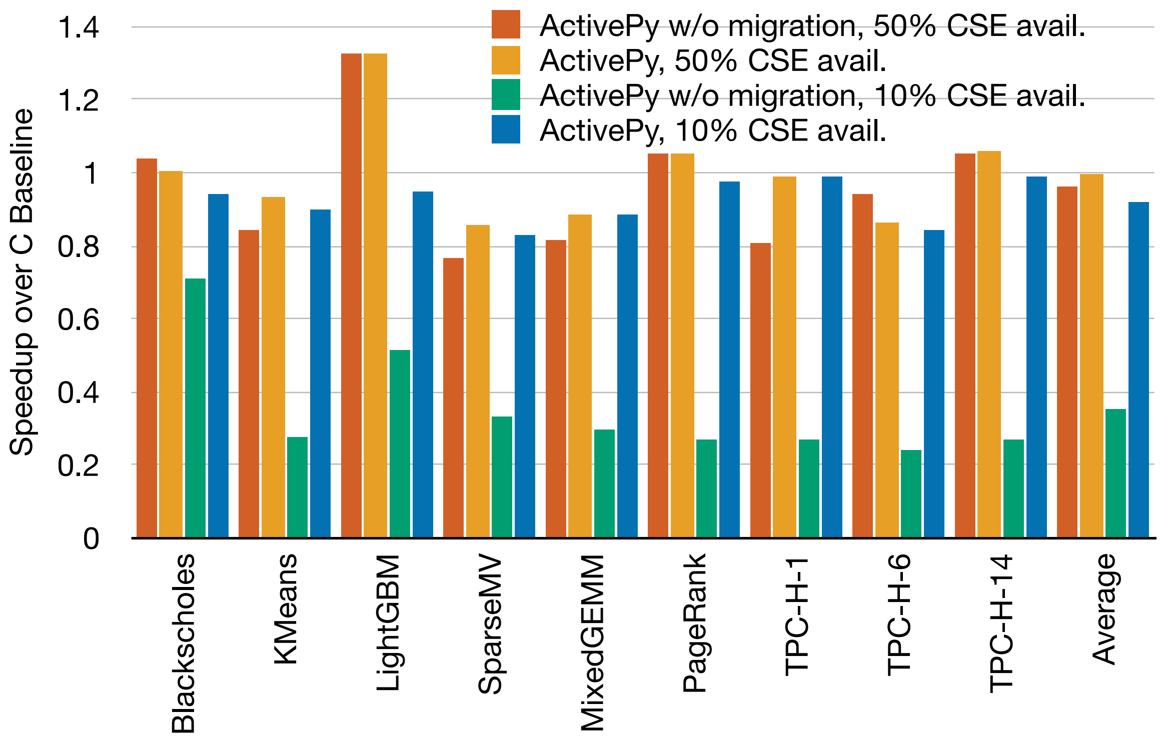


Figure 3.8: The speedup of all workloads on ActivePy under 50% and 10% available CSE resources with and without task migration

erate high-performance binaries on each computing unit with reasonably low overhead. We created a version of ActivePy that cannot migrate tasks dynamically (ActivePy w/o migration) to compare the performance without migration mechanism to the performance of the full-fledged ActivePy. To demonstrate ActivePy's task-migration capabilities, we stressed the CSD processor by executing similar workloads right after each application's ISP tasks on the CSD start making their progress in simulating a situation where the CSD must load multiple tasks.

Figure 3.8 lists the cases when the availability of CSD computing resources is only 50% and 10% to ISP workloads. In both cases, full-fledged ActivePy outperforms ActivePy w/o migration. When the CSD has only 10% computing resources available for the assigned CSD tasks, ActivePy w/ task migration outperforms ActivePy w/o task migration by 2.82×. Relative to the no-CSD-assisted, baseline condition, ActivePy suffers an 8% slowdown on average for the overhead of regenerating code on the host and accessing live data in CSD from the host computer. However, compared to the baseline, which does not offload tasks to CSD at all, ActivePy without migration mechanisms that always allocates tasks on CSD can lead to an average of 67%, up to 88% performance loss when only 10% computing resource is available, This demonstrates the importance of the ActivePy migration mechanism, which helps to mitigate the performance degradation caused by static task-allocation ISP programs.

When the CSD has 50% computing resources, the case shows that ActivePy is still capable of balancing the trade-offs of migration or slower ISP tasks. In the case of 50% computing resources available, ActivePy decides to migrate for Blackscholes, KMeans,

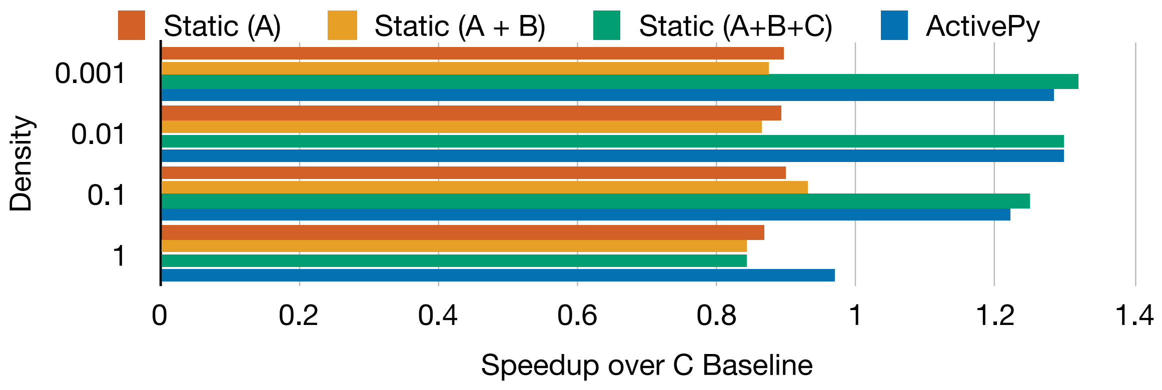


Figure 3.9: The speedup of different task-allocation strategies using different datasets

SparseMV, MixedGEMM, TPC-H-1, and TPC-H-14. ActivePy's decisions outperform ActivePy w/o task migration in all cases except for Blackscholes.

3.7.3 ActivePy's sensitivity to input datasets

Another dynamic that affects program task allocation is the input dataset. We used sparse matrix-vector multiplication to illustrate the effectiveness of ActivePy in achieving performance gains, independent of input-dataset type. Figure 3.9 compares the performance of ActivePy with various programmer-directed, static task-allocation strategies using different datasets. ActivePy can make decisions that are always close to the fastest programmer-directed strategies.

Figure 3.9 shows experimental results for datasets with different data densities. The x-axis in Figure 3.9 represents the fraction of non-zero elements in the raw data we created, and the y-axis shows the speedup over the baseline program (without any assistance from CSD). The baseline program fetches raw data, converts the raw data into python objects, and scans the sparsity of the input data. If more than half of the input-data elements (density is larger than 0.5 or 50%) are non-zero, the baseline program classifies the data as a dense matrix and invokes the GPU kernel to multiply input matrices. Otherwise, the baseline program converts the input data into CSR format and invokes the GPU kernel optimized for sparse matrices.

We tested the complete ActivePy and the programmer-directed, static version (the programmer-directed ISP in Section 3.7.1) using three different strategies for each dataset: (1) *Static (A)*, which statically offloads code region A, (2) *Static (A+B)*, which extends *Static (A)* so that scanning for sparsity is also statically offloaded to our CSD, and (3) *Static (A+B+C)*, which extends *Static (A+B)* so that CSR format creation is statically offloaded to our CSD if the input data is sparse enough (meaning that all tasks except for those of GPU compute kernels are offloaded). The first two strategies do not benefit from bandwidth reduction as our CSD adds metadata and parses data without reducing data volumes, but the first two strategies allow our CSD to exploit internal parallelism. The third strategy uses internal parallelism and significantly reduces the resulting data size if the matrices are sparse.

The third strategy, *Static (A+B+C)*, performs the best among all cases with sparsity less than or equal to 50%; however, the baseline performs better when sparsity is greater than 50%. This performance difference comes from the computation overhead in forming Python objects in CSR format, the cost of which exceeds the benefit of exploiting our CSD's internal parallelism. Fully automatic ActivePy, on the other hand, can identify the need for Python-object creation in its profiling phase, so ActivePy can intelligently identify the task assignments achieving the best performance among all possible configurations.

3.8 Related works

In addition to ISP and runtime compilation, ActivePy leverages the insights gained from prior work in the areas of near-data processing, RDMA, and parallel language runtime systems.

Other near-data/in-memory processing

In addition to near-data processing using CSDs (a.k.a. ISP), embedding/utilizing intelligence in SmartNICs [170, 47, 92, 162, 117], Computational RAM (C-RAM/PIM) [51, 116, 151, 46, 93, 77] also exhibits huge potential in accelerating data-intensive applications, similar to the set of applications that CSDs can help. Sharing the same programming issues with CSDs, existing SmartNICs and C-RAM also heavily rely on programmers' knowledge to statically map code regions to these units. ActivePy's design philosophy would help address similar issues for these platforms. In fact, we can easily extend ActivePy to quite a few SmartNICs as many of them use the same processor core architecture with ActivePy.

Efficient datapath

Inspired by prior efforts in using RDMA [160, 9, 65, 145, 122, 113, 159, 24, 70] and PCie P2P [110, 6, 137, 176, 155, 90] to achieve zero or almost zero memory copies for data exchanges among different system components, ActivePy's memory abstraction intensively used this infrastructure in providing efficient communication among peripherals (e.g., CSDs, GPUs and FPGAs).

Parallel and distributed computing in dynamic languages

There are many existing runtime systems that replace or remedy the inefficiency of dynamic-based languages in parallel or distributed computation by improving the language's global locking mechanism in single multithreaded, multicore computers [147, 42, 43] or task-scheduling mechanisms in distributed machine clusters [99, 100, 126, 50, 148]. Dandelion [128] also explored the idea of using a dynamic language in scheduling tasks in accelerator-based cloud servers. Comparing the hardware capabilities in CSDs and those for high-performance computing, ActivePy deals with processors with stringent constraints that any overhead and inappropriate scheduling decisions can lead to unwanted consequences.

Both ActivePy and Popcorn [18] use a shared memory model for computation located on heterogeneous processors to facilitate task migrations. ActivePy additionally allows memory regions to map to different devices and further reduces the data exchange traffic, while Popcorn simply makes the system main memory visible to all and requires mechanisms to cache and maintain coherency. ActivePy also shares a similar problem in distributing computation to computing resources while considering the cost of data movements as in mobile computing. MAUI [39] and ActivePy both partitions the tasks dynamically. However, MAUI adopts a language back-end similar to Java that does not require recompilation of code, but ActivePy emits the native machine code of the target computing units, potentially more efficient and less resource-consuming. CloneCloud [38] uses static analysis and dynamic profiling to partition applications, where the profiling method is similar to ActivePy. However, without ActivePy's infrastructure that shares memory regions among the host main memory and peripherals, CloneCloud cannot avoid the memory, latency, and movement overhead of "cloning" data. To predict the execution time in assisting the decision of task allocations, prior work uses sampling to collect data points and then uses regression-based models [168, 19] or machine learning models [13]. ActivePy's prediction does not rely on complex models with the least overhead while maintaining the same level or better accuracy.

Location-based task scheduling

As data movement becomes the major source of overhead in parallel architectures, taking data locations into account when scheduling tasks becomes increasingly important. TOM [60] assigns tasks to GPU cores near their data source in device memory. AMS [154] exploits a similar idea in a system setting with non-uniform memory access (NUMA) that lacks compiler support, while Kislal et al. [74] solve the data-location problem through compiler optimizations. However, unlike ActivePy, all of the above focus on each homogeneous computing unit within heterogeneous computing environments, which contain different components such as CPUs, GPUs, CSDs and etc.

3.9 Conclusion

This chapter presents ActivePy, an intelligent, dynamic-language-based programming framework for ISP. ActivePy makes ISP more accessible to application designers by automatically identifying ISP tasks and generating code that works on processors near data storage without programmers' interventions. Unlike traditional ISP frameworks that generate static task-allocation strategies, ActivePy can dynamically migrate tasks among different compute units to prevent performance degradation when system dynamics change.

Through experiments conducted with the prototype ActivePy platform, ActivePy demonstrates that interpreted languages with our proposed optimizations can be as competitive as compiled languages. ActivePy successfully identifies ISP use cases in a diverse range of applications, yielding a speedup of 1.33×, the same level as the results of the programmer-directed version. However, ActivePy's flexibility is underscored by its capacity to migrate ISP tasks among different units while minimizing the impact of inappropriate task assignments when system dynamics change. Without ActivePy's migration mechanism, it leads up to 88% performance loss when only 10% CSD computing resource is available compared to ActivePy with a migration mechanism.

Algorithm 1 CSD code assignment

Input: P a collection of lines (L_0, L_1, …, L_n) in the original program

Output: (P_{host}, P_{csd}), a collection of lines in the host program and in the CSD program

```
1:  $T_{csd} = T_{host}$
2:  for each: $L_i \in P$ do
3:      if $i == 0$ or $L_{i-1} \in P_{csd}$ then
4:          $T_{L_i \in P_{csd}} = T_{csd}$ - $CT_{i,host} + CT_{i,device}$ - $\frac{Din_i}{BW_{D2H}} + \frac{Dout_i}{BW_{D2H}}$
5:      else
6:          $T_{L_i \in P_{csd}} = T_{csd}$ - $CT_{i,host} + CT_{i,device} + \frac{Din_i}{BW_{D2H}} + \frac{Dout_i}{BW_{D2H}}$
7:      end if
8:      if $T_{L_i \in P_{csd}} < T_{csd} \leq T_{host}$ then
9:          $P_{csd} = P_{csd} \cup L_i$
10:         $P_{host} = P_{host} - L_i$
11:         $T_{csd} = T_{L_i \in P_{csd}}$
12:     end if
13: end for
14: return ($P_{host}$, $P_{csd}$)
```

Application Name	Input Size	Description of Single-Entry-Single-Exit Code Regions						
		A	B	C	D	E	F	G
Black-Scholes	9.1 GB	Isolate columns from table	Convert spot prices, strike prices, interest rates, dividend, volatility, time					
KMeans	5.3 GB	Deserialize from ASCII to binary	Convert integer to float					
LightGBM	7.1 GB	Deserialize object to a model and a table	Run inference	Calculate RMSE				
MatrixMul	6.0 GB	Create memory objects (matrices) from binary inputs	Count sparsity	Generate CSR format				
MixedGEMM	9.4 GB	Create memory objects (matrices) from binary input	Convert double-precision to half-precision					
Pagerank	7.7 GB	Build an adjacent matrix graph representation from the raw binary input	Convert the adjacent matrix to CSR format					
TPC-H-1	6.9 GB	Filter and project requested attributes from lineitem table	Aggregate attributes and sort results based on the return flag and the line status					
TPC-H-6	6.9 GB	Filter and project requested attributes from lineitem table	Sum up price times discount as revenue					
TPC-H-14	7.1 GB	Filter partkeys from the part table and construct a partkey set	Select items from the lineitem table which key exists in set and date matches query and aggregate results as promotion revenue					

Table 3.1: The applications, their input data sizes and description on their Single-Entry-Single-Exit code regions.

Chapter 4

UDSL: Universal Domain-Specific Languages for heterogeneous computers

4.1 Abstract

The conventional von Neumann architecture has incorporated heterogeneous hardware accelerators to fulfill emerging demands of high-performance computing in applications and compute kernels. Since those heterogeneous processing units have different processing models and architectures, architects have developed architecture-specific programming interfaces for programmers to access these components. However, architecture-specific programming interfaces make applications difficult to convert code snippets to others implemented in different programming interfaces and scale up the performance when adopting

different architectures. UDSL provides an application-specific programming interface that makes applications independent of architectural details. At the same time, UDSL's programming interface can fully utilize the computation power of underlying processing units by linking to highly-optimized standard libraries. In this way, UDSL makes applications easier to adopt and scale up with different processing units. To evaluate UDSL, we collected a set of applications implemented with architecture-specific compute kernels, and we replaced those compute kernels with UDSL's APIs, which implemented the same algorithms. The result shows that UDSL achieves 8.72× times speedup, proving that UDSL's programming interface provides both programmability and performance.

4.2 Introduction

As same story mentioned in Section 3.2 and Section 2.2, because of the rapid growth of application demands, the conventional von Neumann architecture incorporates hardware accelerators offering orders-of-magnitude performance gains to fulfill those application demands. We have seen processing units evolve from scalar processing model (e.g., CPU) and vector processing model (e.g., GPU) to matrix processing model (e.g., Tensor Core (TCU) [109] or Tensor Processing Unit (TPU) [66]). Nowadays, those processing units are assigned specific tasks according to their architectures in a heterogeneous computer. For example, a CPU helps the task scheduling and communications among different peripherals in a heterogenous computer because of the CPU's general-purpose instruction set architectures. A GPU is responsible for any graphics-related work, and a matrix processing unit, such as TPU, solves compute-intensive AI/ML algorithms that have been integrated into modern applications. Because of their domain-specific purposes, those hardware components have their own specific architectures for optimal performance in their domains.

Architecture-specific programming interfaces are designed for programmers to fully utilize those processing units in a heterogeneous computer by mapping the architectures and exposing the functionalities to the interfaces or APIs, and those architecture-specific programming interfaces can be mapped to different programming models. For example, the programming interface designed for scalar processing models prefers programmers to only requires programmers to think about the operation of each element on each step. Since vector processing units tend to spawn a bunch of threads for parallelism, the programming interface for vector processing models requires programmers to distribute data elements for threads to maximize the computation throughput. Modern matrix processing units are usually used on domain-specific applications, such as ML applications. As a result, the programming interface for matrix processing models is usually integrated into domain-specific languages/frameworks.

Although architecture-specific programming interfaces potentially allow programmers to fully utilize the computational power of processing units, those interfaces not only require programmers to have full knowledge of architectural details, but also require programmers to rewrite the code if an application wants to adopt a new processing unit with a different architecture. The learning curves and the efforts in adopting a new processing unit make programmers difficult to only focus on developing applications, since programmers also have to tune the application to guarantee performance at the same time.

Even though programmers spend time and effort in learning and tuning architectural details of the targeted processing units, the resulting customized kernel still might not outperform the standard library function implemented in the same algorithm by hardware vendors because (1) hardware vendors do not expose all hardware details or instructions to the programming interfaces. (2) customized kernels do not scale up with newer architectures of the same processing models. These reasons make using a programming interface to implement customized kernels less worthwhile.

As a result, we proposed UDSL, Universal Domain-Specific Language, to address the issues mentioned above caused by architecture-specific programming interfaces. UDSL proposes a new programming paradigm leveraging the idea of domain-specific languages that operations are exposed through a set of general APIs. These operations are frequently used in modern applications and exposed through UDSL APIs. Those APIs are mapped to functions from underlying standard libraries or highly-optimized backends that guarantee performance. Between the UDSL programming interface and architecture-specific libraries, we implemented a heterogeneous-aware runtime that helps applications to adopt and fulfill requirements for different architectures.

UDSL resolves the development overhead of adopting different processing units for applications and offers several benefits. First, UDSL programming interface exposes a set of architecture-independent APIs that makes applications agnostic to the architectural details of the targeted processing units. Second, UDSL's heterogeneous-aware runtime links UDSL programming interface to architecture-specific standard or highly-optimized libraries to guarantee optimal performance and make applications easily scale up with different processing units. Third, since UDSL's transparent programming interface, programmers have less overhead in developing applications and algorithms, while hardware vendors and backend engineers can still implement and optimize their standard libraries to provide more features and performance gains.

We built a heterogeneous computer and implemented a prototype UDSL to evaluate UDSL's performance compared to applications implemented with customized kernels in architecture-specific programming interfaces. UDSL demonstrates that programmers can simply replace customized kernels with UDSL's APIs implemented with the same algorithms, The result shows that UDSL applications achieve 8.72× times speedup, thanks to UDSL's heterogeneous-aware runtime linking to highly-optimized, architecture-specific libraries. Besides the performance, those UDSL applications do not contain any architecture-specific code snippets, so UDSL applications can easily adopt other architectures and be more portable on other computers.

This work makes the following contributions:

(1) It presents UDSL's application-specific programming interface that makes the applications agnostic to underlying hardware architectures.

(2) It proposes UDSL's heterogeneous-aware runtime that bridges UDSL programming interface and architectural details, and guarantees the performance at the same time.

(3) We collected a set of applications implemented with customized kernels that can potentially leverage the latest processing units.

(4) It evaluates UDSL by building a heterogeneous computer and experimenting with a set of collected applications mentioned above.

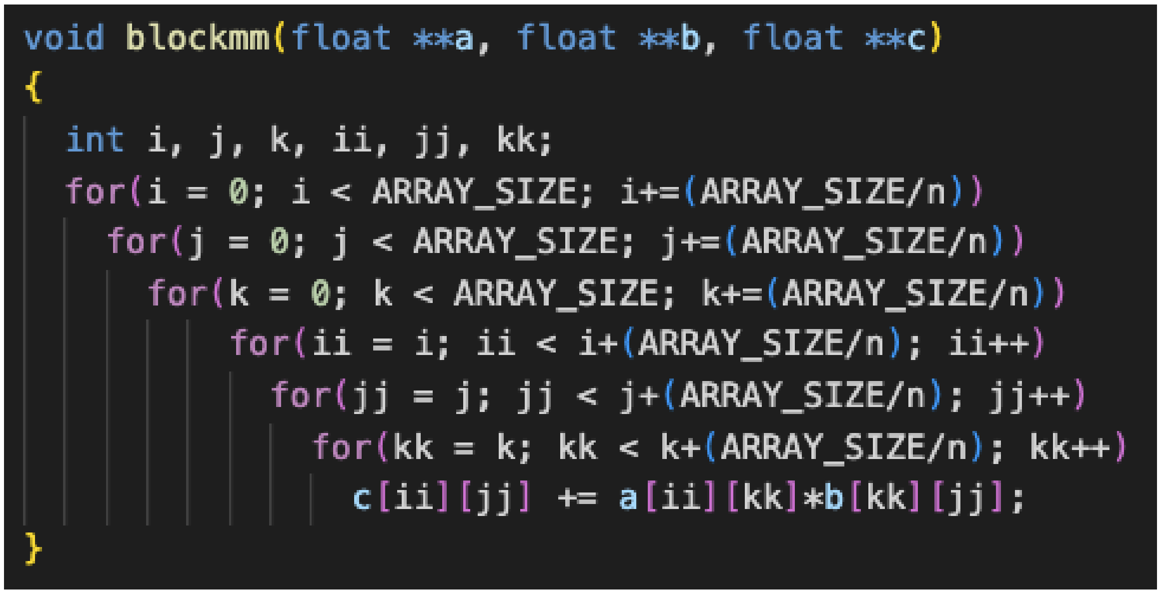

```c
void blockmm(float **a, float **b, float **c)
{
  int i, j, k, ii, jj, kk;
  for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n))
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n))
      for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n))
          for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
            for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
              for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                c[ii][jj] += a[ii][kk]*b[kk][jj];
}
```

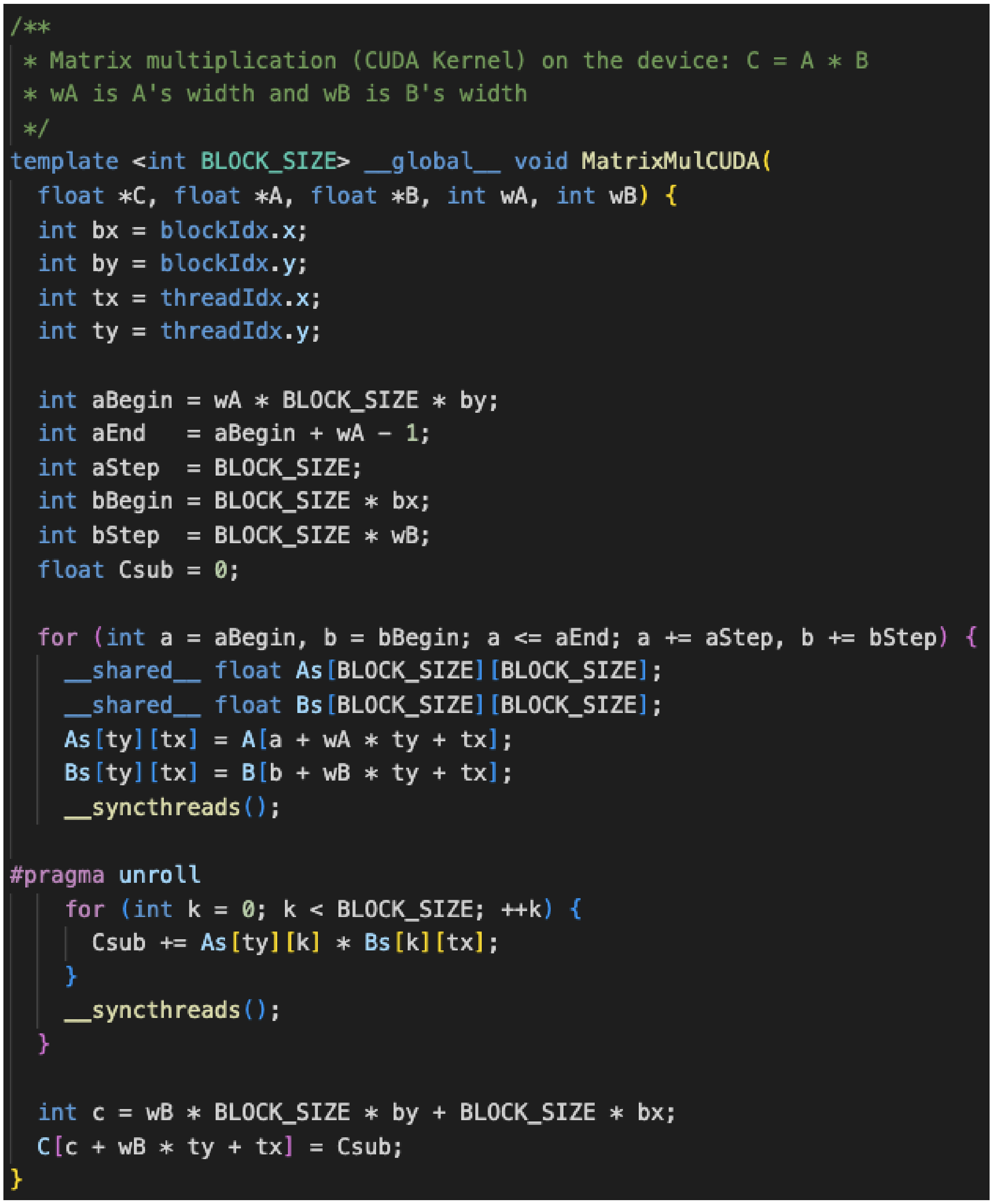
Figure 4.1: The CPU code example of matrix-multiplication implemented in C language.

4.3 Background

This section describes the problems introduced by architectures-specific programming interfaces.

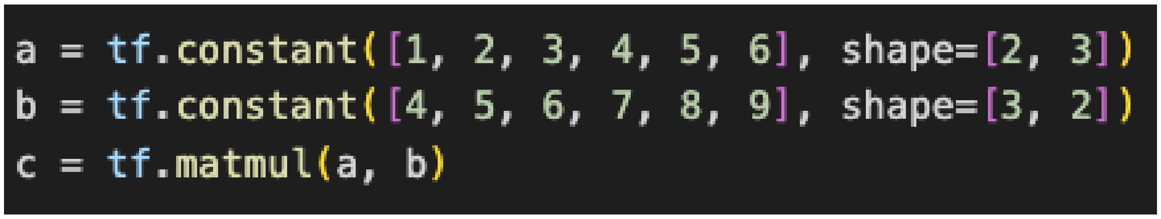
4.3.1 Inconvertible code snippets among architectures-specific programming interfaces

As mentioned in Section 4.2, each processing unit in a heterogeneous computer has a specialized architecture for different purposes, and the functionalities of those processing units are exposed by their architecture-specific programming interfaces. Such architecture-specific programming interfaces require programmers to have full knowledge of processing



```
/**
 * Matrix multiplication (CUDA Kernel) on the device: C = A * B
 * wA is A's width and wB is B's width
 */
template <int BLOCK_SIZE> __global__ void MatrixMulCUDA(
  float *C, float *A, float *B, int wA, int wB) {
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int aBegin = wA * BLOCK_SIZE * by;
  int aEnd   = aBegin + wA - 1;
  int aStep  = BLOCK_SIZE;
  int bBegin = BLOCK_SIZE * bx;
  int bStep  = BLOCK_SIZE * wB;
  float Csub = 0;

  for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];
    __syncthreads();

#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k) {
      Csub += As[ty][k] * Bs[k][tx];
    }
    __syncthreads();
  }

  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] = Csub;
}
```

Figure 4.2: The GPU code example of matrix-multiplication implemented in CUDA.



```
a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3])
b = tf.constant([4, 5, 6, 7, 8, 9], shape=[3, 2])
c = tf.matmul(a, b)
```

Figure 4.3: The TPU code example of matrix-multiplication implemented in TensorFlow framework.

units and make programmers spend more effort to implement and optimize the code. Besides programmers' efforts, since programming interfaces are dedicated to specific architectures, the programming models differ from one another and almost cannot be interchangeable. As a result, if an application wants to migrate from one processing unit to another, programmers have to almost rewrite the whole application to adopt the new architecture, which makes the applications unsustainable.

Figure 4.1 to 4.3 demonstrates the code examples that programmers have to implement to compute matrix multiplication on CPU, GPU and TPU. Figure 4.1 is the CPU's code example implemented in C language. Since traditionally CPU is a scalar processor, programmers only need to think about the operation of each element on each step. In this example, we can see that the CPU has to iterate both input matrices, fetch one element from both matrices respectively, compute the scalar result and put the result back into the output matrix one element by one element. On the other hand, Figure 4.2 shows a different programming model from the one in the CPU's code example. The code snippet in Figure 4.2 is the GPU version of the customized kernel implemented in CUDA. Since GPU is a vector processor that can spawn a bunch of threads at the same time, programmers have to consider the task for each thread and the interactions between threads on each step. In the GPU example, spawned threads first load the input matrices to the shared memory to optimize memory access. Then, each spawned thread takes care of each row or column from submatrices stored in the shared memory, calculates the partial results and merges the results coming from different threads. TPU's programming interface is a lot different from the aforementioned code examples. TPU is a highly specialized hardware accelerator designed for domain-specific problems, such as ML problems, that involve lots of matrices or higher-dimensional computation, so the architecture of TPU is able to process matrices in one instruction. Since TPU is designed for domain-specific problems, the operations are usually abstracted by domain-specific languages or frameworks, such as TensorFlow [2], that makes programmers only need to write fewer lines to implement the same algorithm compared to previous CPU and GPU programming interfaces. In Figure 4.3 that shows the TPU version of matrix multiplication, programmers only need to write down one line of code to make the application invoke matrix-multiplication compute kernel on TPU.

These code examples demonstrate that programming interfaces to utilize modern processing units are dedicated to specific architectures. Suppose programmers want to make applications adopt different processing units and guarantee performance. In that case, they have to fully understand those programming interfaces related to hardware architectures to use processing units appropriately. Also, programmers need to almost rewrite the whole application using different architecture-specific programming interfaces.

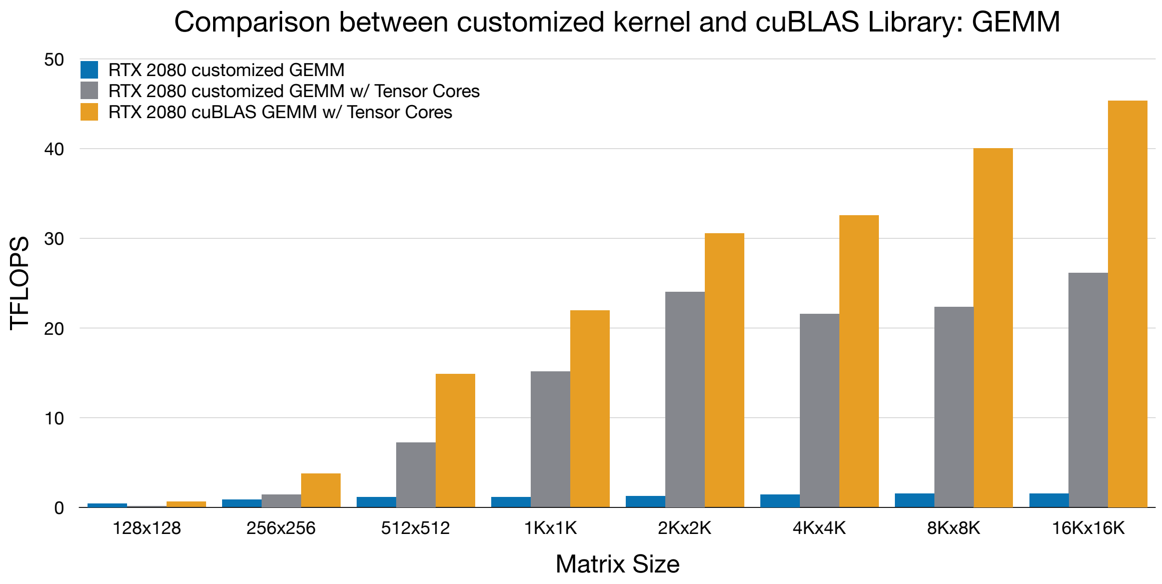


Figure 4.4: The performance of the customized matrix-multiplication kernel with GPU cores only, TCU enabled and cuBLAS GEMM APIs with TCUs enabled among different input/output matrices shape.

4.3.2 Under-utilized computational power through customized kernels

Even though programmers are totally familiar with the target's architecture and its architecture-specific programming interface to implement and optimize customized kernels for applications, the resulting customized kernels still might not outperform the same algorithms implemented in the standard libraries provided by hardware vendors.

Figure 4.4 shows the performance comparisons among four different hardware-accelerated implementations running matrix multiplication on NVIDIA RTX 2080 GPU. We implemented two customized kernels: one with GPU CUDA cores executes matrix multiplication through a vector processing model, and the other leverages TCUs, matrix processors on NVIDIA's modern GPU architectures, to execute matrix multiplication. These implementations are collected from NVIDIA's official CUDA examples [107] To compare with those customized kernels, we implemented the same applications but replaced the customized kernels with the cuBLAS GEMM API with TCUs enabled [106]. The x-axis indicates the shape of the input matrices and the output matrix, and the y-axis shows the performance numbers in TFLOPS (Tera-floating-point-operation per second).

With the largest matrices (16384×16384 floating-point elements in our example) already presented in the limited capacity of the device's memory, the optimized, customized kernel without leveraging TCUs is able to achieve 1.51 TFLOPS, which is the baseline in the following experiments. However, one may argue that the implementation with vector processing units is not optimized enough, since there exists matrix processors, TCUs, on the latest GPU architectures. Unlike TPU that is only exposed through the domain-specific framework, NVIDIA's TCU is exposed through its CUDA programming interface [108], which allows programmers to use the architecture-specific programming interface to access these matrix processors. We ran the same experiment but replaced the compute kernel with the one running on TCUs, and got 26.1 TFLOPS this time, 17.3× speedup compared to the baseline. The sources of performance improvements are from: (1) helps from specialized matrix processing units that reduce the matrix calculation to 1 instruction. (2) reduction in memory traffic in the device since TCUs take IEEE 754 half-precision floating-point format as inputs.

Instead of using CUDA, APIs in NVIDIA's standard libraries, such as `cublasGemmEX()` function in cuBLAS, also allow programmers to utilize TCUs by setting the flags correctly. Figure 4.4 shows that using such GEMM API from a standard library, with TCUs enabled, the performance number is able to achieve up to 45.33 TFLOPS on

RTX 2080. The resulting number is 30× faster than the baseline, and 1.73× faster than the customized GEMM kernel with TCUs enabled. More importantly, compared to adopting from GPU to TCU through customized kernels that requires programmers to rewrite the whole compute kernels to fulfill demands of the underlying architecture, collaborating with TCU through standard libraries' APIs only need programmers to simply change one or two lines of code in APIs' configurations. However, compared to rewriting the whole kernels to adopt the new architecture, a simple modification through APIs not only makes programmers more focused on application development, but also gives programmers a significant performance gain instead of putting time into performance optimization.

4.4 Overview of UDSL

The proposed UDSL, Universal Domain-Specific Language, follows the two design principles to address issues mentioned in Section 4.3.1 and Section 4.3.2.

Make applications easily adopt different architectures.

As noted in Section 4.3.1, architecture-specific programming interfaces tend to be more general-purpose and have completed sets of control flow, expression, operators, datatypes and data structure capabilities. Although these features grant programmers more flexibility and more power, they make application development more complicated at the same time, since programmers have to learn a different programming paradigm and details of the targeted architecture, and tune compute kernels for optimal performance.

UDSL offers a programming interface, like the concepts of domain-specific languages, abstracting a set of application-specific operators. The resulting programming interface is independent of underlying processing units, and it helps programmers to stay away from low-level architectural details and to focus on the algorithms for applications. At the same time, applications using UDSL become more sustainable since UDSL does not require programmers to rewrite the code to adopt different processing units.

Make applications easily utilize the full capabilities of the targeted processing unit.

The microbenchmark in Section 4.3.2 indicates that customized kernels implemented by an experienced programmer may still not outperform standard libraries' APIs implemented with the same algorithms, as shown in Figure 4.4. This experiment gives us an insight that the proposed programming interface should stick with standard libraries if possible, since standard libraries provided by hardware vendors usually guarantee the optimal performance of target devices. UDSL's heterogeneous-aware runtime helps its programming interface connected to highly-optimized, architecture-specific standard libraries through different adaptors in the runtime. Those adaptors are able to fulfill the missing pieces (e.g., parameters, device configurations) between the front-end and the back-end. As a result, this runtime helps to make the programming interface agnostic to the underlying architectural demands.

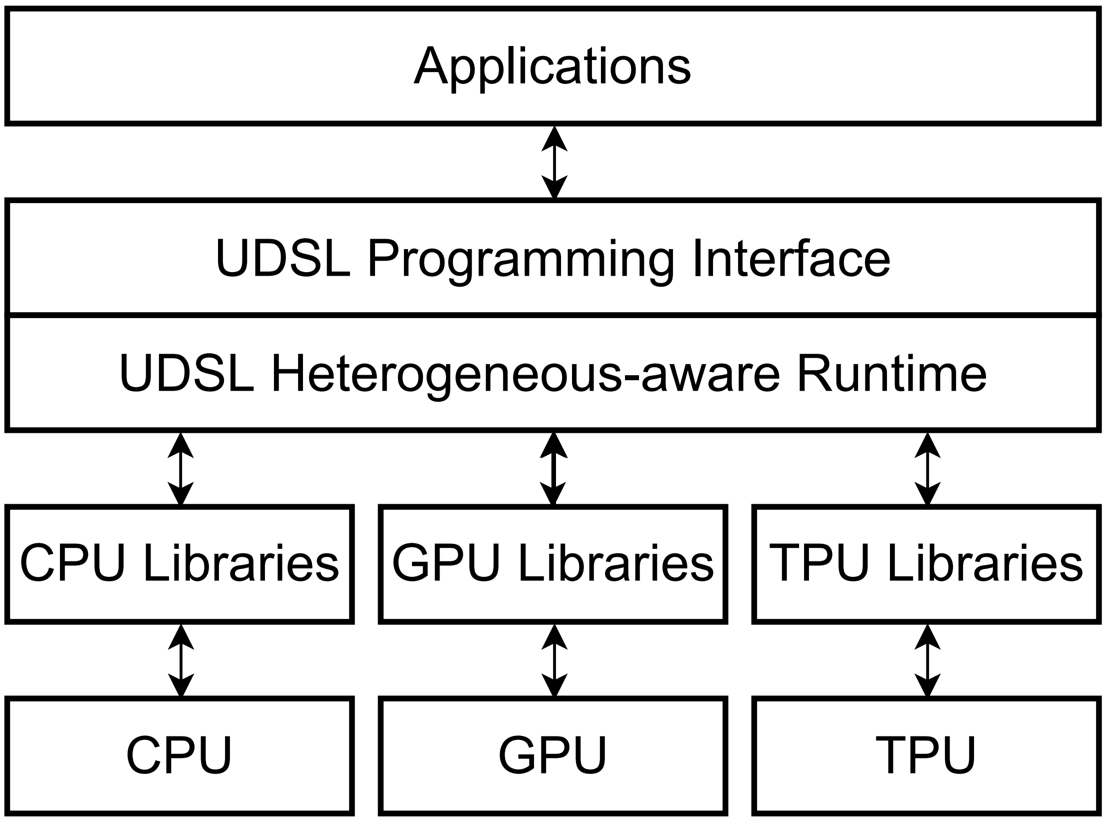


Figure 4.5: UDSL's Architecture

Figure 4.5 shows how UDSL presents an architecture for applications to utilize the processing unit on a conceptual level. Applications can compose algorithms through UDSL's programming interface. (If UDSL does not support the targeted algorithms, programmers still need to implement customized kernels.) Once the application is written, there are two ways to specify the targeted architecture. (1) programmers decide the targeted architecture by simply sending a parameter indicating the desired architecture to the compiler before the compilation, or (2) UDSL decides the targeted architecture by checking the available devices and picking up the processing unit that can potentially provide the best performance for the application. Next, the compiler will compile the application with the architecture-specific adaptor defined in UDSL's heterogeneous-aware runtime, and link the corresponding standard library that will utilize the targeted processing unit during the program execution. During the runtime, the application calls the UDSL's API, and the architecture-specific adaptor inside the API will configure device internals and generate missing parameters to make the standard library's API work correctly.

UDSL makes the programming interface agnostic to the underlying architectures. With UDSL's proposed programming interface, programmers can fully concentrate on developing applications and algorithms, while hardware vendors and backend developers can spend time on their architecture-specific implementations and optimizations to provide optimal performance. UDSL's heterogeneous-aware runtime can bridge the gap between the programming interface and backend libraries. Inside the UDSL runtime, architecture-specific adaptors help to fill the missing device configurations and parameters to make underlying APIs work correctly.

4.5 UDSL Implementation

This section describes the dilemmas we faced and the decision we made when designing UDSL. We split this section into two parts, UDSL interface design and UDSL runtime implementation.

4.5.1 UDSL Interface Design

This section explains the challenges of designing a programming interface independent of architectural details and shows an example of UDSL's API implementation.

General-Purpose Interface vs. Domain-Specific Interface

Programming interfaces usually fall into two categories: general-purpose programming interfaces and domain-specific programming interfaces. A general-purpose programming interface, such as C programming language, is featuring economy of expression, modern control flow and data structure capabilities, and a rich set of operators and data types, and it is capable of creating a wide variety of applications [124]. On the other hand, domain-specific programming interfaces are created for specific domains and purposes, containing pre-defined abstractions focusing on a specific class of applications [121, 48].

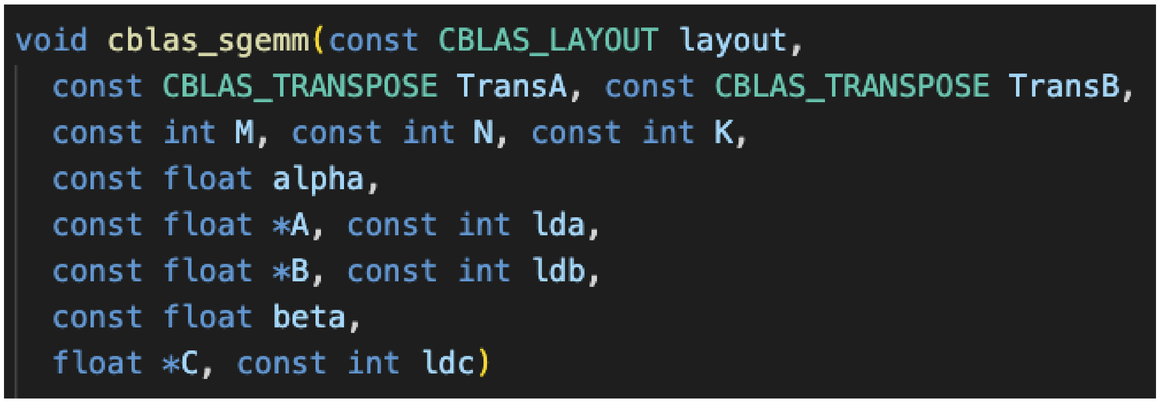
A general-purpose programming interface provides generality and an absence of restrictions to make the language more convenient and effective for various types of applications [124]. It can express the fundamental flow-control constructions required for well-structured programs and make programmers access low-level architectural details, so the behavior of the resulting application is more predictable and controllable for programmers.

Although a domain-specific programming interface has limited expressiveness, it helps programmers to stay away from low-level architectural details and focus on the solutions for domain-specific problems. Also, the pre-defined abstractions and concrete syntax may be clearer and more intuitive to programmers. Because of pre-defined abstractions and concrete syntax, the compilers for domain-specific programming interfaces may optimize the code and perform error detection more efficiently [40].

General-purpose programming interfaces grant programmers more control and flexibility in developing various applications. Still, they tend to be architecture-specific, requiring programmers to take care of architectural details to describe the actual behavior of the application. Since UDSL is designed to mitigate programmers' efforts to reinvent the wheel in different architecture-specific programming interfaces, the design of UDSL's programming interface should be closer to domain-specific that collects operators frequently used in modern applications and also exposed through standard libraries' APIs. In this way, programmers can use the predefined abstraction and concrete syntax to compose applications, without worrying about applications' sustainability and performance.

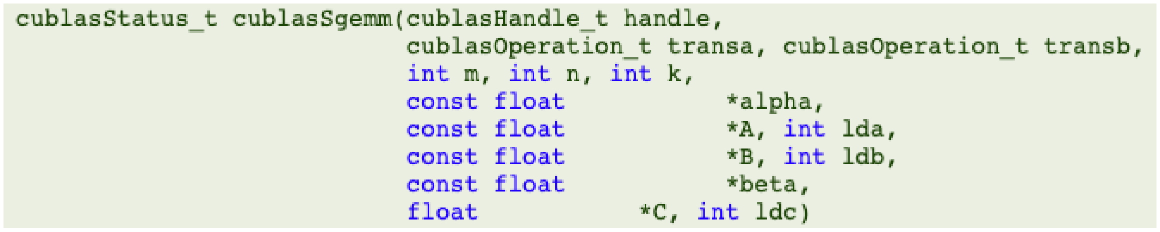
UDSL Domain-Specific Interface Design: GEMM example

This paragraph demonstrates how we design UDSL's domain-specific programming interface, using GEMM APIs as examples. We collected GEMM APIs from well-known libraries or frameworks on different architectures and presented in Figure 4.6- 4.8. Figure 4.6 is the GEMM API collected from OpenBLAS [179], an optimized BLAS library for



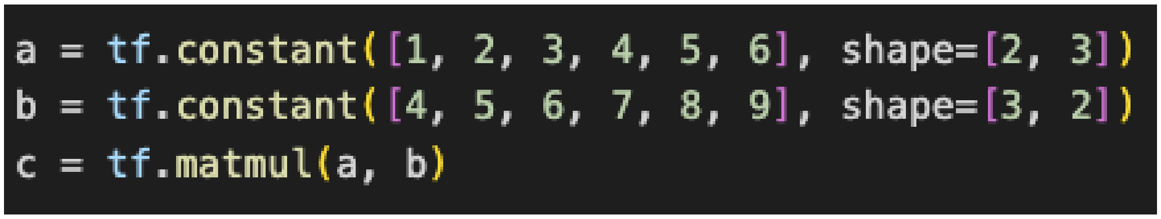
```
void cblas_sgemm(const CBLAS_LAYOUT layout,
  const CBLAS_TRANSPOSE TransA, const CBLAS_TRANSPOSE TransB,
  const int M, const int N, const int K,
  const float alpha,
  const float *A, const int lda,
  const float *B, const int ldb,
  const float beta,
  float *C, const int ldc)
```

Figure 4.6: The GEMM API in OpenBLAS.



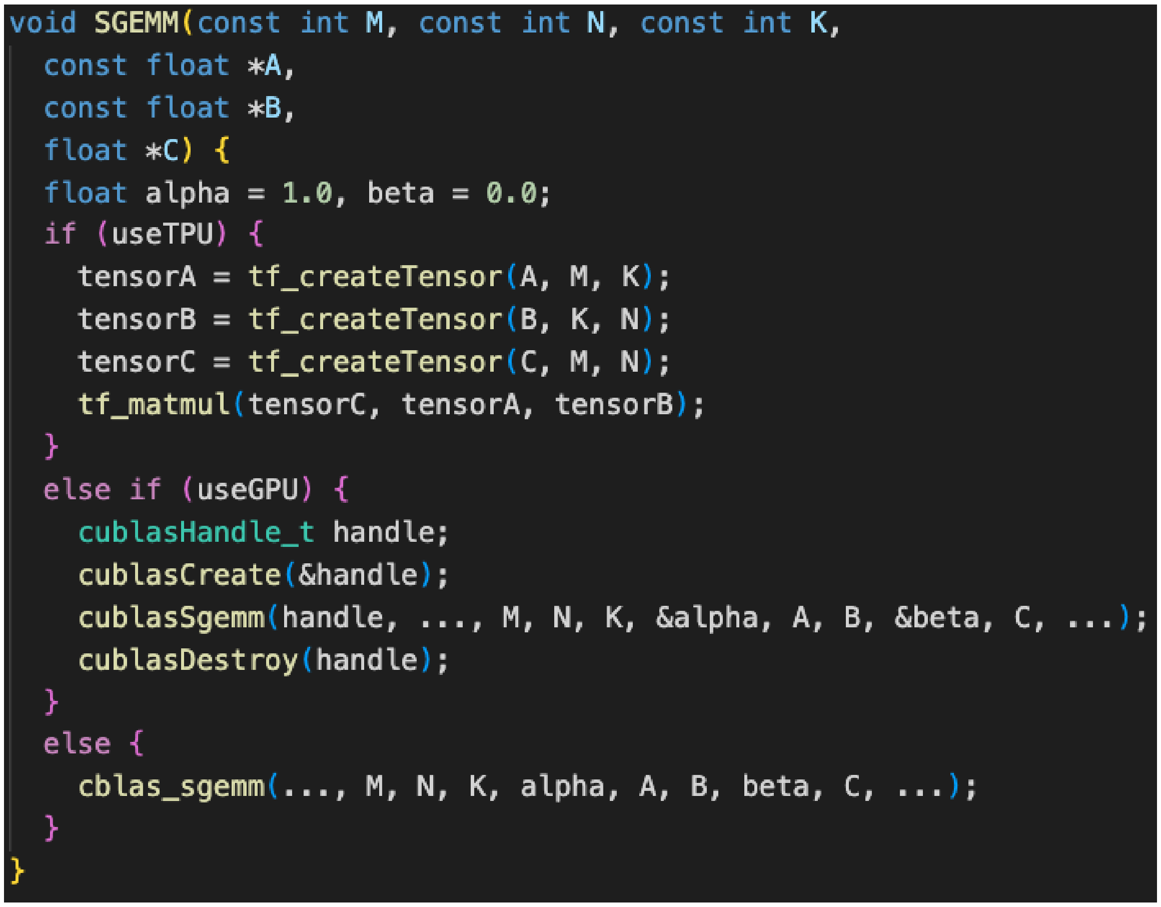
```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float           *C, int ldc)
```

Figure 4.7: The GEMM API in cuBLAS.



```
a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3])
b = tf.constant([4, 5, 6, 7, 8, 9], shape=[3, 2])
c = tf.matmul(a, b)
```

Figure 4.8: The GEMM API in TensorFLow.



```
void SGEMM(const int M, const int N, const int K,
  const float *A,
  const float *B,
  float *C) {
  float alpha = 1.0, beta = 0.0;
  if (useTPU) {
    tensorA = tf_createTensor(A, M, K);
    tensorB = tf_createTensor(B, K, N);
    tensorC = tf_createTensor(C, M, N);
    tf_matmul(tensorC, tensorA, tensorB);
  }
  else if (useGPU) {
    cublasHandle_t handle;
    cublasCreate(&handle);
    cublasSgemm(handle, ..., M, N, K, &alpha, A, B, &beta, C, ...);
    cublasDestroy(handle);
  }
  else {
    cblas_sgemm(..., M, N, K, alpha, A, B, beta, C, ...);
  }
}
```

Figure 4.9: The UDSL GEMM interface and implementation.

CPUs. Figure 4.7 shows the GEMM API from cuBLAS [106] that leverages computation to NVIDIA's GPUs. Invoke the matrix-multiplication kernel on TPU, shown in Figure 4.8, is different from previous examples; programmers need to use TensorFlow [2], a domain-specific framework implemented in Python, to define input tensors by passing the data and the shape as parameters. Then, programmers can simply pass the created tensors into TensorFlow's GEMM API to invoke computation on TPU, and get the result tensor.

Figure 4.9 shows the interface of UDSL's GEMM API. We observed common parameters among those APIs from architecture-specific libraries, even though those APIs also require architectural configurations for underlying processing units. For example, those GEMM APIs all need the dimensions of input and output matrices (e.g. M, N and K), two data objects (e.g., A and B) that indicate the location of the input matrices, and one data object (e.g., C) that points to the location of the output matrix. Then, we can design UDSL's GEMM API based on these common parameters. The resulting UDSL's GEMM API takes M, N and K that describe the shape of matrices and three objects that hold the locations of input or output matrices.

4.5.2 UDSL heterogeneous-compute-aware runtime design

This section describes the implementation of UDSL's heterogeneous-compute-aware runtime and shows an actual implementation in a UDSL's API.

Static Runtime vs. Dynamic Runtime

Similar to the issue brought in Section 3.3.2, the static runtime requiring programmer-directed instructions lacks the flexibility of adjusting tasks once the application is compiled, so the resulting program might miss the opportunity of the best task-allocations on peripherals in a heterogenous computer, leading to performance degradation when system dynamics change. On the other hand, a dynamic runtime, such as the one in ActivePy, requires an appropriate profiling method, the just-in-time compilation, a performance monitor and a migration mechanism. Implementing a dynamic runtime requires programmers' extra efforts to design algorithms and formulas and then implement the ways of communication among different components in a heterogenous computer.

Although we proposed a dynamic runtime in ActivePy (Chapter 3) to support code region identification and generation, performance monitor and dynamic migration during the execution, we choose to implement a static runtime for UDSL because of the following reasons:

(1) the main purpose of using UDSL is to mitigate the difficulty of adopting different hardware, not to find the best opportunity for task allocations.

(2) Even though we want to follow the same logic of ActivePy's design, there are no clear "winning formulas" like we can get advantages from reducing data size in the ISP model.

(3) as underlying architectures become various and complex, the overheads caused by profiling, code generation and communication among devices during the runtime also become significant.

Based on the aforementioned reasons, the UDSL heterogenous-aware runtime is a static runtime that executes the compiled, static instructions on specific hardware architecture. However, the decision can be made by either programmers or UDSL before the UDSL program is compiled, as mentioned in Section 4.4, and the compiler will follow the decision to link the architecture-specific adaptor and corresponding backend libraries to the UDSL program. In this way, UDSL can avoid software overheads caused by profiling, just-in-time compilation and aforementioned dynamic features during the runtime.

UDSL Runtime Implementation: GEMM example

Figure 4.9 also shows the runtime implementation in UDSL's GEMM API. This implementation contains several architecture-specific adapters, and UDSL runtime will use one of the adapters based on the decision made before the compilation and run the corresponding code snippet inside the targeted adaptor. These adapters generate necessary parameters for the targeted architecture and link to architecture-specific library APIs.

For example, the GPU adapter will generate and configure a handler to use the cuBLAS library API, and then the GPU adapter will copy data objects from the system main memory to the device memory to make the API work correctly. Take the TPU adapter for another example. TPU framework requires the input data objects to be `Tesnor` objects. As a result, the TPU adapter will create desired Tensor objects containing the shape of the data and the location of buffer, and pass these Tensor objects to the library API for the computation.

Although the runtime implementation looks simple and requires backend engineers to implement adapters for each architecture to execute operations correctly, the runtime implementation does help to bridge the UDSL programming interface to architecture-specific libraries APIs to guarantee the optimal performance for applications.

4.5.3 UDSL's Collection of operations

We collected operations to the API set of the UDSL programming interface and exposed them to applications. The criteria of these collected operations are: (1) those operations are essential computations in various fields. (2) those operations are frequently used in modern applications and exposed through standard libraries' APIs. (3) those operations can leverage the latest hardware accelerators for better performance. Based on those criteria, we currently included two operations: matrix multiplication and forward convolution.

Matrix multiplication

Matrix multiplication is an essential operation in linear algebra and is also a basic tool in statistics, economics, physics and engineering, and it is frequently used in modern applications, such as AI/ML or scientific applications. This essential operation is included in BLAS libraries and domain-specific frameworks with highly-optimized or hardware-accelerated implementations, and it is exposed as APIs through these libraries and frameworks.

Forward 2D convolution

Convolution is another essential operation applied to statistics, signal processing, image processing, computer vision, physics and engineering. It is frequently used in modern applications, such as AI/ML or physics simulation. Standard libraries or frameworks, such as cuDNN [36], cutlass [105] and TensorFlow [2], have included this operation and exposed it through APIs with highly optimized or hardware-accelerated implementations.

4.6 Experimental methodology

We evaluated the proposed UDSL by replacing the baseline applications' customized kernels with UDSL's APIs representing the same algorithm. This section describes the system configurations and applications we collected and modified.

4.6.1 Experimental platform

We used an 8-core AMD RyZen 3700X processor with a clock rate of up to 4.4 GHz. We installed Ubuntu 16.04 (Linux kernel version 4.15), CUDA 11.0 and cuBLAS 11.2.0. The host machine also contains a heterogeneous hardware accelerator, an NVIDIA RTX 2080 GPU with 8 GB of device memory, to demonstrate the easy adoption provided by UDSL.

4.6.2 Applications

Table 4.1 presents the workloads we implemented with UDSL to evaluate the effectiveness of UDSL. The set of applications falls into two categories: data mining and physics simulation. The criteria for application selection are: (1) the application provides highly efficient customized kernels in its open-sourced implementation (mentioned in Table 4.2), which is the baseline implementation. And, (2) the customized kernel in the targeted application can be replaced by UDSL's API. Besides the criteria of application selection, we exhaustively explored the size and shape of the input dataset for each application that maximizes the computation throughput.

Wordload Name	**Category**	**UDSL API**	**Description of Targeted Compute Kernel**
Hotspot (Hotspot)	Physics Simulation	conv	L2 distances for each point to other points
K-Nearest Neighbor (KNN)	Data Mining	gemm	Averaged temperature of neighbor points

Table 4.1: Workloads, UDSL APIs used and descriptions on their targeted compute kernels.

Wordload Name	**Category**	**Source of Baseline**
Hotspot (Hotspot)	Physics Simulation	Rodinia [34]
K-Nearest Neighbor (KNN)	Data Mining	Rodinia [34], knn-CUDA [52, 158]

Table 4.2: Workloads and the sources of their baseline implementations.

K-nearest neighbors (K-NN)

K-NN is a popular clustering algorithm that forms the kernel of many data-mining applications. Inside the K-NN algorithm, there is one step the algorithm calculates the L2 distances for each point to its neighbor points to find out k-nearest neighbors in the rest of the steps, and this L2 distances calculation can be done by matrix multiplication [52] We implemented the baseline version k-NN application based on knn-cuda [158]. The baseline version uses an optimized and customized GPU kernel to calculate the L2 distances, and we replaced the customized L2 distance compute kernel with UDSL's GEMM API in the UDSL version for comparison.

Hotspot (HS)

HotSpot [63] is a thermal simulation tool used for estimating processor temperature based on an architectural floor plan and simulated power measurements. The baseline implementation that includes the 2D transient thermal simulation kernel of HotSpot is based on the Rodinia benchmark [34]. The 2D transient thermal simulation kernel iteratively solves a series of differential equations for block temperatures, and each cell in the grid represents the average temperature value of the neighbor area of the chip. Since getting the average temperature value of the neighbor area of the chip for each output cell perfectly maps to the calculation in the convolution operator, we used UDSL convolution API to implement this step and replace the original customized kernel.

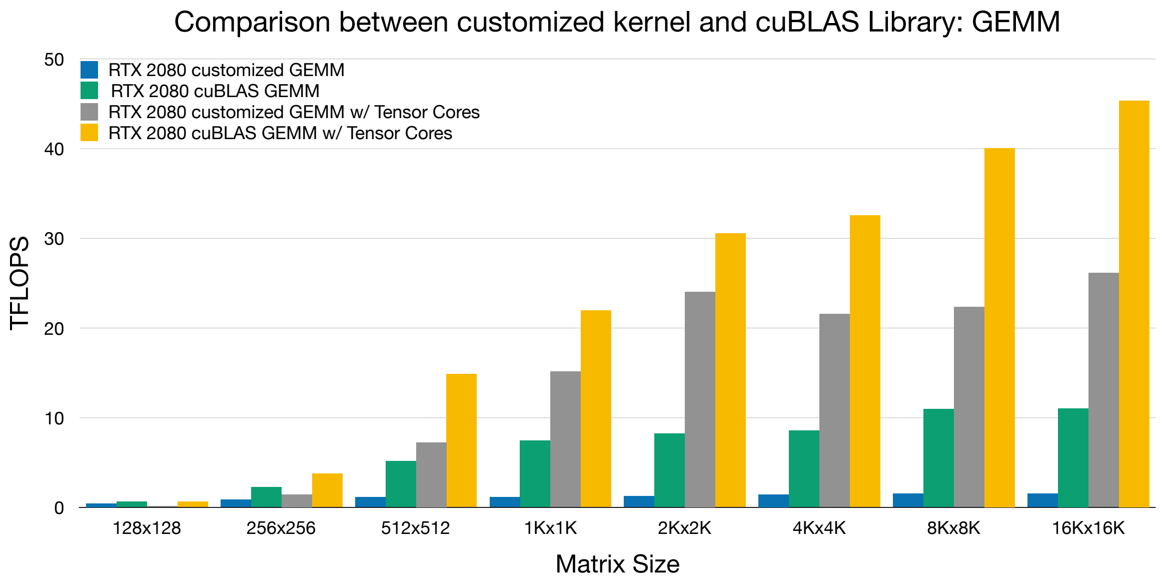


Figure 4.10: The matrix-multiplication performance of the customized kernel and cuBLAS GEMM APIs with only GPU cores, the customized kernel and cuBLAS GEMM APIs with TCUs enabled among different shapes of input/output matrices shape.

4.7 Results

This section summarizes our evaluation of UDSL by running microbenchmarks and real-world applications.

4.7.1 Microbenchmarks

We created a set of microbenchmarks containing matrix multiplication and forward convolution on an NVIDIA RTX 2080 and the system mentioned in Section 4.6.1.

Matrix Multiplication

Figure 4.10 extends the example mentioned in Section 4.3.2. In Figure 4.10, we compared four different hardware-accelerated implemenetations of matrix multiplication on

NVIDIA RTX 2080 GPU. The first two implementations are the customized kernel that uses GPU CUDA cores (vector processing units), as our baseline, and the other customized kernel that leverages TCUs (matrix processing units). Like the example in Section 4.3.2, these implementations are collected from NVIDIA's official CUDA examples [107]. The other two implementations use cuBLAS GEMM APIs to replace those customized kernels. Similar to customized kernel implementations, in these two implementations, one only uses GPU cores, and the other enables TCUs to accelerate the computation. The x-axis indicates the shape of the input matrices and the output matrix, and the y-axis shows the performance numbers in TFLOPS (Tera-floating-point-operation per second).

We tested different shapes from 128×128 to 16384×16384 32-bit floating-point matrices, which is the maximum size that the device's memory can contain. With the baseline configuration and using 16384×16384 input matrices, the matrix-multiplication computation achieves 1.51 TFLOPS. Compared to the baseline, cuBLAS GEMM function without using TCUs is able to achieve 11.07 TFLOPS, 7.33× speedup. These results demonstrate that using APIs from standard libraries guarantees UDSL the best performance provided by the hardware accelerators.

Next, we rewrote a new customized kernel for adopting TCUs. By feeding 16384×16384 input matrices to the new customized kernel, it achieves 26.1 TFLOPS, 17.3× speedup compared to the baseline configuration. However, programmers only need to change a few lines of code to make cuBLAS GEMM API collaborate with TCUs and easily achieve 45.33 TFLOPS, 30× faster than the baseline, with the same input matrices.

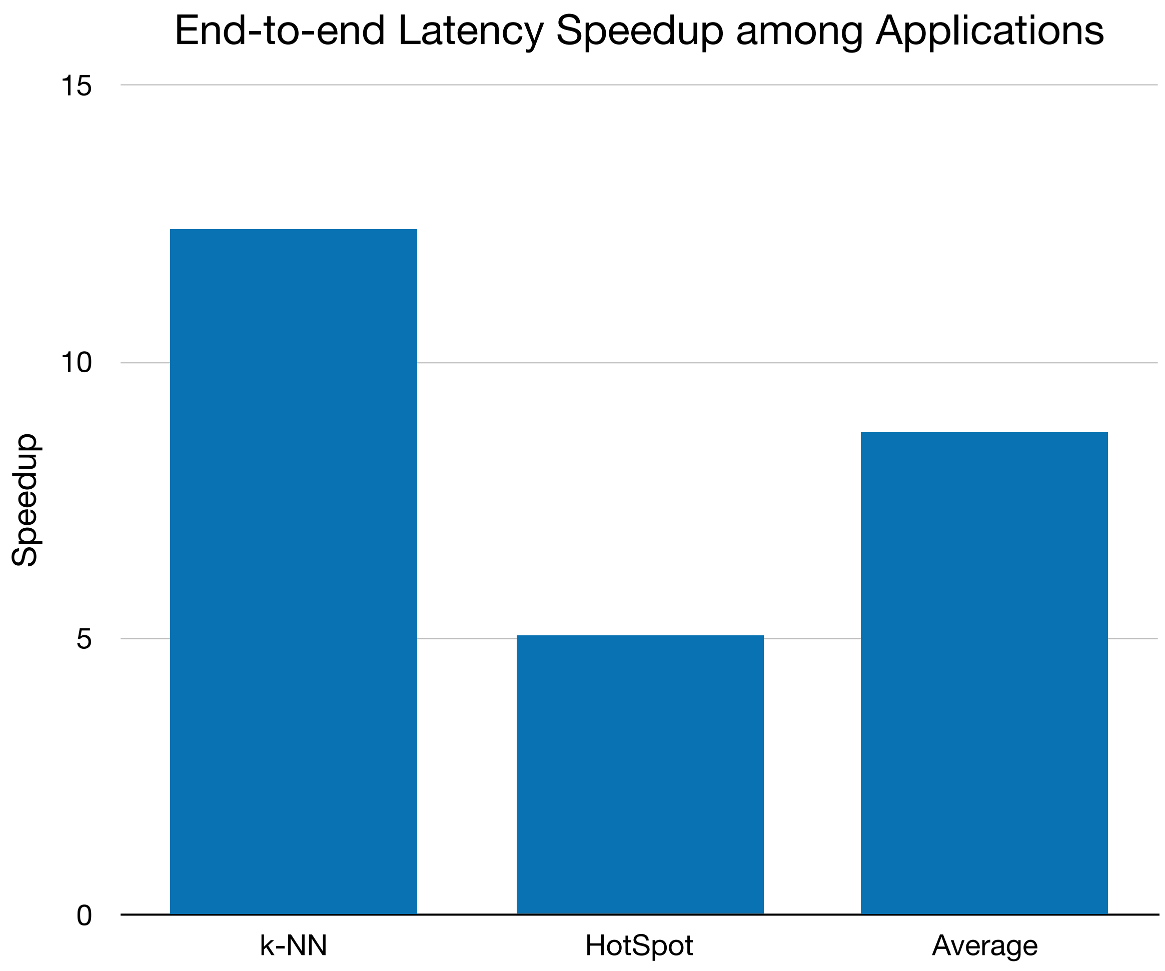


Figure 4.11: The end-to-end latency speedup of applications using UDSL

This comparison shows that using standard libraries' APIs helps applications to easily adopt different architectures, and UDSL's implementation should include the same idea.

4.7.2 The speedup of end-to-end latency of applications using UDSL

Figure 4.11 shows the speedup of end-to-end latency of running applications using UDSL. The horizontal axis shows the application we evaluated the effectiveness of UDSL, and the vertical axis is the end-to-end latency speedup of running application using UDSL compared to the baseline, where applications are implemented with customized kernels, mentioned in Section 4.6.2. Results in Figure 4.11 show that using UDSL's APIs for compute kernels can speed up the application by 8.72× on average. At the same time, thanks to UDSL's programming interface, programmers can simply replace customized kernels with UDSL's API implemented the same algorithms. This result demonstrates two things: (1) UDSL's heterogeneous-aware runtime utilizing highly-optimized standard libraries successfully helps applications achieve optimal performance. (2) UDSL's application-specific programming interface helps programmers to focus on developing applications and algorithms, without worrying about implementing compute kernels and tuning the performance of applications. This result also shows that UDSL successfully resolves problems introduced by architecture-specific programming interfaces, mentioned in Section 4.3.

4.8 Related works

In addition to the architecture-independent programming interface and the heterogeneous-aware runtime, this section describes a few UDSL-related works.

Redesign algorithms for different processing models

Emerging demands of applications have pushed processing models from the scalar processing model to the matrix processing model. However, matrix processing units not only can be applied to matrix computation. If programmers put data layout correctly, those matrix processing units can also be applied to vector computation. Some works have explored opportunities of applying matrix processing units to redesigned algorithms, such as using TCUs for scan and reduction [41], or using TCUs for join database operation [61], and those works have proved using TCUs on redesigned algorithms improves performance. Since some UDSL's APIs also leverage matrix processing units, programmers using UDSL can have a chance to rethink and redesign the algorithms and apply them to matrix processing units through UDSL's APIs for better performance.

Benchmark suites

Besides problems mentioned in Section 4.3, current benchmark suites enlighten us on designing the UDSL programming interface, its heterogeneous runtime and the criteria for selecting workloads.

AxBench [172] is a multiplatform benchmark suite for approximate computing. This benchmark suite contains diverse applications for CPUs, GPUs, and other hardware architectures, and gives us insights into designing an architecture-independent programming interface for applications.

On the other hand, Rodinia benchmark suite [34] is a famous GPU benchmark suite that collects various fields of GPU-accelerated applications. These applications are implemented in optimized, customized kernels, but they still cannot outperform standard libraries APIs, demonstrated in Section 4.3.2. This observation tells us we should stick with standard libraries' APIs when implementing UDSL's heterogeneous runtime.

4.9 Conclusion

This chapter introduces a programming interface called UDSL and describes prototype UDSL implementation. UDSL successfully addresses problems, such as inconvertible code snippets and under-utilized computational capabilities, caused by architecture-specific programming interfaces by providing the following implementations: (1) UDSL's application-specific programming interface, containing a set of operations, decouples the application development from architecture-specific implementations. (2) UDSL's heterogeneous-aware runtime bridges UDSL's frontend and highly optimized, architecture-specific backend libraries to fully utilize underlying processing units. With these two implementations, UDSL makes applications agnostic to architectural details, so that UDSL applications do not require programmers to rewrite the code to adopt a different architecture, but UDSL applications still guarantee optimal performance. Through the prototype UDSL implementation, we measured that UDSL applications achieved an average 8.72× speedup compared to applications implemented in optimized, customized kernels.

Chapter 5

Conclusions

The evolution of technologies is always trying to fulfill the demand of real-world applications. We have seen heterogeneous accelerators, memory technologies and computational storage device scale up the performance of applications. However, since we are hesitating to change the conventional way of communication, the efficiency and programmability of interfaces are falling behind the evolution. This thesis demonstrates the power of changing the interfaces through the following works:

Firstly, NDS, mentioned in Chapter 2, explores the idea of making the storage interface multi-dimensional and allows applications to have their own view of datasets. The multi-dimensional storage interface helps to reduce the overhead of data-object transformation and gauge the demands of applications from storage devices. With the new multi-dimensional storage interface, space-translation layer and building blocks, NDS delivers $5.73\times$ speedup compared to the conventional storage interface and SSDs.

Secondly, ActivePy, mentioned in Chapter 3, provides a programming interface that intelligently identifies and offloads tasks to the computational storage devices without programmers' intervention, Besides the interface, ActivePy's runtime can dynamically monitor and migrate tasks from busy CSDs and resume on the host computer to avoid performance degradation. ActivePy achieves 1.33× speedup, as good as the conventional static programming interfaces. However when the CSD only has 10% computational resources left, ActivePy is able to achieve a performance number as close as the no-task-offloaded baseline and 2.82× speedup compared to the conventional static programming interfaces without the migration mechanism.

Lastly, UDSL, mentioned in Chapter 4, presents an architecture-independent programming interface that makes applications agnostic to the architectural details of underlying processing units. UDSL's programming interface helps applications easily to adopt different architectures, while UDSL's heterogeneous-aware runtime guarantees the optimal performance of each architecture. UDSL achieves 8.72× compared to applications implemented with customized kernels, and UDSL is able to save programmers' efforts in rewriting architecture-specific code snippets for switching to another architecture.

Besides these proposed works, we believe there are more interfaces in the system stack of heterogeneous computers needed to be revised. The revised interfaces should reflect the demands of modern workloads and unleash the power of innovative system components.

Bibliography

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 967–980, New York, NY, USA, 2008. Association for Computing Machinery.

[2] 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, Dandelion 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. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

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

[4] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active Disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 81–91, New York, NY, USA, 1998. ACM.

[5] Ian F. Adams, John Keys, and Michael P. Mesnier. Respecting the block interface – computational storage using virtual objects. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[6] Advanced Micro Devices, Inc. FirePro DirectGMA Technical Overview, 2014.

[7] Francesc Alted, Martin Durant, Stephan Hoyer, John Kirkham, Alistair Miles, Mamy Ratsimbazafy, Matthew Rocklin, Vincent Schut, Anthony Scopatz, and Prakhar Goel. Zarr.

[8] Amber Huffman. NVM Express Revision 1.1, 2012.

[9] Roberto Ammendola, Massimo Bernaschi, Andrea Biagioni, Mauro Bisson, Massimiliano Fatica, Ottorino Frezza, Francesca Lo Cicero, Alessandro Lonardo, Enrico Mastrostefano, Pier Stanislao Paolucci, Davide Rossetti, Francesco Simula, Laura Tosoratto, and Piero Vicini. Gpu peer-to-peer techniques applied to a cluster interconnect. In *IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 806–815, 2013.

[10] Apache Software Foundation. Apache Avro(TM) 1.10.2 Documentation.

[11] Apache Software Foundation. Apache Parquet.

[12] Apache Software Foundation. The smallest, fastest columnar storage for Hadoop workloads.

[13] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 725–737, New York, NY, USA, 2015. Association for Computing Machinery.

[14] Armin Rigo and Maciej Fija?kowski and Carl Friedrich Bolz and Antonio Cuni and Benjamin Peterson and Alex Gaynor and H?kan Ard? and Holger Krekel and Samuele Pedroni. PyPy.

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

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

[17] Antonio Barbalace and Jaeyoung Do. Computational storage: Where are we today? In *CIDR*, 2021.

[18] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[19] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, page 368–377, New York, NY, USA, 2008. Association for Computing Machinery.

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

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

[22] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31, 2011.

[23] Fabian Beuke. Githut 2.0–a small place to discover languages in github, 2020.

[24] Ray Bittner, Erik Ruf, and Alessandro Forin. Direct GPU/FPGA Communication Via PCI Express. *Cluster Computing*, 17(2):339–348, June 2014.

[25] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The Linux open-channel SSD subsystem. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, pages 359–373, USA, 2017. USENIX Association.

[26] S. Boboila, Youngjae Kim, S.S. Vazhkudai, P. Desnoyers, and G.M. Shipman. Active flash: Out-of-core data analytics on flash storage. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12, April 2012.

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

[28] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, page 233–244, New York, NY, USA, 2009. Association for Computing Machinery.

[29] Stuart Byma, Sam Whitlock, Laura Flueratoru, Ethan Tseng, Christos Kozyrakis, Edouard Bugnion, and James Larus. Persona: A high-performance bioinformatics framework. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 153–165, Santa Clara, CA, July 2017. USENIX Association.

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

[31] J. Carter, W. Hsieh, L. Stoller, M. Swanson, Lixin Zhang, E. Brunvand, A. Davis, Chen-Chi Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse:

building a smarter memory controller. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pages 70–79, 1999.

[32] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. 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, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society.

[33] C Cecka. Pro Tip: cuBLAS strided batched matrix multiply, 2018.

[34] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

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

[36] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[37] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.*, 2(OOPSLA):123:1–123:30, October 2018.

[38] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. Association for Computing Machinery.

[39] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. Association for Computing Machinery.

[40] Krzysztof Czarnecki. Overview of generative software development. In *International workshop on unconventional programming paradigms*, pages 326–341. Springer, 2004.

[41] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*, pages 46–57, 2019.

[42] Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. Efficient and thread-safe objects for dynamically-typed languages. *SIGPLAN Not.*, 51(10):642–659, 2016.

[43] Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. Parallelization of dynamic languages: Synchronizing built-in collections. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.

[44] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart SSDs: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[45] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. Programmable Solid-State Storage in Future Cloud Datacenters. *Commun. ACM*, 62:54–62, 2019.

[46] 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. The architecture of the diva processing-in-memory chip. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 14–25, 2002.

[47] Marc E. Fiuczynski, Richard P. Martin, Tsutomu Owa, and Brian N. Bershad. 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, pages 7–12, 1998.

[48] Martin Fowler. *Domain-specific languages.* Pearson Education, 2010.

[49] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478. ACM, 2009.

[50] J. J. Galvez, K. Senthil, and L. Kale. Charmpy: A python parallel programming model. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 423–433, Sep. 2018.

[51] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 113–124. IEEE, 2015.

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

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

[54] Serban Giuroiu. CUDA k-means data clustering, 2012.

[55] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. SparTen: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 151?V165, New York, NY, USA, 2019. Association for Computing Machinery.

[56] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 24–33, New York, NY, USA, 2009. Association for Computing Machinery.

[57] 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. Biscuit: A framework for near-data processing of big data workloads. *SIGARCH Comput. Archit. News*, 44(3):153–165, June 2016.

[58] N. Hajinazar, P. Patel, M. Patel, K. Kanellopoulos, S. Ghose, R. Ausavarungnirun, G. F. Oliveira, J. Appavoo, V. Seshadri, and O. Mutlu. The virtual block interface: A flexible alternative to the conventional virtual memory framework. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1050–1063, 2020.

[59] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. ExTensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–333, 2019.

[60] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W. Keckler. Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 204–216, Piscataway, NJ, USA, 2016. IEEE Press.

[61] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. Tcudb: Accelerating database with tensor processors. *arXiv preprint arXiv:2112.07552*, 2021.

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

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

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

[65] J. Jenkins, J. Dinan, P. Balaji, N.F. Samatova, and R. Thakur. Enabling fast, non-contiguous gpu data movement in hybrid mpi+gpu environments. In *2012 IEEE International Conference on Cluster Computing*, CLUSTER, pages 468–476, Sept 2012.

[66] 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. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[67] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. BlueDBM: An appliance for big data analytics. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 1–13, New York, NY, USA, 2015. ACM.

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

[69] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a true direct-access file system with devfs. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, Oakland, CA, 2018. USENIX Association.

[70] Shinpei Kato, Jason Aumiller, and Scott Brandt. Zero-copy I/O processing for low-latency GPU computing. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, pages 170–178, 2013.

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

[72] Kevin Modzelewski. Introducing Pyston: an upcoming, JIT-based Python implementation, 2014.

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

[74] Orhan Kislal, Jagadish Kotra, Xulong Tang, Mahmut Taylan Kandemir, and Myoungsoo Jung. Enhancing computation-to-core assignment with physical location information. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 312–327, New York, NY, USA, 2018. ACM.

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

[76] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

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

[78] Gunjae Koo, Kiran Kumar Matam, Te I, Hema Venkata Krishna Giri Narra, Jing Li, Steven Swanson, Hung-Wei Tseng, and Murali Annavaram. Summarizer: Trading bandwidth with computing near storage. In *50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 2017, 2017.

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

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

[81] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6. ACM, 2015.

[82] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[83] Michael A. Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. Input Responsiveness: Using Canary Inputs to Dynamically Steer Approximation. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 161–176, 2016.

[84] Sangwon Lee, Gyuyoung Park, and Myoungsoo Jung. TensorPRAM: Designing a scalable heterogeneous deep learning accelerator with byte-addressable prams. In Anirudh Badam and Vijay Chidambaram, editors, *12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020, July 13-14, 2020*. USENIX Association, 2020.

[85] Levy and Lipman. Virtual memory management in the VAX/VMS Operating System. *Computer*, 15(3):35–41, 1982.

[86] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou. Optimizing memory efficiency for deep convolutional neural networks on GPUs. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 633–644, 2016.

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

[88] Shuo Li, Nong Xiao, Peng Wang, Guangyu Sun, Xiaoyang Wang, Yiran Chen, Hai Helen Li, Jason Cong, and Tao Zhang. RC-NVM: Dual-addressing non-volatile memory architecture supporting both row and column memory accesses. *IEEE Transactions on Computers*, 68(2):239–254, 2019.

[89] Wei-Kang Liao. Parallel k-means data clustering, 2003.

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

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

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

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

[94] Deepak Majeti, Kuldeep S. Meel, Rajkishore Barik, and Vivek Sarkar. Automatic data layout generation and kernel mapping for CPU+GPU architectures. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 240–250, New York, NY, USA, 2016. Association for Computing Machinery.

[95] Vadim Markovtsev and Máximo Cuadros. src-d/kmcuda: 6.0.0-1, February 2017.

[96] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, , and Murali Annavaram. GraphSSD: Graph Semantics Aware SSD. In *46th International Symposium on Computer Architecture*, ISCA 2019, 2019.

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

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

[99] Michael M McKerns, Leif Strand, Tim Sullivan, Alta Fang, and Michael AG Aivazis. Building a framework for predictive science. *arXiv preprint arXiv:1202.1056*, 2012.

[100] MM McKerns and M Aivazis. Pathos: a framework for heterogeneous computing. 2010.

[101] Microchip Technology Inc. Flashtec NVMe Controllers, 2020.

[102] Micron. MT29F32G08CBADAWP Datasheet, 2015.

[103] Rory Mitchell and Eibe Frank. Accelerating the XGBoost algorithm using GPU computing. *PeerJ Computer Science*, 3:e127, July 2017.

[104] National Institute of Standards and Technology. Cryptographic Standards and Guidelines, 2021.

[105] NVIDIA. cutlass.

[106] NVIDIA. cuBLAS, 2019.

[107] NVIDIA. CUDA Samples, 2022.

[108] NVIDIA. CUDA Toolkit Documentation, 2022.

[109] NVIDIA. Tensor Cores, 2022.

[110] NVIDIA Corporation. Developing a Linux Kernel Module Using RDMA for GPUDirect, 2014.

[111] NVM Express, Inc. Nvm express explained, 2013.

[112] NVM Express, Inc. NVM Express(R) Moves Into The Future, 2016.

[113] L. Oden and H. Froning. GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters. In *2013 IEEE International Conference on Cluster Computing*, CLUSTER ' 13, pages 1–8, Sept 2013.

[114] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[115] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

[116] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. Intelligent ram (iram): chips that remember and compute. In *Solid-State Circuits Conference, 1997. Digest of Technical Papers. 43rd ISSCC., 1997 IEEE International*, pages 224–225, Feb 1997.

[117] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for NIC-Accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, Carlsbad, CA, October 2018. USENIX Association.

[118] PMC-Sierra. Flashtec NVMe Controllers, 2014.

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

[120] Srjan Popić, Dražen Pezer, Bojan Mrazovac, and Nikola Teslić. Performance evaluation of using protocol buffers in the internet of things communication. In *2016 International Conference on Smart Systems and Technologies (SST)*, pages 261–265, 2016.

[121] Ivens Portugal, Paulo Alencar, and Donald Cowan. A survey on domain-specific languages for machine learning in big data. *arXiv preprint arXiv:1602.07637*, 2016.

[122] S. Potluri, H. Wang, D. Bureddy, A.K. Singh, C. Rosales, and D.K. Panda. Optimizing mpi communication on multi-gpu systems using cuda inter-process communication. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, IPDPSW ' 12, pages 1848–1857, May 2012.

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

[124] Dennis M Ritchie, Stephen C Johnson, ME Lesk, BW Kernighan, et al. The c programming language. *Bell Sys. Tech. J*, 57(6):1991–2019, 1978.

[125] Robert Kern. line_profiler and kernprof, 2017.

[126] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 130 – 136, 2015.

[127] Horacio Hoyos Rodriguez and Beatriz Sanchez Piña. JSOI: A JSON-based interchange format for efficient model management. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 259–266, 2019.

[128] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 49–68. Association for Computing Machinery, 2013.

[129] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing In-Storage computing system for emerging High-Performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, Renton, WA, July 2019. USENIX Association.

[130] Alex Rubinsteyn, Eric Hielscher, Nathaniel Weinman, and Dennis Shasha. Parakeet: A just-in-time parallel accelerator for Python. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2012. USENIX.

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

[132] Stephan Saalfeld. N5.

[133] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, Broomfield, CO, October 2014. USENIX Association.

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

[135] Z. Shao, N. Chang, and N. Dutt. PTL: PCM translation layer. In *2012 IEEE Computer Society Annual Symposium on VLSI*, pages 380–385, 2012.

[136] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka. Tensor contractions with extended BLAS kernels on CPU and GPU. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 193–202, 2016.

[137] Mustafa Shihab, Karl Taht, and Myoungsoo Jung. Gpudrive: Reconsidering storage accesses for gpu acceleration. In *Workshop on Architectures and Systems for Big Data*, 2014.

[138] Simon, Raina Mason, Tom Crick, James H. Davenport, and Ellen Murphy. Language choice in introductory programming courses at australasian and uk universities. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 852–857. ACM, 2018.

[139] SNIA. What is computational storage?, 2020.

[140] SNIA Computational Storage Technical Working Group. Computational Storage Architecture and Programming Model Version 0.8 Revision 1, 2021.

[141] Edgar Solomonik, Devin Matthews, Jeff R Hammond, John F Stanton, and James Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, 2014.

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

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

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

[145] J.A. Stuart and J.D. Owens. Multi-GPU MapReduce on GPU Clusters. In *2011 IEEE International Parallel Distributed Processing Symposium*, IPDPS ' 11, pages 1068–1079, May 2011.

[146] Qian Sun. Parallel implementation of Bellman Ford algorithm, 2018.

[147] Fuad Tabba. Adding concurrency in python using a commercial processor's hardware transactional memory support. *SIGARCH Comput. Archit. News*, 38(5):12–19, April 2010.

[148] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M Badia, Jordi Torres, Toni Cortes, and Jes?ys Labarta. Pycompss: Parallel computational workflows in python. *The International Journal of High Performance Computing Applications*, 31(1):66–82, 2017.

[149] The Linux Foundation. Open Neural Network Exchange – The open standard for machine learning interoperability.

[150] Devesh Tiwari, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Simona Boboila, and Peter J. Desnoyers. 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, 2012.

[151] J. Torrellas. Flexram: Toward an advanced intelligent memory system: A retrospective paper. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 3–4, Sept 2012.

[152] Transaction Processing Performance Council (TPC). TPC BENCHMARK H (Decision Support) Standard Specificatio, 2021.

[153] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schüpbach, and Bernard Metzler. Albis: High-performance file format for big data systems. In *USENIX Annual Technical Conference*, 2018.

[154] Po-An Tsai, Changping Chen, and Daniel Sanchez. Adaptive scheduling for systems with asymmetric memory hierarchies. In *51th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 2018, 2018.

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

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

[157] Kenton Varda. Protocol buffers: Google's data interchange format. Technical report, 2008.

[158] Michel Barlaud Vincent Garcia, Éric Debreuve. kNN-CUDA, 2018.

[159] Hao Wang, S. Potluri, D. Bureddy, C. Rosales, and D.K. Panda. GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2595–2605, Oct 2014.

[160] Hao Wang, Sreeram Potluri, Miao Luo, AshishKumar Singh, Sayantan Sur, and DhabaleswarK. Panda. MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. *Computer Science - Research and Development*, 26(3-4):257–266, 2011.

[161] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 2016.

[162] P. Willmann, H. Kim, S. Rixner, and V.S. Pai. An efficient programmable 10 gigabit ethernet network interface card. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 96–107, 2005.

[163] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proc. VLDB Endow.*, 7(11):963–974, July 2014.

[164] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an unwritten contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[165] Carl Yang, Aydin Buluç, and John D. Owens. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *CoRR*, abs/1908.01407, 2019.

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

[167] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 3, USA, 2012. USENIX Association.

[168] L. T. Yang, Xiaosong Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 40–40, 2005.

[169] Yannis Papakonstantinou Yanqin Jin, Hung-Wei Tseng and Steven Swanson. Kaml: A flexible, high-performance key-value ssd. In *High Performance Computer Architecture (HPCA)*, 2017.

[170] Yong Yao and Johannes Gehrke. The Cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, September 2002.

[171] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. AxBench: A multiplatform benchmark suite for approximate computing. *IEEE Design Test*, 34(2):60–68, April 2017.

[172] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. Axbench: A multiplatform benchmark suite for approximate computing. *IEEE Design & Test*, 34(2):60–68, 2016.

[173] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The yin and yang of processing data warehousing queries on GPU devices. *PVLDB*, 6(10):817–828, 2013.

[174] Zach Zimmerman. MSplitGEMM: Large matrix multiplication in CUDA, 2016.

[175] J. Zhang, G. Park, D. Donofrio, J. Shalf, and M. Jung. DRAM-Less: Hardware acceleration of data processing with new memory. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 287–302, 2020.

[176] Jie Zhang, David Donofrio, John Shalf, Mahmut Kandemir, and Myoungsoo Jung. NVMMU: A non-volatile memory management unit for heterogeneous GPU-SSD architectures. In *The 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT 2015, 2015.

[177] Jie Zhang and Myoungsoo Jung. Flashabacus: A Self-governing Flash-based Accelerator for Low-power Systems. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 15:1–15:15, New York, NY, USA, 2018. ACM.

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

[179] Martin Kroeker Zhang Xianyi. OpenBLAS.

[180] W. Zhao. Dimacs10/ak2010, 2010.

[181] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 14–23, 2009.

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

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

[184] Marcin Zukowski, Niels Nes, and Peter Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Proceedings of the 4th International Workshop on Data Management on New Hardware*, DaMoN '08, pages 47–54, New York, NY, USA, 2008. Association for Computing Machinery.