UNIVERSITY OF CALIFORNIA
RIVERSIDE

Efficient Accelerator-Rich Computers for Future Applications

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

Doctor of Philosophy

in

Computer Science

by

Yu-Ching Hu

September 2024

Dissertation Committee:

Dr. Hung-Wei Tseng, Chairperson
Dr. Daniel Wong
Dr. Elaheh Sadredini
Dr. Nael Abu-Ghazaleh

The Dissertation of Yu-Ching Hu is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to thank Professor Hung-Wei Tseng for his guidance as my thesis advisor and chairperson of my committee. Without his support and advice, I would not be here today. I would also thank Professor Daniel Wong, Professor Elaheh Sadredini and Professor Nael Abu-Ghazaleh for their insightful feedbacks and inputs as my committee members.

I would like to acknowledge Dr. Yuliang Li, who offered theory and technical support throughout my research journey. His insights helped me conduct more interesting research topics and build useful skills.

I am grateful for all of my friends, collaborators, and colleagues from ESCAL group for the highs and lows we experienced together.

Lastly, I would like to express my gratitude to my wife Yi-Zhen Tsai and my parents for their mental and financial support as I pursued a Ph.D. in computer science as well as master's degrees in material science.

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

ABSTRACT OF THE DISSERTATION

Efficient Accelerator-Rich Computers for Future Applications

by

Yu-Ching Hu

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2024
Dr. Hung-Wei Tseng, Chairperson

With the advancement of processor technology, numerous hardware accelerators beyond CPUs and GPUs are emerging to meet the rapid growth in computation demands. Particularly, the demand for AI and ML applications is outpacing the improvements in general-purpose hardware, prompting researchers to integrate hardware accelerators into architectural designs. This raises a fundamental research question: Are we fully exploiting these AI/ML hardware accelerators?

This dissertation addresses this question from multiple perspectives. Firstly, have we used approximate hardware efficiently and effectively? Optimal performance requires that the system supplies data smoothly to powerful computing units. Secondly, the portability of hardware accelerators. While designed for compute-intensive AI/ML workloads, can other domains benefit from these accelerators? Lastly, do we need more accelerators, or are current ones sufficient for evolving AI-assisted applications?

To answer the first question, I proposed Varifocal Storage (VS), an architecture that reduces unnecessary data via in-storage processing, mitigating data traffic within in-

terconnects and minimizing data transformation overhead. By dynamically adjusting data resolutions, VS addresses the demands for performance, flexibility, cost, and quality, necessitating a hardware/software co-design within the approximate computing framework.

For the second question, I proposed TCUDB, a relational database query engine leveraging Tensor Core to significantly accelerate SQL query processing, achieving orders of magnitude speedup even for non-AI/ML queries. TCUDB revisits application algorithms and data layouts for emerging hardware accelerators, demonstrating versatility across various analytic queries and use cases including matrix multiplication, entity matching, and graph applications.

Finally, driven by the growth in AI-based personal assistant applications and the shift from traditional PCs to mobile devices, I proposed the Personal Assistant Multi-device Machine Learning Benchmark (PAMLB) to address complexities in data processing pipelines. Existing benchmarks like Rodinia and TPC-H fail to capture the real-world experience of AI-assisted applications, which heavily rely on small user devices and data center interactions. PAMLB aims to develop comprehensive workloads to optimize/deploy modules on different devices for these advanced applications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The insatiable demand for computing power from increasingly complex AI models [183, 27, 38, 184] has exceeded the capabilities of traditional technology, despite the tremendous gains achieved by CPU and GPU vendors. This harsh reality highlights the urgent need for specialized hardware accelerators made expressly to effectively handle the particular processing demands of AI workloads.

Figure 1.1 illustrates the explosive growth in demand for computing power driven by the rise of artificial intelligence models over the past decade. While the performance of traditional CPUs and GPUs has steadily improved, with roughly 7× and 14× speedups respectively from 2014 to 2024, the demand for compute resources by AI models has skyrocketed exponentially, particularly between 2014 and 2018.

As we enter an era of AI proliferation across various domains, it becomes crucial to leverage hardware accelerators optimized for computations not just limited to AI. These heterogeneous architectures have the potential to bridge the widening gap between the de-

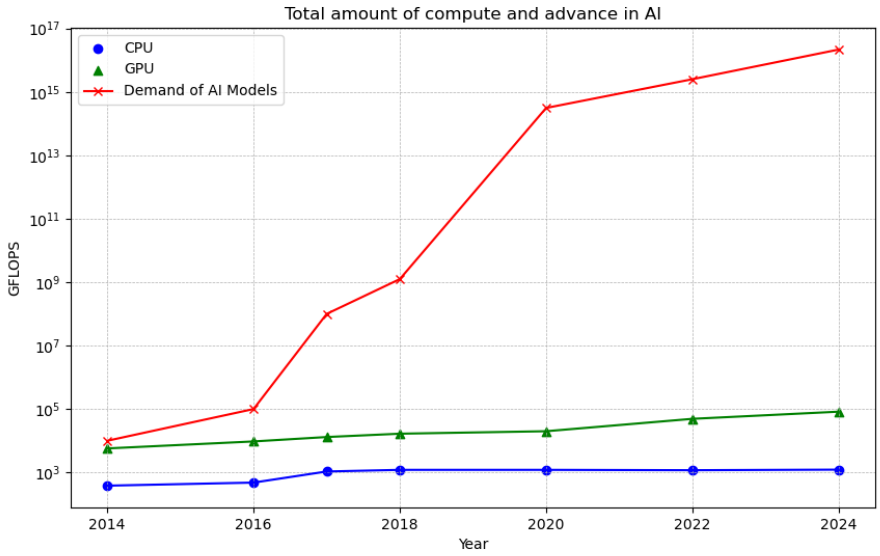


Figure 1.1: Total amount of compute and advance in AI.

mand for compute resources and the capabilities of conventional CPUs and GPUs, enabling more efficient and scalable system development and deployment.

The overhead of preparing input datasets becomes the most crucial step in the data-processing pipeline because approximate hardware improves significantly. Figure 1.2 shows this shifting bottleneck by presenting a striking comparison between the time spent on data preparation and the actual computation phase for various workloads, including scientific computing, machine learning, and deep learning applications. It reveals that the data preparation stage is significantly more expensive than the computing stage for most of the benchmarks.

The findings call for a paradigm shift in the way we approach hardware and software co-design, moving away from a narrow focus on a single architectural component,

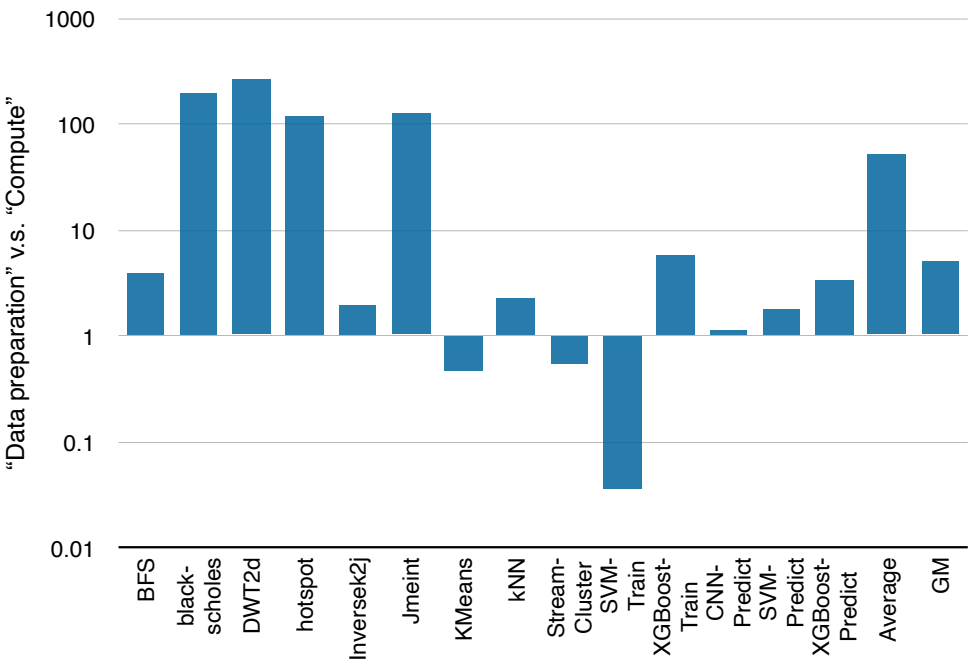


Figure 1.2: Data preparation overhead compared against the compute kernel.

optimization, and embracing a holistic system-level perspective that considers all stages of the application pipeline, including data preparation, data movement, and computation.

Another inherent limitation of approximate hardware is precision. Table 1.1 shows the data precision range supported by the emerging approximate hardware, it is important for the system architect and programmer to consider whether these accelerators can handle workloads properly without sacrificing too much accuracy.

Chapter 2- 4 are written in the conference paper format: each chapter has its own introduction, background, problem statements, challenges, proposed solution, system implementations, methodology, results, related works, and conclusion. This dissertation is organized in the following way.

Table 1.1: Supported data precision.

Approximate hardware	**Data type precision**
Tensor Cores (4th Gen)	8, 16, 32, 64-bit
TPU	8-bit
Edge TPU	8-bit
Digital Signal Processor	16, 24, 32-bit

Chapter 2 presents Varifocal Storage: Dynamic Multiresolution Data Storage, which addresses the data preparation overhead by adjusting the data resolution and performing quality control within the storage device.

Chapter 3 presents TCUDB: Accelerating Database with Tensor Processors, which demonstrates matrix processors can be applied to relational databases and accelerate SQL queries by revisiting application algorithms and data layout.

Chapter 4 presents PArtnerM^{2}LB: Personal Assistant Multi-device Machine Learning Benchmark, which provides a multi-device machine learning benchmark suite to assess performance and energy efficiency.

Chapter 5 concludes this dissertation.

Chapter 2

Revisiting the Hardware/Software Interface for Storage device

Approximate computing is increasingly being integrated into commercial systems for the sake of better performance and energy efficiency. Many applications nowadays can tolerate minor inaccuracies in input data [34, 61, 137, 160, 263, 167, 156, 147]. Current developments in approximate hardware accelerators, such as mixed-precision support in GPGPUs [177], NGPUs [260], NPUs [58], and CPUs [109], have further reduced compute kernel execution times and expanded the distance between data preparation and computation in approximate applications.

In approximation computing, most current research is to speed up compute kernels by design optimization of algorithms, programming frameworks, or architectural elements. However, modern computer systems hosting approximate computing still rely on storage system stacks created for traditional exact computing. The overhead related to preparing

input datasets—such as obtaining data from storage devices and modifying data resolutions—emerges as the most crucial step in the data processing pipeline when employing the most recent generation of GPGPUs to run approximate computation kernels.

2.1 Holistic system architecture

To tackle the previously described bottleneck in approximate computing, it is imperative that the storage device collaborates with the running program to provide datasets at the necessary resolution. This type of hardware-software co-designed can minimize the total bandwidth demand from the data source by lowering resolution, which in turn reduces the most latency-critical data-transfer overhead. Compute kernels can directly use these low-resolution inputs to avoid unnecessary data conversion. In spite of the clear benefits of a storage device that can effectively implement data-resolution reduction, building such a storage device is challenging, as the design must consider all of the following:

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

Quality Reducing data resolutions lowers the latency in data transfer but also has the potential to degrade output quality [120, 131, 132, 95]. If the input data leads to significantly inaccurate results, the application must recompute and/or iteratively retrieve the reduced data, both of which increase end-to-end latency.

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

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

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

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

mechanism to specify both the approximate operator and the appropriate resolution to further reduce programmer burden in designing applications.

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

Varifocal Storage makes the following contributions:

(1) It presents VS, a system architecture that optimizes the performance of approximate computing in full-stack design by dynamically changing data resolutions in storage devices to address the demands of performance, flexibility, cost, and quality.

(2) It demonstrates the potential benefits of adding another layer of quality control to reduce programmer burden by introducing the Autofocus and iFilter mechanisms that automatically determine data resolutions or even operators.

(3) It describes an implementation of VS to demonstrate the feasibility of VS architecture in modern storage devices and to evaluate the performance of VS using a wide range of approximate-computing applications.

By running a wide range of applications, we show that the manually controlled VS can speed up the most critical data preparation by 2.02× without significantly affecting accuracy. With the Autofocus mechanism determining the resolution, VS speeds up performance by 1.70× on average. Using the fully automatic iFilter mechanism, VS can achieve a speedup of 1.74×. Comparing the end-to-end latency of VS with that of conventional approximate-computing architecture, VS is 1.52× faster with programmer optimization, 1.43× faster using Autofocus, and 1.46× faster using iFilter.

2.2 Background

This section presents the motivation for our design, describes missed opportunities in modern computer architecture, and discusses alternative solutions.

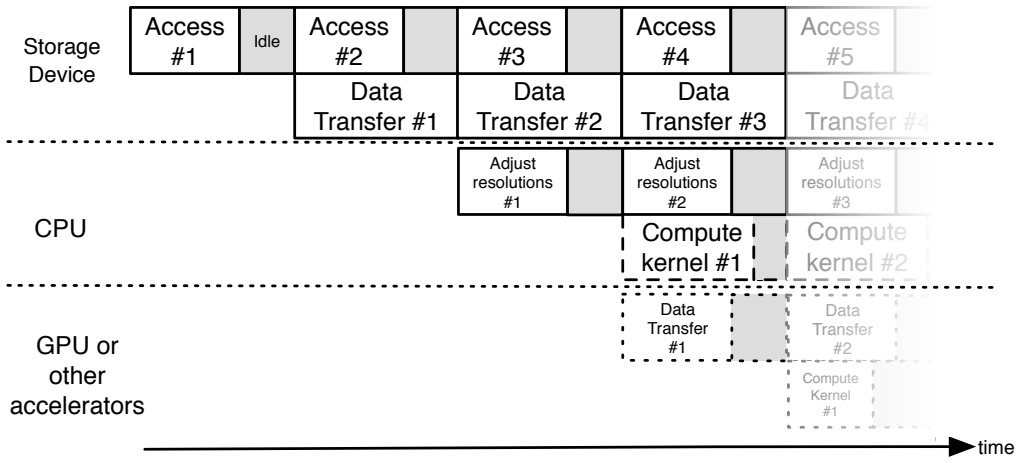


Figure 2.1: The data-processing pipeline of approximate applications using the conventional execution model.

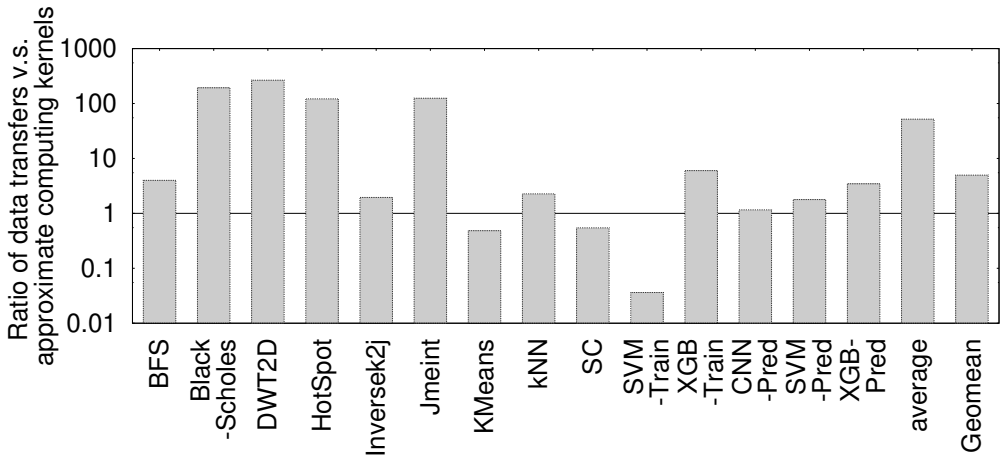


Figure 2.2: The data-exchange overhead compared against the execution time of performing compute kernels on the same amount of data.

2.2.1 The Overhead of Presenting Datasets in Different Resolutions

Figure 2.1 illustrates the data-processing pipeline of approximate-computing applications in modern heterogeneous computers. The computer first needs to issue I/O commands for the storage device to access raw data from its internal data arrays and then transfer the raw data through the underlying system interconnect while simultaneously serving other data-access requests. Once the host computer receives a chunk of data, the CPU can start producing datasets in lower resolutions. The approximate-compute kernel can then perform computations using the resolution-adjusted datasets. If the kernel can leverage a GPU, TPU, NPU, or other hardware accelerator, the system must additionally exchange among different components through the interconnects before the accelerator can compute on the prepared data.

With these approximate-computing-based acceleration techniques, the latency of retrieving data from the storage system becomes the most critical stage in the data-processing pipeline. Figure 2.2 compares the latency of receiving raw data chunks from a high-end NVM-Express (NVMe) storage device against the execution time of performing approximate/mixed-precision compute kernels on the same data chunks using an NVIDIA Tesla T4 GPU [177] for a set of applications [259, 208, 250, 31] (detailed description in Section 4.4). Using a highly optimized I/O library that saturates the NVMe I/O bandwidth, the overhead of receiving datasets exceeds the kernel execution as the most critical stage in a majority of these applications.

2.2.2 Missed Opportunities in Modern NVM-Based Storage Systems

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

Reduced data size Since approximate computing works on lower-resolution datasets, the compute kernels usually consume fewer bytes than exact computing ones. However, conventional storage interfaces, including those based on the latest NVMe standard [9], only support read/write commands that exchange raw data between source and destination; applications can never reduce the bandwidth demand of exchanging raw data between the storage device and the host.

Rich device-internal bandwidth Conventional storage interfaces waste the rich internal bandwidth of storage devices. The controllers found in modern datacenter SSDs, including the controller in the prototype SSD that we used for this paper (Section 4.4.1), support up to 32 channels. The internal bandwidth of our prototype SSD can reach up to 8 GB/s if the SSD uses MLC flash memory chips with an average reading latency at 35 μs for each 8 KB page [163, 82, 196]. However, the application only works on the host computer and exchanges data with the SSD using limited PCIe bandwidth. With newer, faster NVM technologies (e.g., ZNAND [214] or 3DXPoint [101]), the mismatch between the internal and external bandwidths can become more significant.

In-storage processing power Conventional interfaces also hide the freely available processing power in SSD controllers. Figure 2.3(a) shows the architecture of a modern datacenter SSD. In addition to NVM chips, an SSD contains general-purpose cores and

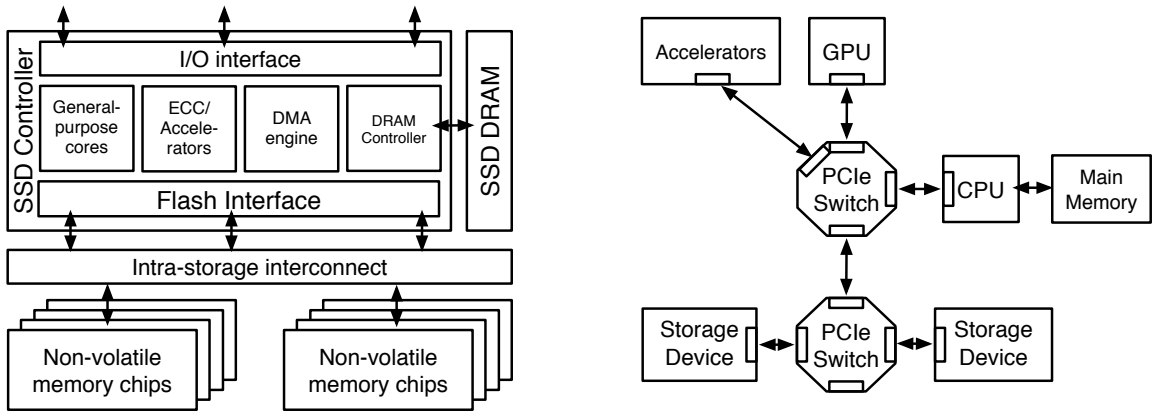


Figure 2.3: (a) The architecture of modern SSDs. (b) The modern PCIe system-interconnect architecture.

DRAM to execute firmware programs and to cache/buffer data. In spite of the limitations and dynamics of the outgoing bandwidth, the SSD controller can still access its own data-storage arrays with channels and banks. Nonetheless, the SSD's general-purpose cores remain unavailable to applications because conventional interfaces only support access to raw data.

Due to the relatively longer latency of accessing NVM devices and the over-provisioning of processing power to avoid the cost of adding an embedded operating system, SSD cores are idle for significant amounts of time. To accurately determine processor idle time, we analyzed the loading of each processor core in our baseline data-center SSDs under different scenarios. The maximum utilization appeared when we saturated the outgoing PCIe bandwidth by continually issuing 32 MB read requests. Under this scenario, the busiest SSD processor core spent 70.4% of its time parsing NVMe requests, and the second

busiest core spent 46.5% of its time receiving commands from the PCIe interconnect. All other processors responsible for managing data accesses for flash data were only busy 12.5% of the time. When the SSD is performing garbage collection, none of the processors are busy for more than 20% of the time due to the long latency of erase and write operations characteristic of SSDs. Consistent with these results, previous studies of data-center-class SSDs and common SSD prototypes [196, 264] have shown the average utilization of their SSD processors to be lower than 30%. With frameworks such as FlashAbacus [271], the SSD controller typically has even more idle time to spare for non-essential workloads.

2.2.3 Alternative Approaches

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

Increasing I/O bandwidth The most direct approach to improving data-transfer performance between the storage device and the host computer is to increase the I/O bandwidth of the storage device. However, this approach is difficult and expensive in modern architectures. Figure 2.3(b) shows the topology of attaching peripheral devices, host processors, and other accelerators in the most popular system interconnect for a PCI Express (PCIe). Most modern SSDs attach to a PCIe using 4× PCIe Gen3 lanes that provide up to 4 GB/sec of bandwidth. As modern CPUs incorporate their memory controllers on-chip and use an exclusive processor-memory bus, the bandwidth that the host application can use to

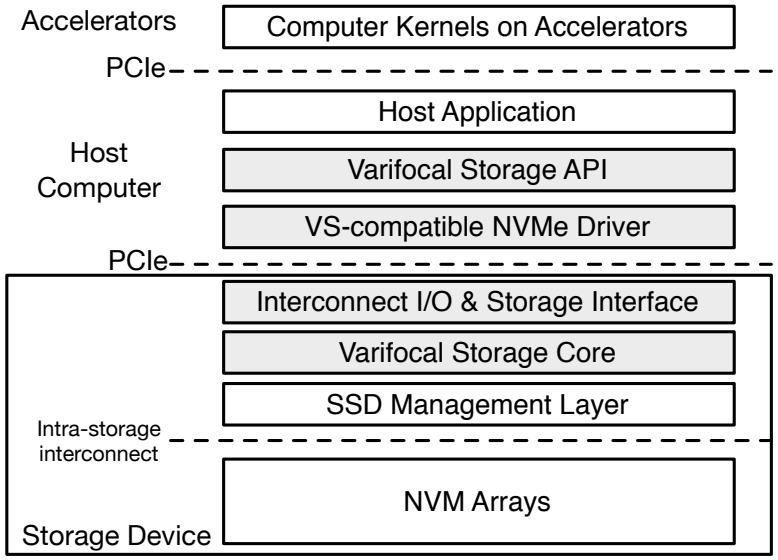


Figure 2.4: The VS system architecture.

communicate with other devices (including GPUs, NICs, hardware accelerators, and SSDs) is limited by the total PCIe bandwidth to which the CPU connects. As a result, the actual outgoing bandwidth that the SSD can use is narrower than the theoretical bandwidth, as it is usually the case that multiple devices are competing for the bandwidth going into the CPU/memory controller. In this modern system-interconnect architecture, increasing the bandwidth is very challenging since it requires the CPU to make more PCIe lanes available (i.e., increase the pin count of the processor) or reduce the number of peripheral devices that the system can connect.

Data compression Although lossy and lossless data-compression algorithms [28, 32, 193, 258] both help to reduce data size and save I/O bandwidth, the overhead of decompressing data on the destination computing device can easily cancel the benefit of reducing

data-transfer time; without appropriate hardware support, data compression may lead to performance degradation [138]. This is precisely what we observed (see Section 2.8.2). If the storage device stores data using lossy algorithms, the system sacrifices support for exact-computing.

In-storage processing (ISP) General-purpose intelligent storage frameworks such as Willow [217], Samsung's SmartSSD [51], Morpheus [237], Biscuit [83], Summarizer [128] and FlashAbacus [271] allow applications to use the processing power inside SSDs. These platforms fall short of approximate computing for the following reasons. (1) These platforms aim at offloading computation from exact computing and can lead to suboptimal performance for approximate computing. For example, Summarizer's filter operation picks pages that render datasets distorted from the raw dataset, which increases data exchanges and computation. (2) These platforms require the programmer to customize near-storage computation (e.g., resolution adjustments in approximate computing) on per application basis, increasing the burden of programmers and creating security concerns. (3) Even language support makes programming easier, the programmer can easily overestimate the capability of controllers and hurt performance.

Approximate storage Approximate storage systems store data using unreliable memory cells. Since these cells do not faithfully store raw data, systems that use them can neither support exact computation nor dynamically generate data in different resolutions; such approximate-storage systems simply sacrifice flexibility [133, 213, 69, 104].

Quality control Without revisiting storage-interface design, existing quality-control mechanisms must request full-size, raw data from the storage device [120, 131,

95, 212, 204, 228, 153]. Most frameworks control output quality by comparing subsets of results for exact and approximate computation, missing the opportunity to capture low-quality input that failed the requirement before computation begins. Section 2.8.2 presents the superiority of VS over conventional approaches in this respect.

2.3 Overview of VS

Figure 2.4 shows VS in a heterogeneous computer system. VS revisits the storage-system stack to allow the device to dynamically produce data with different resolutions on demand. The VS core layer resides inside the storage device to change data resolutions presented to applications. The VS layer interacts with existing system I/O interfaces and provides an extended interface for resolution adjustments. The VS layer also works together with the SSD management layer (i.e., the flash translation layer in flash-based, solid-state drives) to locate the requested data. The host system needs an extended kernel driver and API functions in order for the applications to send requests, exchange data, and receive feedback from the VS core layer. The host application interacts with the API and sends commands specifying the raw data types and operators that VS should work on.

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

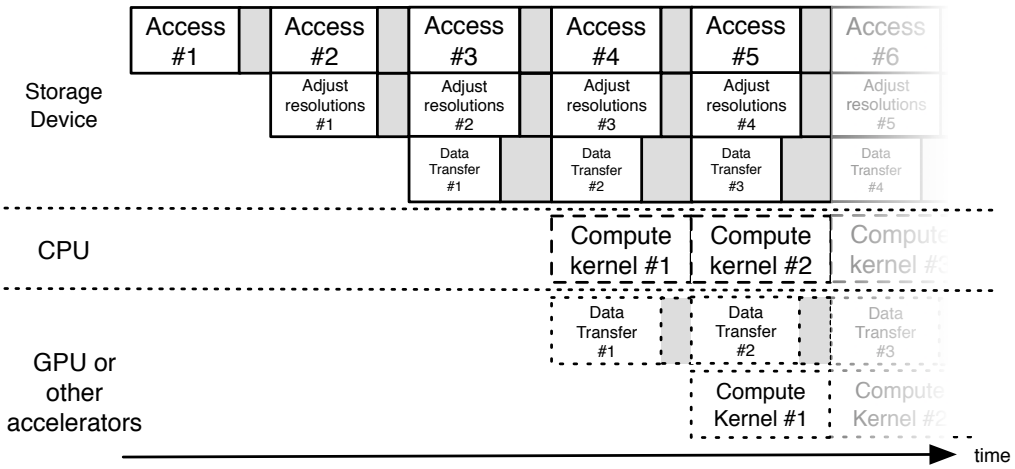


Figure 2.5: The data-processing pipeline of VS.

Figure 2.5 illustrates the data-processing pipeline that VS enables to tackle the challenges of performance, quality, flexibility, and cost. By using operators and quality-control mechanisms inside the storage device, VS exploits the richer internal bandwidth and idle processing power to efficiently adjust/prepare datasets in lower resolutions for approximate computing applications. Instead of always sending raw data, VS allows the storage device to send adjusted datasets to the host, reducing the total latency of transferring data over the system interconnect. In this way, VS mitigates the idle time in compute units and frees up CPU resources to tackle more useful workloads, leading to performance gains for approximate applications on the host side.

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

Table 2.1: Sample functions from the VS API.

2.4 The VARIFOCAL STORAGE programming model

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

VS provides a set of library functions for applications. The programmer uses these functions to set up (1) the operators required to read data and whether Autofocus or iFilter is enabled, and (2) the parameters that allows the underlying storage device to adjust data as well as control variables that Autofocus and iFilter use to control the adjusted data. Table 2.1 lists three representative API functions; the functions are used when an

application calls `open` to create a file descriptor. If the offset of an open file descriptor needs to be manipulated, the application simply uses conventional file system functions like `lseek` or `fseek`.

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

Figure 2.6: A KMeans code sample with inserted VS function calls.

Figure 2.6 shows KMeans code (Rodinia benchmark suite [208]) with VS function calls inserted. In the example, KMeans uses conventional system-library functions (e.g., `open` and `close`) to manage the file descriptor. If the program reads data using standard I/O functions (as in the two `read` function calls in the code), VS does not change the resolution of the accessed data. The modified KMeans code initiates VS for the `infile` file descriptor by calling `vs_setup`. This version of the code sets the desired operator and resolution. The `vs_setup` function also accepts an argument that describes the data formats. In the KMeans code sample, VS will interpret the file content as floating point numbers.

VS starts adjusting data only if the application calls the `vs_read` function. This function resembles the existing Linux `read` function except that (1) the resulting data size may be different from the requested data size, since operators will trim data sizes in most cases, and (2) the function will provide feedback regarding the resolution that VS selects. If the program calls a regular `read` function to replace the `vs_read` in Figure 2.6, VS will not change the data resolution (even if the program previously initiated VS using `vs_setup`). These API functions (e.g., `vs_read`) can interact with the underlying file system cache to further improve performance if another application is requesting the same dataset with the same resolution.

If VS successfully adjusts the data, the application can use a compute kernel that supports lower-resolution input (e.g., `cluster_approximate`) to further reduce the total execution time of the program. If the kernel is elastic to changes in dataset size (like machine learning algorithms), then no need to change the compute kernels. In many cases, the programmer can compose approximate versions of compute kernels by slightly

modifying the original kernel functions to operate on less precise data types or summarized input datasets [211, 210]. The application can also use library functions (e.g., Mixed-Precision CUDA libraries and FANN library for NPU [58]) leveraging approximate hardware accelerators to perform the approximation in compute kernels.

Depending on the approximate compute kernels that the application uses, the programmer can choose different VS operators for data adjustments when calling the `vs_setup` function. To determine the desired resolution, the programmer can leverage existing language frameworks and profiling tools [212, 211, 210, 22]. In addition to traditional approaches for determining resolutions, VS provides the Autofocus mechanism to automatically decide the resolution using a set of control variables that the programmer can optionally pass as parameters. The resolution-reduction choices Autofocus makes are usually more conservative than those of a programmer, but Autofocus can nonetheless help applications adapt to datasets. To ensure the quality of the execution result, VS may leverage existing approximate frameworks [120, 131, 212, 204, 228, 153].

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

2.5 The Core VARIFOCAL STORAGE Layer

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

2.5.1 VS Operators

VS provides several operators to adjust data resolutions before shipping the data to host applications. To achieve the best performance using the VS model, operators are selected in accordance with the following criteria: (1) The computation overhead must match the processing power inside the storage device. Thus, VS can minimize the impact on access latency and power consumption and avoid extra hardware costs. (2) A wide range of applications must be able to apply the operator, thereby allowing for more efficient use of valuable device resources (VS identifies the most useful operators from previous efforts [211, 210]). (3) The operator must allow VS to take advantage of mismatches between external and internal bandwidths and downsize the outgoing data—the VS model is most effective when the data adjustment can reduce the demand of interconnecting bandwidth.

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

Data Packing The data-packing operator trims the dataset size by using fewer bytes to express each item and by condensing the layout in memory. A data-packing operator is suitable for applications that only use a small range within the number space of

the original data type and for applications that can tolerate some inaccuracies in the input data. Since the data-packing operator translates raw data into a less-precise data type, it can potentially decrease accuracy (e.g., double→float→half or int64→int32→short→char).

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

Reduction/Tiling The reduction operator applies a function (e.g., average) to a group of input values and yields a single output value. After applying a reduction operator, VS sends only the resulting value of each group in order to reduce the amount of data passing through the system interconnect. This operator is especially useful for machine learning and statistics applications when the input data is uniformly distributed [210].

Sampling The sampling operator chooses a subset of items from the raw data and sends the selected items to the host computer. Operators in this category can perform uniform/random data selection or report only the most representative data. The sampling operator helps to filter out repetitive/similar inputs that make no contribution to the final application result. If the compute kernel is elastic with respect to the number of records within the dataset, the sampling operator can achieve the same effect as that of loop perforation [165, 223, 180] but without any code modification (without the VSampling operator, conventional loop perforation needs the raw data to be present in system memory).

Besides, by providing the preceding types of operators, VS gives system designers the chance to extend the number of operator types using the mechanisms described in Section 2.6.3.

2.5.2 Autofocus and iFilter

The Autofocus and iFilter mechanisms provide quality control and reduce the amount of programmer effort required to adjust data resolutions. Autofocus and iFilter are inspired by two previously observed phenomena: (1) The quality of the input data affects the quality of the result in approximate computing [120, 131, 132]. (2) A small subset of input data is representative of the rest of the input data in approximate-computing applications that tolerate inaccuracies [131]. Building upon these observations, Autofocus and iFilter can select the resolution/operator using only a small portion of the raw input data from a requested dataset and then monitor the quality of the adjusted input data.

Autofocus

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

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

Algorithm 1 Autofocus

Input: op, CVs ▷ CVs are optional

1: **for** each $r \in R$ **do** ▷ r is sorted in ascending order
2: $D \leftarrow RawData$
3: **for** each $d \in D$ **do**
4: $d' \leftarrow adjust\_data(d, op, r)$
5: $\Delta \leftarrow compute\_CVs(d, d', op)$
6: **if** Δ satisfies CVs **then**
7: remove d from D
8: **if** D is empty **then**
9: **return** r
10: **end if**
11: **else**
12: **break**
13: **end if**
14: **end for**
15: **end for**

VS Operator	Function $compute\_CVs$	Description
`Data Packing`	$abs(data_{raw}, data_{adjusted})$ and $min_{new\_data\_format} \leq data_{new} \leq max_{new\_data\_format}$	For data packing, VS calculates and check if (1) the absolute difference between the original data and adjusted data is smaller than the given threshold and (2) adjusted data falls in the range of the target data type.
`Quantization`	$abs(data_{raw}, data_{adjusted} * scale\_factor)$, where $scale\_factor = \frac{max(data_{old\_data\_format}) - min(data_{old\_data\_format})}{max(data_{new\_data\_format}) - min(data_{new\_data\_format})}$	For quantization, VS controls the quality by rescaling the adjusted data back to the raw data format and measuring the absolute difference. VS drops the adjustment if the difference is greater than the given threshold.
`Reduction/Tiling`	$abs(data_{raw}, data_{adjusted})$	For reduction/tiling, VS computes the absolute difference between raw data and adjusted data. VS compares if the absolute difference is smaller than the given threshold.
`Sampling`	$binary\_distance(data_{raw}, data_{adjusted})$ [195]	For sampling, VS calculates the Hamming distance between raw data and adjusted data and drops the current decision if the distance is larger than the given distance.

Table 2.2: Summary of function $compute\_CVs$.

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

the remaining adjusted data. Table 2.2 summarizes how we compute the control variables for each VS operator.

If every checked piece of the adjusted data successfully passes through the $compute\_CVs$, VS will report the current resolution to the host application and transfer the adjusted data (Line 9 of Algorithm 1) through the system interconnect. If the quality of the adjusted data (d') fails on the control variables, Autofocus will fall back to the next resolution (Line 12 of Algorithm 1).

iFilter

iFilter can work without programmer input and is more effective than Autofocus for applications having compute kernels that are compatible with multiple VS-operators. Algorithm 2 shows how the iFilter mechanism works. The iFilter algorithm includes a *decision-making phase* (Line 1–Line 18) and a *monitoring phase* (Line 19–Line 32). In the decision-making phase, iFilter will try out all available VS operators (OP) that can be applied to the input data type for the first few pages (8 in our experiments) of the requested data. The iFilter algorithm is similar to the Autofocus algorithm in that it selects the most appropriate resolution for each operator, except that iFilter will keep track of the resolution ($min\_res[op]$) and the resulting data size ($min\_size[op]$) for each operator (Line 5 & Line 11).

After the decision-making phase, iFilter will enter the monitoring phase and select the operator that yields the smallest data size (Line 19). iFilter uses the selected operator (op) to adjust every piece of raw data. If iFilter successfully reaches the end of the request,

iFilter will report the selected operator and resolution (Line 27) and send the adjusted data to the host. If iFilter fails to reach the end of the request, it will remove the current resolution from the available set of resolutions ($R[op]$) and restart the decision-making phase to choose the next appropriate operator and resolution (Line 30 & Line 30). The computation overhead for iFilter is thus higher than that of Autofocus since iFilter examines more operators to choose the one with the minimum amount of data going through the system interconnect. However, the additional overhead is negligible with large datasets because the relevant VS operators need only be applied to the first few chunks of the dataset.

2.6 Building a Storage Device Compliant with VARIFOCAL STORAGE

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

2.6.1 NVMe Extensions for VS

Conventional storage interfaces such as the popular NVMe protocol only support read/write commands for data access. Therefore, the NVMe extensions for VS need to provide commands to set up VS operators and apply those operators on datasets. The extended NVMe interface aligns with the programming model in Section 2.4 to simplify the complexity of software implementation.

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

VS uses an abstraction similar to an instruction-set architecture that allows the API to map the demanding operators to each stream. Each operator starts with a 4-byte opcode followed by a 4-byte integer for the number of arguments, which is then followed by the arguments (e.g., target data resolutions, quality control variables). For each category of operator, VS provides a different opcode for different data types. The API generates a sequence of operators and works with the driver to store the sequence in a host DMA page for the SSD to access.

Upon receiving the `vs_setup` command, the SSD will add the page specifying the operators into its internal data structure, which usually resides in the DRAM space of the SSD. Later commands can use the descriptor number to indicate the operators that a `vs_setup` command previously set and look up the corresponding operators from the

internal data structure. When the application does not need the setup operators for the I/O stream, the `vs_release` command will signal the SSD to release the descriptor, allowing a later `vs_setup` command to reuse the descriptor number.

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

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

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

Host System	
CPU	Intel Core i7-7700K [100] @ 4.2 GHz
GPU	NVIDIA Tesla T4 [177]
OS & file system	Linux Kernel 4.15 & EXT4
baseline/VS-compliant SSD	
Controller	Microsemi flashtec controller with 32 channels [196]
DRAM	2GB DDR4 DRAM
Capacity	768 GB with 10% overprovisioning
Flash Chip	MLC NAND/8 KB page size [163]
I/O interface	NVMe through PCIe 3.0×4

Table 2.3: The platform configuration used for evaluation.

2.6.2 Architecting a VS-compliant SSD

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

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

accessing its own data array if the applied operator does not create more than the average data-access latency in the pipeline.

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

2.6.3 Adding New Operators

In our SSD, VS operators are implemented as overlay functions in the firmware programs. With the extended NVMe protocol providing a mechanism to exchange information for adjusting data resolutions, the overlay functions receive the same set of arguments (including the resolution and the pointer to the SSD DRAM data-buffer location) and report the data size and resolutions through a data structure defined in our framework. To add a new operator, our current tool chain requires the designer to first write C functions. The designer also needs to update a header file where the firmware program identifies and locates the new operator. The designer can then use a cross-compiler to generate machine code for the controller's microarchitecture. Finally, the system deploys the compiled firmware program to the SSD through the standard firmware update command in the NVMe protocol [9].

2.7 Experimental Methodology

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

2.7.1 Experimental Platform

We built a VS-compliant SSD by extending a commercialized, datacenter-class SSD. We attached the VS-compliant SSD to a high-end heterogeneous machine with a GPU. The host operating system contains the extended NVMe driver to support additional VS NVMe commands. Table 2.3 lists the key specifications of the host computer and the SSD. The VS-compliant SSD runs our modified firmware programs. The firmware is also compatible with a standard NVMe. Since we did not modify the code that handles regular NVMe commands, the firmware achieves the same performance as a regular NVMe SSD with the same hardware configuration. Throughout our tests, the baseline SSD achieved a 3.2 GB/s bandwidth when communicating with the host systems, but the theoretical internal bandwidth is twice of that.

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

2.7.2 Benchmarks

The workloads we used for VS-performance assessment are shown in Table 2.4. We used these workloads on both the baseline configuration and the VS-enabled configurations. We selected the given set of applications based on the following criteria: (1) the application had to be representative of approximate computing workloads found in a publicly available repository, and (2) the application had to accept large, publicly available datasets or provide a data generator capable of producing large, arbitrary datasets that could serve as meaningful input. Table 2.4 lists the dataset sizes that we used in experiments; these are also the largest dataset size that our GPU can accommodate but do not represent a limitation of our SSD or the VS programming model. We followed examples found in previous work in modifying the compute kernels of Black-Scholes [259], Hotspot, DWT2D, and KMeans [211, 210]. We also implemented approximate-computing versions of kNN, SC, SVM, and XGBoost by leveraging the native mixed-precision support in NVIDIA's latest Turing architecture. When running these workloads, we used the default parameters that each workload or its demo script suggested. For each application, we also tried our best to exploit pipeline parallelism that overlaps I/O, resolution adjustment and compute kernels to hide latencies.

Table 2.4 also lists the lowest data resolutions and the corresponding operators that these approximate-computing applications can accept. For each workload, we carefully profiled and chose the operators and their parameters to limit the relative error rate to less than 1% compared to the exact version of the same application.

In our experiments, three groups of benchmark applications were chosen to use the

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

For the basic/programmer-directed VS version, we applied operators and target resolutions as shown in Table 2.4. When the Autofocus and iFilter mechanisms are enabled, our implementations check the feedback from the VS API. If VS decides to adjust data resolutions, our code can choose to apply appropriate compute kernels to process data. Otherwise, our code uses the baseline compute kernels. When using Autofocus and iFilter, we selected a set of default control variables that were relatively conservative across all applications. For control variables, we used a delta value of 1% for packing, reduction, and quantization as well as 1% binary difference [195] for sampling, since we are targeting at less than 1% error rate.

2.8 Results

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

2.8.1 The Overhead of VS Operators and Mechanisms

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

The Autofocus and iFilter mechanisms also use the SSD general-purpose cores to execute their algorithms. For the Autofocus mechanism, VS takes at most 25 μs to stabilize the resolution for an operator working on binary numbers. For iFilter, the decision-making phase takes about 150 μs to make its first decision because we need to perform sampling for all operators. Once Autofocus and iFilter have determined the required resolution, both mechanisms simply compute on values for control variables, so the overhead is negligible and the throughput unaffected.

2.8.2 The Performance of Data-Resolution Adjustments

Figure 2.7 shows the speedup in reading input datasets and adjusting data resolutions for each workload using different VS modes; VS is compared with the conventional

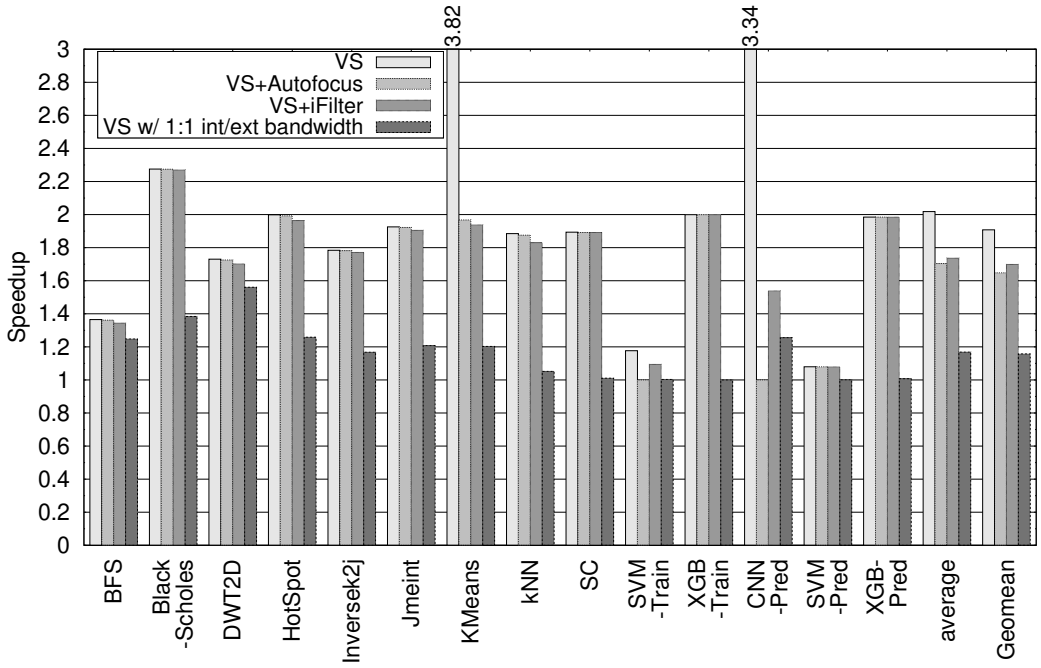


Figure 2.7: The speedup of reading inputs and adjusting data resolutions using VS.

approximate programming model that relies on the host to adjust data resolutions.

Programmer-Directed VS

Choosing the default VS settings and specifying the desired operator and resolution can speed up the performance of data adjustment by 2.02×. For KMeans and CNN-Pred, which tolerate very low-resolution inputs, the speedup of data adjustment can reach up to 3.82× since the storage device only needs to send out 25% of the raw data size to the host. For Black-Scholes, adjusting raw data on the host is more time-consuming than data transfer. Therefore, VS can achieve more than 2× speedup since VS also takes the advantage from the ISP model for data adjustment. Even with the geometric mean that discounts outliers, VS still exhibits a 1.91× speedup (Figure 2.7).

Autofocus and iFilter

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

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

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

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

In our experiments, the relative error rate of computation observed when using

Autofocus and iFilter never exceeded the values in Table 2.4 because Autofocus and iFilter always made more conservative choices than the programmer.

Internal/external bandwidth

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

The "VS w/ 1:1 int/ext bandwidth" bar in Figure 2.7 shows the speedup from using this modified version of our prototype SSD. Without being able to rely on the host CPU for data adjustment, the basic VS still speeds up the total latency of preparing datasets by 1.17×. Additionally, VS reduces the size of data going through the system interconnect, making applications more adaptive when many devices have to compete for the same set of limited PCIe links.

Case study: shared datasets

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

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

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

Case study: diverse datasets

Since Autofocus determines the most appropriate resolution by examining the characteristics of datasets, Autofocus makes VS more adaptive to changes in input datasets for each approximate-computing application. In addition, Autofocus does not rely on feed-

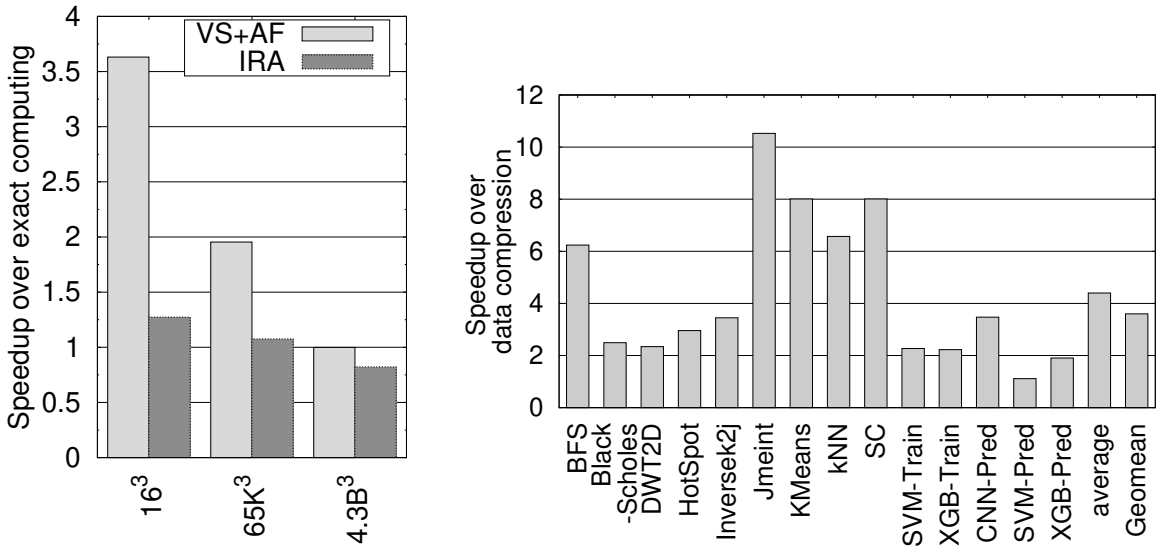


Figure 2.8: (a) The speedup of end-to-end latency using VS and conventional approximate-computing framework. (b) The speedup of data preparation using VS, compared with data compression.

back from kernel computation results and does not require the storage device to send raw data in the beginning, so Autofocus is more efficient than conventional approaches tackling the same problem. To illustrate this strength of VS, we modified the data generator of Jmeint from AXBench to generate random points in various sizes of 3D spaces. We next present the results when using Autofocus with datasets from three different dimensions: 32^3, 65536^3 ($65K^3$) and 4294967296^3 ($4.3B^3$).

Figure 2.8(a) shows that VS with Autofocus exhibits significantly shorter end-to-end latency for all datasets compared to the conventional approximate-computing approach using IRA [131]. We used the unmodified exact-computing version of Jmeint as the baseline. Since Autofocus does not need to send raw datasets to the host, VS outperforms IRA by more than 2.86× in the case of the 32^3 dataset, with VS only sending data encoded in 8-byte integers. For the $65K^3$ dataset, Autofocus down-samples the datasets to short data type, leading to a 1.80× speedup over IRA.

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

Data Compression Comparisons

Since the most significant VS performance gain comes from reducing data-movement overhead, we also compared VS with several high-performance lossy/lossless compression algorithms: FPC [28], C-Pack [32], BDI [193], and ZSTD [258]. We clocked the time of reading compressed data, of decoding data, and of adjusting resolutions. We excluded the overhead of compressing data. We use the best-performing compression algorithm as our baseline in Figure 2.8(b), showing the speedup of using VS comparing against data compression.

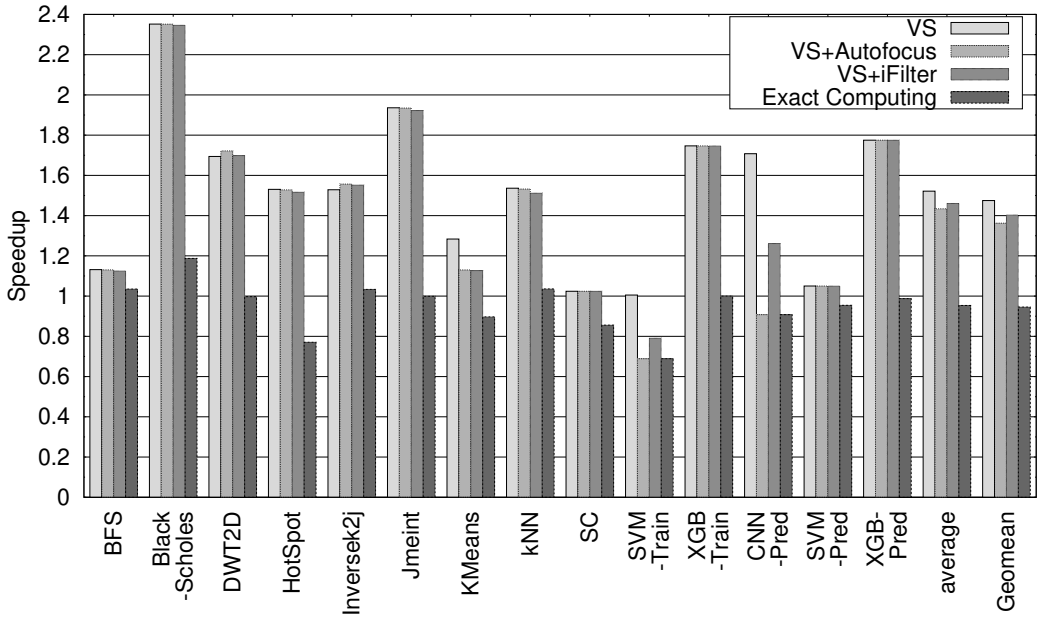


Figure 2.9: The speedup of the end-to-end latency.

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

2.8.3 The Impact of VS on Total Application Latency

Figure 2.9 shows VS's impact on the relative end-to-end latency of running a complete workload using workloads with the conventional approximate computing approach with GPU-accelerated kernels as the baseline. Since VS efficiently prepares input datasets in

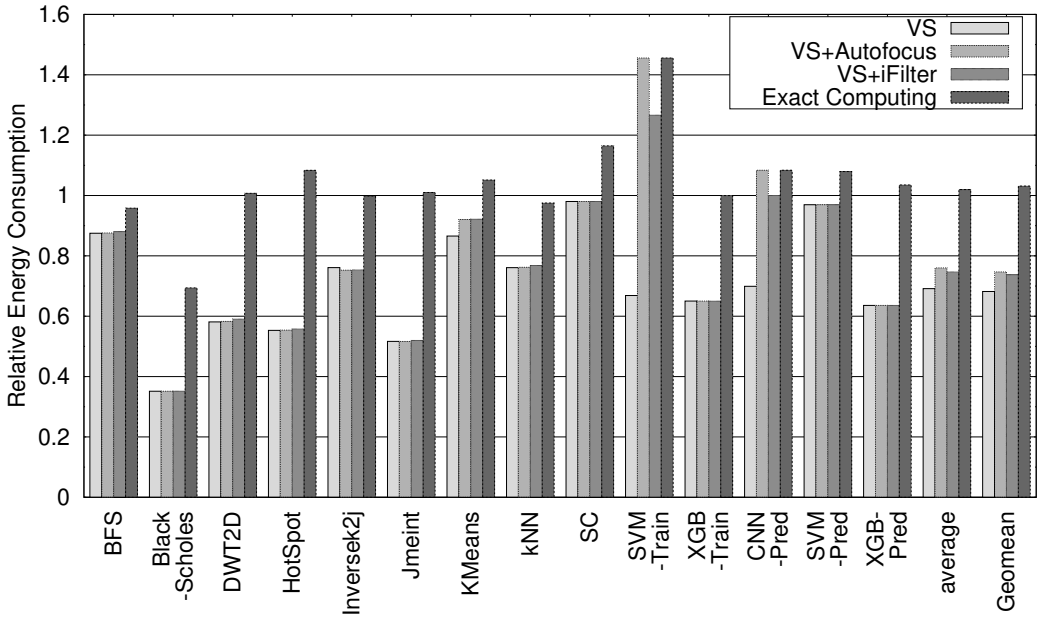


Figure 2.10: The total system energy consumption.

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

2.8.4 Power and Energy

To quantify the effect of reducing the CPU workload, total power, and energy consumption, we first examined the CPU frequency when performing data packing on the VS-compliant SSD using the baseline host-version implementation. We sampled the CPU frequency every 500 ms. Even though packing is a very lightweight operation, adding this computational burden to the host program still forces the CPU frequency to go beyond 3 GHz most of the time. For VS, which requires that the CPU handle DMA or issue NVMe commands, the peak CPU frequency during the data I/O is only 1274 MHz. Using a Watts Up meter to measure the power consumption, the total system consumes 64.7 W for this frequency. Without VS, the system consumes an average of 70.8 W during the whole data I/O process.

Since VS reduces both the power consumption during I/O and the total application latency, VS also reduces the energy consumption. To measure power consumption, we used Watts Up to measure the power draw every 200 ms. Figure 2.10 shows that the basic VS achieved an average energy savings of 32% for these applications compared to the conventional approximate-computing approach. Even without a programmer's aggressive decision in adjusting data resolutions, VS's Autofocus and iFilter still achieve the same level of energy savings in most applications, except for SVM-Train and CNN-Pred due to their increased end-to-end latency as Section 2.8.2 explains. Autofocus and iFilter provide energy savings of 25% and 27%, respectively. In contrast to this, the conventional architecture with aggressive data adjustments and approximate-computing kernels could only improve energy consumption over exact computing by 5%.

2.9 Other Related work

Approximate computing has a significant presence among solutions that tackle the limitations of modern hardware design. Using simplified algorithms, smaller ALUs/FPUs, or faster operators, approximate computing maximizes the area-efficiency of silicon chips [109, 232, 87, 113, 117, 139, 162, 218, 238, 262, 278]. By designing simpler, faster approximate circuits (e.g., circuits that use neural-network accelerators [168], load value approximation [164], or approximate memoization [7]), approximate computing also avoids intensive usage of slower but precise circuits for better performance or energy efficiency. In addition, approximate computing allows hardware designers to use unreliable transistors that are commonly found in advanced process technologies [119, 44, 35]. Yet all of the approximate-computing research cited above still follows the single-point design principle, creating the resolution-adjustment problem that this work tries to address. VS is complementary to these projects and can work together with them to address the issues they raise.

To reduce the overhead of applying approximate hardware or software-based approximate-computing solutions, current research projects provide support and analysis through programming language extensions and compilers [120, 131, 212, 211, 210, 22, 119, 44, 35, 15]. Since VS simply exposes its features to applications through an API and proposes extensions in the I/O protocol and firmware programs, applications can adapt VS without programming language extensions or compilers. Further, the Autofocus and iFilter mechanisms control input quality after applying VS operators within storage devices, so VS can react before the compute-intensive kernel starts. VS and existing projects are also orthogonal; the

system can incorporate VS with existing approximate-computing programming frameworks to use VS operators and mechanisms more efficiently.

Even though VS shares the benefits from recent advances in ISP [271, 217, 237, 83, 128, 20, 231, 118, 252, 30, 112, 203, 115, 216, 36] and near-data processing [190, 233, 55, 155, 126, 63, 152, 5, 227], these frameworks need the mechanisms that VS offers in order to execute approximate computing applications efficiently. And while using approximate computing in channel encoding [123, 187] and memory controller [111] can achieve an effect similar to that of VS in terms of reducing data-movement overhead, VS is independent of these projects and requires no changes in hardware.

2.10 Conclusion

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

Through experiments conducted with a VS-compliant SSD and the experience gained from tailoring applications on the platform, this paper also demonstrates that a VS-compliant architecture requires very few modifications to hardware or software. A clear indication of VS's efficiency relative to conventional approximate-computing architectures

may be found in the 2.02× speedup observed for VS-based data-resolution adjustments and the 1.52× speedup observed for total end-to-end latency, with both improvements producing a change in results of less than 1%. In summary, VS improves performance, maintains flexibility, guarantees quality, and incurs no storage-space overhead for adjusting data resolutions—all at a low cost.

Algorithm 2 iFilter

Input: OP, CVs ▷ OP, CVs are optional

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

Workload Name	**Application Category**	**Operator**	**Resolution**	**Raw Data Size**	**Relative Error Rate**
Breadth-First Search (BFS) [208]	Graph Traversal	Packing	62.5%	3.5 GB [208]	0%
Black-Scholes [259]	Financial	Packing	50%	3 GB [259]	< -0.25%
HotSpot [208]	Physics Simulation	Reduction	25%	2 GB [208]	< -0.15%
2D Discrete Wavelet Transform (DWT2D) [208]	Image/Video Compression	Reduction	50%	1.6 GB [208]	< 0.1%
Inversek2j [259]	Robotics	Packing	50%	2 GB [259]	< -0.01%
Jmeint [259]	3D gaming	Packing	50%	2 GB [259]	< -0.02%
KMeans [208]	Data Mining	Quantization	25%	1.36 [208]	< -0.97%
k-Nearest Neighbors (kNN) [159]	Data Mining	Packing	50%		< -0.01%
streamcluster (SC) [208]	Data Mining	Packing	50%		< -0.01%
ThunderSVM–Train (SVM-Train) [250]	Machine learning	Sampling	75%	2.6 GB [224]	+ 0.5%
ThunderXGB (XGB) [251]	Machine learning	Packing	50%		< -0.10%
CNN–Pred [3]	Machine learning	Quantization	12.5%	0.95 GB [224]	< -0.6%
ThunderSVM–Pred (SVM-Pred) [250]	Machine learning	Packing	50%		< -0.01%
ThunderXGB–Pred (XGB-Pred) [31]	Machine learning	Packing	50%		< -0.10%

Table 2.4: Workloads, default VS operators, input data sizes, and error rates.

Chapter 3

Repurposing the Matrix Processors

The enormous demand for artificial intelligence (AI) and machine learning (ML) workloads has driven the development and integration of accelerators containing instructions operating on two-dimensional tensors (i.e., matrices). Examples include NVIDIA's Tensor Core Units (TCUs) [158], Google's Tensor Processing Units (TPUs) [215], and Apple's Neural Processing Units (NPUs) [12]. Improving matrix algebra through matrix units (MXUs), which popular AI/ML models heavily rely on, drastically increases the orders of magnitude speedup and energy efficiency. However, these hardware accelerators are application-specific, and designed for neural network-based AI/ML models. This limits the applicable domains of these powerful hardware accelerators.

3.1 Overview of TCUDB

To explore the different use cases other than AI/ML, we explore opportunities of integrating Tensor Core Units (TCUs) into a database engine's architecture. Despite being

originally designed for AI/ML workloads, tensor processors also hold potential performance improvements for database engines. This is due to both the increasing demand for native support of linear algebra queries (e.g., matrix multiplication itself) in SQL DB engines [93, 4, 56, 150, 53] and the observation that a large number of regular query operators can be cast into matrix multiplication. For example, one can show that the most commonly used natural joins [11, 45] and group-by aggregates can be encoded as matrix multiplication, which enables TCUs to deliver exceptional performance.

However, the presence of these AI/ML accelerators, or more generally matrix processors, does not provide a drop-in upgrade to the query engine's performance. Three major challenges must be addressed.

Challenges. First, the conventional GPU databases primarily implement the physical operators (e.g., the partitioned hash join algorithm [114]) in a non-matrix-friendly manner. These algorithms and operators typically do not operate on tensors directly. As a result, it is hard to modify them with the intent of taking advantage of TCUs' computation power.

Second, although DB operators such as joins can theoretically be encoded as matrix multiplications, executing all of them as dense multiplication might not always be beneficial. For example, the underlying data distributions can cause the two operands to be sparse matrices, which require a different data organization and APIs to achieve the best performance.

Next, a DB engine with TCUs must prevent itself from generating erroneous query results because of the low-precision nature of the tensor processors. The current tensor processors are limited in precision as AI/ML applications are error-tolerant because NVIDIA's

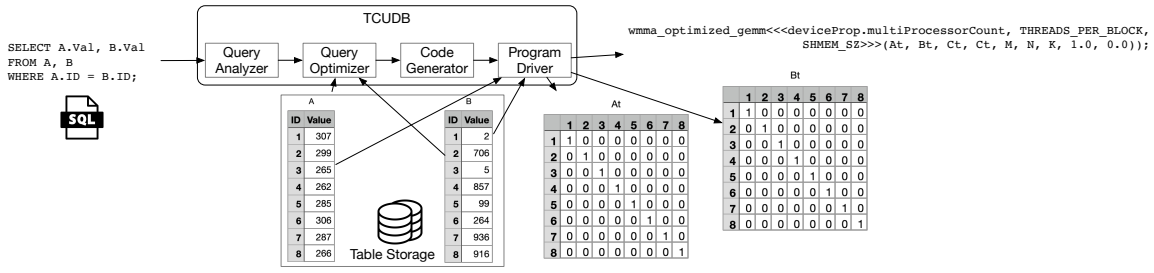


Figure 3.1: An overview of TCUDB's workflow.

TCUs only support 16-bit floating-point numbers while Google's TPUs only work on at most 8-bit integers. Moreover, these tensor processors share the same data movement overhead with other hardware accelerators while additionally suffering from the data transformation overhead (i.e., table $\rightarrow$ tensor). A higher precision requirement means introducing more data movement and transformation overhead. As a result, the proposed system must maintain a balance between two factors.

TCUDB.

We present TCUDB, an analytic database query engine that explores the potential of tensor processors to accelerate analytic query workloads using TCUs by tackling the aforementioned challenges. Figure 3.1 provides an overview of the system architecture of TCUDB. TCUDB extends the common architecture of GPU-accelerated databases [80, 242, 255, 267, 25, 246, 256, 136, 191, 219] as a way to further accommodate executing query operators with TCU acceleration in the query analyzer, the query optimizer, the code generator, and the program driver.

To address the challenge of executing queries using matrix operations, we re-engineered a set of query operators that are theoretically feasible to be mapped to ten-

sor/matrix algebra operations for TCUDB. The query operators cover a large set of commonly used ones including natural joins and group-by aggregates. As shown in Figure 3.1, TCUDB features a code generator for generating executable code mapping input tables to tensor format and processes the query as matrix multiplication via WMMA or cuBLAS API calls. Depending on the data sparsity, TCUDB provides the option of sparse tensor encoding with sparse matrix multiplication. We developed the TCU-SpMM operator to support sparse matrix multiplication with TCU acceleration. Then, the TCUDB query analyzer is capable of generating query plans, which use these TCU-accelerated physical operators.

To resolve the challenge of limited precision and overhead in modern tensor processors, TCUDB's query optimizer carefully gauges the parameters in precision, data movement overhead, data transformation overhead, and computation throughput — as using lower data precision yields lower data movement overhead and higher computation throughput, but also takes higher risks of leading into unacceptable answers as well as higher data transformation overhead. TCUDB presents an adaptive mixed-precision query optimization that dynamically selects the most appropriate precision in delivering the desired level of accuracy using the shortest end-to-end latency to handle queries.

TCUDB makes the following contributions:

- We explored the space of opportunities of optimizing a GPU-accelerated analytic query engine by leveraging TCUs. In our initial investigation, we found that TCU delivers >5× performance gains for matrix multiplication compared to the conventional CUDA cores in GPUs. This finding contradicts the conventional wisdom that considers matrix multiplication a slow operator because of its high computational complexity. As such, TCUs provide new opportunities to optimize processing analytic queries as matrix multiplication.

- Next, we identified a collection of query patterns that can potentially be accelerated by TCUs. The query patterns include the most commonly used SQL operators in analytic queries such as joins and group-by aggregates (e.g., `SUM` and `COUNT`). We demonstrate simple algorithms for transforming relational tables into matrix format and translating SQL operator into one or more matrix multiplication operators. Our algorithmic design is generic as it can be generalized to multi-way joins and aggregation over joins.

- We designed and implemented TCUDB, a TCU-accelerated analytic database engine. On top of a traditional GPU database [1], TCUDB features a query optimizer that identifies (1) the most efficient TCU query plan and (2) the best GPU/CPU-based plan and decides which plan to execute via cost estimation. If a TCU-accelerated plan is selected, TCUDB leverages a code generator to rewrite (parts of) the query into C

[1] We archive the source code and workloads at our GitHub page: https://github.com/escalab/TCUDB

programs that invoke NVIDIA's CUDA API. To the best of our knowledge, TCUDB is the first analytic database engine with TCU-accelerated built-in.

- We evaluated TCUDB on 4 real-world use cases: (1) linear algebra (LA) queries, (2) entity matching (EM), (3) graph analytics, and (4) analytic queries such as the star-schema benchmark. TCUDB demonstrates an outstanding performance advantage over a GPU-based engine (YDB), by achieving up to 288× speedup. Our results also highlight the necessity of the query optimizer and TCUDB's scalability advantage in future GPU architecture.

3.2 Background and Motivation

This section describes the background of the conventional query processing on a GPU and the motivation inspired by the characteristic of Tensor Core Units (TCUs). By comparing to the traditional vector processing model, we demonstrate the tensor processing model in a database system that can deliver better performance on linear algebra queries in terms of computing capability and scalability.

3.2.1 Tensor Core Units (TCUs)

As deep neural networks heavily rely on operations using matrix multiplications (e.g., convolution), recent hardware accelerators feature matrix units (MXUs) in their microarchitectures to significantly boost the performance in machine learning (ML) workloads. Famous examples include NVIDIA's Tensor Core Units (TCUs), Google's Tensor Processing Units (TPUs), and Apple's Neural Engine.

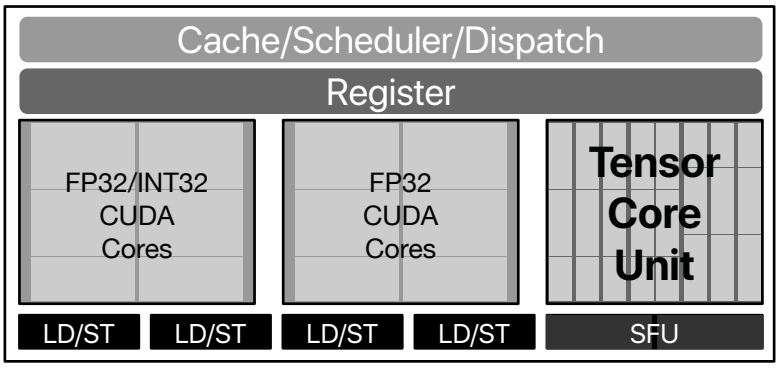


Figure 3.2: The GA102 Streaming Multiprocessor (SM) architecture in GeForce RTX 30-series GPUs.

This paper selects TCUs as the underlying accelerators for the following reasons: (1) Programmability: TCUs expose their low-level C++ API to programmers such as highly optimized cuBLAS APIs or customizable WMMA (Warp Matrix Multiply-Accumulate) APIs, giving programmers complete freedom in implementing algorithms and integrating with existing systems. By contrast, their counterparts are only programmable through domain-specific languages tailored for ML. (2) Accessibility: TCUs are now standardized components in NVIDIA's GPU architectures, ranging from high-end server solutions, gaming solutions, to embedded solutions. Conversely, high-performance TPUs are only accessible through Google's cloud services and Apple's NPUs are only available on their machines. (3) Flexibility: Tensor cores together with other ALUs on the GPU supports multiple data precision with various operations. Other ML accelerators only support limited precision.

TCUs are currently available as separated functional units from conventional vector floating-point and integer ALUs within the current generation of streaming multipro-

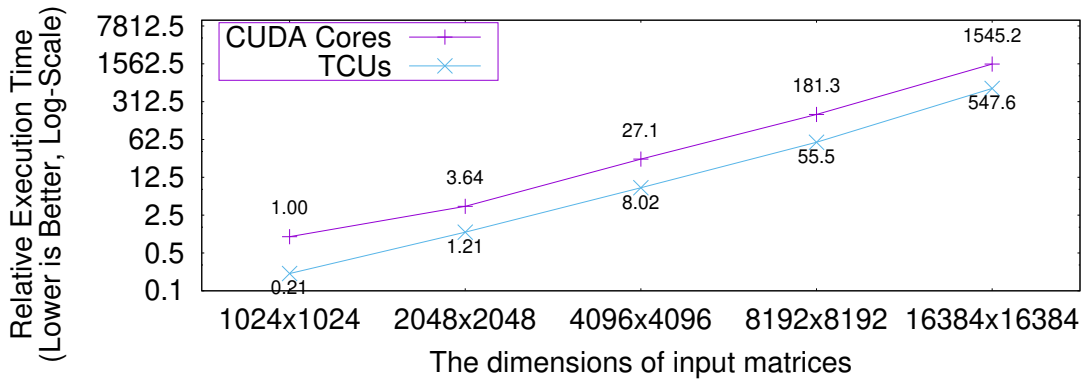


Figure 3.3: The performance of performing matrix multiplications using conventional CUDA cores and TCUs.

cessors (SM) as Figure 3.2 depicts. Figure 3.3 compares the latency of multiplying matrices with different sizes, ranging from 1024×1024 inputs matrices to 16384×16384 ones, using conventional vector processing units (CUDA cores) and TCUs, on NVIDIA's RTX 3090 GPU. The results show that TCUs consistently outperform CUDA cores by up to a 5× speedup. By translating the latency to TFLOPs, we measured a peak of 63 TFLOPs on TCUs and 19 TFLOPs using mixed precision on CUDA cores only.

Despite the significant speedup in matrix operations, TCUs still have limited precision drawbacks seen in other AI/ML accelerators in a way that TCUs only support at most 16-bit numbers as inputs and incur additional overhead in casting data into the desired 16-bit formats. Being separated functional units within an SM and the nature that an SM can only perform a single type of operations simultaneously, a compute kernel can activate either conventional vector units or TCUs, but not both of them due to the power constraints and the hardware architecture. Therefore, if programmers do not specifically enable TCUs and rewrite algorithms to perform matrix multiplications, a GPU program

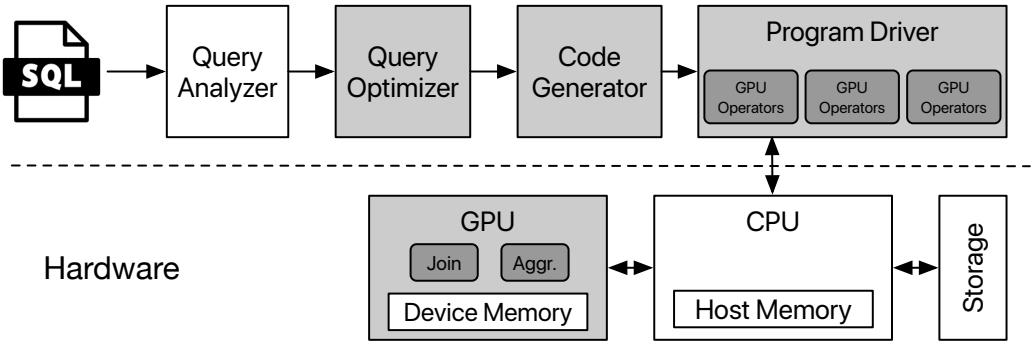


Figure 3.4: Typical GPU-accelerated database architecture.

cannot automatically take advantage of TCUs. Instead, it wastes the rich speedup that the TCUs can provide.

3.2.2 GPU-accelerated Database System Architecture (GPUDB)

Prior to the introduction of TCUs in GPU architectures, database systems have exploited the potential of using the massive amount of vector processing units within GPUs to accelerate query processing [25, 16, 267, 239]. The rich thread-level parallelism from these vector processing units delivers better performance on easily parallelizable operations (e.g., arithmetic computation). Figure 3.4 shows the architecture of a typical GPUDB system that Yinyang DB (YDB) [14, 267] and GPUQP [89] adopt. Upon receiving a query, the GPU-accelerated DB will go through the following stages: (1) Query plan generation: the query parser translates SQL query into query plan tree and the query optimizer analyzes the costs and benefits of query plans to determine the most efficient implementation (i.e., the cheapest plan) as the physical query plan. (2) Code generation: the query engine is in charge of the query execution flow by generating the back-end system-level code (e.g.,

program driver) that maps the selected query plan to utilize CPU and GPU cores. According to the type of target queries, different GPU kernels are implemented to execute relational database operators. (3) Data movements: data movements involve loading table data to the host main memory from back-end storage, moving essential data from the host main memory to GPU device memory and copying results back to the host main memory.

In the aforementioned database system architecture, data movement between GPU and CPU usually dominates the execution time [89] and cancels out the performance gain in the computation part. Therefore, GPU database architecture should make full use of an in-memory technique such as keeping all tables in GPU RAM [72] to mitigate the I/O bottleneck. There is no common-use GPU algorithm suitable for all database systems; the challenge is to identify which operators can leverage the GPU and combine it with traditional database query processing. Additionally, the data storage format also affects the performance of data movement. Due to the GPU memory access pattern, column-store [1, 2, 72] helps to exploit coalesced memory as well as reduce data volume going through the PCIe bus by only sending the needed data.

3.2.3 The Missing Opportunities of GPU Databases in TCUs

Before the emergence of TCUs, conventional wisdom assumed that matrix multiplication is an inefficient operation. Therefore, state-of-the-art GPUDB systems are designed in favor of vector processing, yet completely avoid the usage of matrix multiplications. Without redesigning application algorithms and data layout, existing GPUDB systems cannot reap the benefits of TCUs.

```
-- Matrix multiplication query:
SELECT A.col_num, B.row_num, SUM(A.val * B.val) as res
FROM A, B
WHERE A.row_num = B.col_num
GROUP BY A.col_num, B.row_num;
```

Figure 3.5: Example matrix multiplication query.

The query in Figure 3.5 provides an example of how an existing GPUDB misses the potential of using TCUs. The result of this query is essentially a list of triples of (*row_num*, *col_num*, *val*) with unique combinations of *row_num*, *col_num* and the *val* in each triple is the sum of the pairwise multiplications on *val* fields from a record in table A with its *row_num* matching another record's *col_num* from table B. This is essentially an SQL query that performs matrix multiplication on elements from two tables A and B. This query can be implemented through one matrix multiplication if we can layout the matching elements in matrices appropriately.

However, conventional GPUDB query processing algorithms are designed at the operator level with each operator as a kernel function running on GPUs. To execute the above query, conventional GPUDB uses operators to build hash tables for A and B, scanning both tables, performing `HashJoin`, and aggregating the final result. Among these GPU operators, `HashJoin` where performs join operation in a pairwise, vectorized fashion to find matching tuples between two hash tables usually takes the most time during the query execution.

The aggregation operator is second to `HashJoin`, which is also time-consuming in accumulating the computation result using vector operations. As the above computation only requires vector inner-products, the generated GPU kernel code will never enable TCUs.

3.3 TCU-accelerated query patterns

TCUs can potentially improve the performance of an analytic query by executing (parts of) the query as matrix multiplication. Next, to achieve this goal, we start by identifying a number of query patterns that TCUDB can execute as matrix multiplications.

3.3.1 Two-way natural join

The first supported query pattern is the simple 2-way join. For example, given two tables `A` and `B` with two attributes (`ID`, `Val`), consider the following query:

```
1 -- Q1:
2 SELECT A.Val, B.Val
3 FROM A, B
4 WHERE A.ID = B.ID;
```

To process this query as a matrix operation, we first need to convert the two tables into a matrix format. Suppose table `A` contains n tuples $\{a_1, \ldots, a_n\}$ and table `B` contains m tuples $\{b_1, \ldots, b_m\}$ where each a_i and b_i are unique row IDs. Let `dom(A.ID)` and `dom(B.ID)` be the domains of the `ID` column of `A` and `B` respectively. Let `dom(ID)` to be the union of the two domains `dom(A.ID)` $\cup$ `dom(B.ID)` having k distinct values $\{v_1, \ldots, v_k\}$. To compute

the join, we construct a $n \times k$ matrix $\texttt{mat(A)}$ and a $m \times k$ matrix $\texttt{mat(B)}$ where

$$\texttt{mat(A)}_{ij} = 1 \text{ if } a_i.\texttt{ID} = v_j, \text{ otherwise } 0 \ ;$$

$$\texttt{mat(B)}_{ij} = 1 \text{ if } b_i.\texttt{ID} = v_j, \text{ otherwise } 0 \ .$$

The result of the join $\texttt{A} \bowtie \texttt{B}$ is then the n by m matrix

$$\texttt{C} = \texttt{mat(A)} \times \texttt{mat(B)}^{\texttt{T}}.$$

It is easy to show that a tuple (a_i, b_j) is in the join result if and only if $\texttt{C}_{ij} > 0$.

Alternatively, when the domains $\texttt{dom(A.Val)}$ and $\texttt{dom(B.Val)}$ are small, one can also construct $\texttt{mat(A)}$ and $\texttt{mat(B)}$ as the adjacency matrices where $\texttt{mat(A)}_{ij} = 1$ if $(u_i, v_j) \in \texttt{A}$ (and respectively for $\texttt{mat(B)}$) otherwise 0. The number of rows of $\texttt{mat(A)}$ and $\texttt{mat(B)}$ will be $|\texttt{dom(A.Val)}|$ and $|\texttt{dom(B.Val)}|$ respectively.

Note that in this query pattern, the single attributes A.ID, A.Val, B.ID and B.Val can be generalized to sets of multiple attributes. The attribute sets $*.\texttt{ID}$ and $*.\texttt{Val}$ can potentially overlap thus it is general enough to cover all cases of 2-way natural join.

3.3.2 Multi-way joins

Next, we extend the querying capability with matrix multiplication to multi-way joins. Consider the following snippet of a 3-way join query where the 3 input tables are $\texttt{A}(\texttt{ID}_1, \texttt{Val})$, $\texttt{B}(\texttt{ID}_1, \texttt{ID}_2, \texttt{Val})$, and $\texttt{C}(\texttt{ID}_2, \texttt{Val})$ respectively.

```
1 -- Q2:
2 SELECT A.Val, B.Val, C.Val
3 FROM A, B, C
4 WHERE A.ID_1 = B.ID_1 AND B.ID_2 = C.ID_2;
```

As in conventional join processing, we assume a join order of $\mathtt{A} \rightarrow \mathtt{B} \rightarrow \mathtt{C}$. To evaluate this join, one needs to (1) first compute $\mathtt{A} \bowtie \mathtt{B}$ as $\mathtt{mat(A)} \times \mathtt{mat(B)}^{\mathtt{T}}$, (2) convert the resulting n by m matrix back to table format and (3) compute the join with table $\mathtt{C}$ as a second matrix operator. By repeating step (2) and (3) to convert intermediate results to tables, we can generalize this algorithm from 3-way joins to multi-way joins.

To avoid unnecessary data transfer from GPU memory to the host, in step (2), one can perform the matrix-table conversion with a CUDA-enabled $\mathtt{nonzero}(\cdot)$ operator [189]. Formally, given a matrix M, $\mathtt{nonzero}(M)$ computes $\{(i,j)|M_{ij} > 0\}$. Next, to perform the second join, let

- n' be the size of $\mathtt{nz} = \mathtt{nonzero}(\mathtt{mat(A)} \times \mathtt{mat(B)}^{\mathtt{T}})$,
- m' be the size of table $\mathtt{C} = \{c_1, \ldots, c_{m'}\}$ and
- k' be the size of $\mathtt{dom}(\mathtt{B.ID_2}) \cup \mathtt{dom}(\mathtt{C.ID_2}) = \{u_1, \ldots, u_{k'}\}$.

We denote by $\mathtt{nz}_i$ the i-th pair of the $\mathtt{nz}$ array. Next, we construct a n' by k' matrix $\mathtt{mat(AB)}$ and a m' by k' matrix $\mathtt{mat(C)}$ where

$$\mathtt{mat(AB)}_{ij} = 1 \text{ if } b_{i'}.\mathtt{ID_2} = u_j \text{ for } \mathtt{nz}_i = (\_, i'), \text{ otherwise } 0;$$

$$\mathtt{mat(C)}_{ij} = 1 \text{ if } c_i.\mathtt{ID_2} = u_j, \text{ otherwise } 0.$$

The result of the 3-way join is then $\mathtt{mat(AB)} \times \mathtt{mat(C)}^{\mathtt{T}}$.

There is an exception case where the intermediate matrix-table conversion can be omitted. When $\mathtt{B.Val} = \emptyset$ (i.e., relation $\mathtt{B}$ is projected out entirely), the result of the join can be simplified as

$$\mathtt{mat(A)} \times \mathtt{mat(B)}^{\mathtt{T}} \times \mathtt{mat(C)}^{\mathtt{T}}$$

where $\texttt{mat}(\texttt{B})$ is a k by k' matrix constructed as $\texttt{B}_{ij} = 1$ if $(v_i, u_j) \in \texttt{B}$ otherwise 0.

Similar to the 2-way join case, the method can be generalized to multi-way joins consisting of multiple join and/or return attributes.

3.3.3 Group-by aggregates over joins

A simple yet useful extension of the above two query patterns with joins is to add group-by aggregates. For example, over the same schema $(\texttt{ID}, \texttt{Val})$ of the previous 2-way join case:

```
1 -- Q3:
2 SELECT SUM(A.Val), B.Val
3 FROM A, B
4 WHERE A.ID = B.ID
5 GROUP BY B.Val;
```

A naive method to evaluate this query is to first evaluate the natural join in the TCU-optimized manner, convert the matrix result to the table format, and then compute the group-by and SUM aggregate with CPU or GPU-based methods. We propose the following method that avoids any unnecessary intermediate computation via 2 matrix operations. First, we construct the two input matrices. For the matrix dimensions, we let

- n be the size of $\texttt{A}$,
- m be the size of $\texttt{dom}(\texttt{B.Val}) = \{u_1, \ldots, u_m\}$, and
- k be the size of $\texttt{dom}(\texttt{A.ID}) \cup \texttt{dom}(\texttt{B.ID}) = \{v_1, \ldots, v_k\}$.

We construct a n by k matrix $\texttt{mat(A)}$ and a m by k matrix where

$$\texttt{mat(A)}_{ij} = a_i.\texttt{Val} \text{ if } a_i.\texttt{ID} = v_j, \text{ otherwise } 0;$$

$$\texttt{mat(B)}_{ij} = 1 \text{ if } (u_i, v_j) \in \texttt{B}, \text{ otherwise } 0.$$

Next, the query result can be computed as

$$\mathbf{1}^{1\times n} \times \texttt{mat(A)} \times \texttt{mat(B)}^{\texttt{T}}$$

where $\mathbf{1}^{1\times n}$ is an $1 \times n$ matrix consisting of only ones. We can show the following:

Lemma 1 *(Q3, informal) For every tuple* $(a_i^{\texttt{sum}}, b_i)$ *and for* $M = \mathbf{1}^{1\times n} \times \texttt{mat(A)} \times \texttt{mat(B)}^{\texttt{T}}$, $(a_i^{\texttt{sum}}, b_i)$ *is in the query result of Q3 if and only if* $M_{i,1} = a_i^{\texttt{sum}}$.

Intuitively, we leverage the first multiplication with $\texttt{mat(B)}^{\texttt{T}}$ to compute the join. By filling the input matrices $\texttt{mat(A)}$ with actual values instead of 0's or 1's, we keep track of those values in the intermediate matrix product $\texttt{mat(A)} \times \texttt{mat(B)}^{\texttt{T}}$. The multiplication with $\mathbf{1}^{1\times n}$ then serves as a reduction operator that sums up all columns of $\texttt{mat(A)} \times \texttt{mat(B)}^{\texttt{T}}$.

In addition to `SUM`, we are able to apply the same method to support the `COUNT` and `AVG` aggregate functions. For `COUNT`, when we construct $\texttt{mat(A)}$, we simply need to set $\texttt{mat(A)}_{ij}$ to 1 for $a_i.\texttt{ID} = v_j$ (instead of $a_i.\texttt{Val}$). We can obtain `AVG` by dividing `SUM` by `COUNT`.

For aggregate queries without `GROUP BY`, such as

```
1 -- Q4:
2 SELECT SUM(A.Val * B.Val)
3 FROM A, B
4 WHERE A.ID = B.ID;
```

we set $\texttt{mat(A)}_{ij} = a_i.\texttt{Val}$ for $a_i.\texttt{ID} = v_j$ and $\texttt{mat(B)}_{ij} = b_i.\texttt{Val}$ for $b_i.\texttt{ID} = v_j$ and compute the sum as $\texttt{mat(A)} \times \texttt{mat(B)}^{\texttt{T}} \times \mathbf{1}^{m \times 1}$ with an additional reduction by multiplying $\mathbf{1}^{1 \times n}$.

3.3.4 Other supported operators

The above query patterns can also be extended with the `ORDER BY` clause to sort the results in ASC/DESC order by a certain column. Instead of sorting after the multiplication operators, we preserved the specified order in the input matrices (e.g., `mat(A)` and `mat(B)`) so that the result matrix is naturally sorted.

Another class of supported query pattern is the non-equi join such as:

```
1 -- Q5:
2 SELECT A.Val, B.Val
3 FROM A, B
4 WHERE A.ID < B.ID;
```

We can compute this query by slightly adjusting the translation for Q1 by setting $\texttt{mat(A)}_{ij} = 1$ for $a_i.\texttt{Val} < v_j$. The same method applies to the other comparison operators $\{<, >, \leq, \geq, \neq\}$.

Last but not least, for the query pattern that is of the semantics of matrix multiplication as Figure 3.5 shows, we can directly map the query to the corresponding matrix operation.

Beyond the supported patterns. For queries that do not match exactly with any of the supported query patterns, as part of the query optimization workflow (Figure 3.6), TCUDB relies on pattern matching to identify subqueries that can be TCU-accelerated from the input query's AST. We note that there are common subqueries that are beyond

the expressiveness of the TCU platform, such as aggregation with MIN/MAX or arithmetic operators such as addition and division. The limited expressiveness is mainly due to NVIDIA's current TCU programming interface which only supports matrix multiply-accumulate. However, since the underlying hardware is powerful enough to perform the aforementioned operators, we anticipate a more flexible programming interface in the future so that TCUDB can support a wide range of query patterns.

3.4 TCUDB: A TCU-Accelerated DB Engine

To leverage TCUs for queries in relational database systems, this paper presents TCUDB, a DB engine that identifies, optimizes, evaluates and implements aforementioned query patterns in Section 3.3. This section provides an overview of the design of TCUDB's extensions and discusses the optimizations on a TCU-accelerated query plan.

3.4.1 Overview

TCUDB implements the system architecture in Figure 3.1 to execute queries on TCUs using the following major components.

Query Optimizer In a system with TCUs presented, the query plan in exercising a query is from either (1) the most efficient TCU-accelerated query plan or (2) the most efficient conventional CPU/GPU-based plan, depending on which one can deliver the lowest cost (i.e., the shortest end-to-end latency). TCUDB leverages existing infrastructure in GPUDB to evaluate the second option but extends the query optimizer in creating, optimizing and evaluating the latency of TCU-accelerated query plans.

Program Driver TCUDB extends the program driver to additionally contain a set of library functions that implement operators mentioned in Section 3.3 using TCUs. These functions invoke NVIDIA's CUDA C++ Warp Matrix Multiply and Accumulate (WMMA) or cuBLAS API functions to achieve the series of computation that each operator requires. These operators also present interfaces in various data types to support the demand for the most efficient query plan.

Code Generator If TCUDB selects a TCU-accelerated query plan to exercise an incoming query, the code generator will rewrite the query as C code and dynamically compile the code to execute the selected query plan. The TCUDB code extension is responsible for creating the input matrices, calling operator functions in corresponding data types and remapping the output from the operator outcome.

Among these three intensively extended modules, the query optimizer is the most critical component as it serves as the core controlling the use of TCUs as well as code generation for queries. In the rest of this section, we will focus on the query optimizer.

3.4.2 TCUDB query optimizer

Figure 3.6 shows the workflow of the TCUDB query optimizer. The optimizer takes a subquery from the query AST as input and performs a series of tests to determine whether the subquery should be executed with TCU and how. The optimizer first checks if the subquery falls in one of the supported query patterns. Next, it performs the data range feasibility test (Section 3.4.2) to decide if particular data types can provide sufficient precision to the query. After that, the input tables may also result in matrices

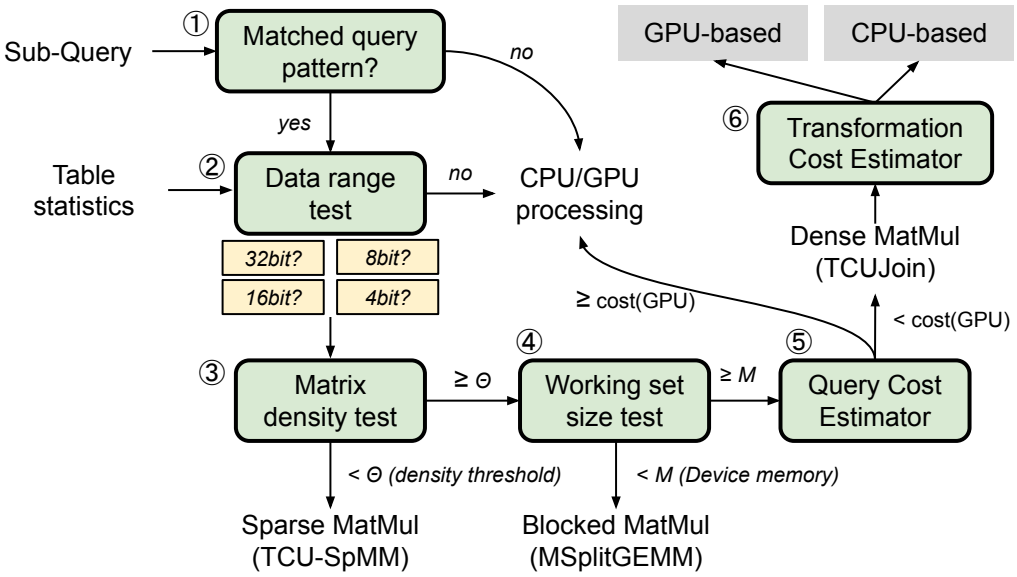


Figure 3.6: The workflow of the TCUDB query optimizer.

too large to fit in the GPU's device memory or sparse matrices for which dense multiplication algorithm is sub-optimal. For these cases, the optimizer estimates the working set sizes and matrix density from statistics pre-computed from input tables. TCUDB applies blocked matrix multiplication (MSplitGEMM, Section 3.4.2) and sparse matrix multiplication (TCU-SpMM, Section 3.4.2) respectively. Finally, the optimizer estimates the query execution cost with TCU and tests whether the cost is lower than the estimated cost with CPU/GPU (Section 3.4.2). If any of the tests fail, TCUDB falls back to the standard CPU or GPU-based query execution.

Note that the query cost estimator needs to take into account the data transformation cost which consists of both computation and data movement overhead. If the original table size plus the working set size fits in the device memory, TCUDB can transform tables into matrix format within GPU to save the overhead of transforming data within CPU and moving large matrices into the GPU device.

Feasibility Test

Even though a query contains patterns matching identified patterns in Section 3.3, a query may still be unfeasible for TCUs due to the limitations of TCUs in input precision and data types. If applying TCUs would result in loss of precision or lead to unwanted outcomes, TCUDB should not use TCUs to evaluate the incoming query.

Therefore, TCUDB must perform a feasibility test for each query that contains qualified patterns by evaluating the input data ranges, identifying the most compact inputs/outputs data types and estimating the working set sizes for operators within a query. To facilitate this process, TCUDB adds metadata to each database table to contain three values for each column, including (1) the minimum value, (2) the maximum value, and (3) the number of distinct values.

If the operator works with the numerical computation on the input data values directly, TCUDB first uses the minimum and maximum values along with the raw data types of the operator's input data. If the input data can be represented by TCU-compatible data types, including 16-bit half floating-point (`half`), 8-bit integers (`int8`), and 4-bit integers (`int4`), this stage will also determine the most compact data type. However, if the dataset cannot leverage any TCU-compatible data type, the feasibility test will suggest that the

system not use TCUs in the incoming query. The database system can use other available options (e.g., a CPU-based or a pure GPU-based query engine) instead.

The number of records, the number of distinct values and the maximum/minimum values of each column also help the feasibility test to identify the case where the result value can surpass the range of 16-bit numbers and potentially lead to errors. Let m_1 represents the maximum of the maximum value and the absolute value of the minimum value within a column of n elements in one of the input matrix and that of a row with n elements is m_2 for another input matrix, the feasibility test can conservatively estimate the maximum value in the resulting matrix as $m_1 \times m_2 \times n$. If the maximum result value falls beyond the range of TCUs 16-bit number ranges, TCUDB will use query executors based on other hardware components instead.

Cost estimation of query plans

The cost of a TCU-accelerated operator contains:

(1) the data transformation cost `DT_op` which equals the latency for creating input matrices to perform the TCU-accelerated operators from the input tables,

(2) the data movement overhead `DM_op` for copying data between the host main memory or data storage to the GPU's device memory, and

(3) the computation time `CT_op`, the actual running time that the TCUs spend on executing the generated TCU code.

Depending on the estimated working set size of the query, the data transformation process of TCUDB can take place using the CPU or the GPU. The costs of `DT_op` and `DM_op` vary according to the approach.

CPU-based data transformation The most general data transformation approach in TCUDB uses the host main memory and CPU to prepare inputs for the designated TCU-accelerated operator. This approach fills input matrices for a TCU-accelerated operator using methods described in Section 3.3 and works regardless of the estimated working set size of the query.

Consider the example of the 2-way natural join. To create the input matrices for an operator, TCUDB typically needs to scan through qualified/valid records for the operator and convert the values into the desired matrix representations. The data transformation cost is linear to the number of qualified/valid records. Let A and B be two input tables (which can also be intermediate results from subqueries) of size m and n respectively. Assume the throughput of the host system in scanning the raw data is a constant α. If their matrix representations `mat(A)` and `mat(B)` are not yet created, the scan operator will take $\texttt{DT\_op} \approx \alpha \cdot (m + n)$ in transforming input data to the desired matrices. The cost can also be $\alpha \cdot m$ or $\alpha \cdot n$ if either matrix is already created.

In this approach, the data movement overhead is controlled by (1) the volume of transformed matrices or input data and (2) the available bandwidth between the GPU and the host processor denoted by $\texttt{Bandwidth}_{\texttt{GPU/host}}$. If A is of dimension $M \times K$ with `type_A` and B is of dimension $K \times N$ with `type_B`, the data movement cost can be estimated by

$$\texttt{DM\_op} \approx \frac{MK \cdot \texttt{sizeof(type\_A)} + NK \cdot \texttt{sizeof(type\_B)}}{\texttt{Bandwidth}_{\texttt{GPU/host}}}. \tag{3.1}$$

GPU-assisted data transformation To optimize the data transformation overhead `DT_op`, the query plan may perform the data transformation on the GPU to leverage its massive parallelism to convert thousands of pairs of values simultaneously into matrix for-

mat. In other words, we can take advantage of the GPU's parallelism to speed up the data transformation operation as well as avoid the additional data movement that copies the transformed matrix from the host memory to the GPU device memory. In contrast to the CPU-based approach, the data movement occurs before the data transformation in the GPU-assisted approach as the raw data must be present in the GPU's device memory in advance for the transformation to begin. Therefore, TCUDB can only use GPU-assisted data transformation when both the estimated working set size and the volume of necessary raw data (e.g., columns from the selected table) for transformation can fit in GPU's device memory. Leveraging the same 2-way natural join example, TCUDB can estimate the corresponding `DM_op` using Equation 3.2 as:

$$\texttt{DM\_op} \approx \frac{M \cdot \texttt{sizeof(type\_A)} + N \cdot \texttt{sizeof(type\_B)}}{\texttt{Bandwidth}_{\texttt{GPU/host}}}. \tag{3.2}$$

where M and N are the numbers of elements in the raw data columns of the joined columns and (`type_A`) and (`type_B`) are the raw data types of both columns before transformation.

In terms of `DT_op`, the GPU-based scan operator still takes $\approx \alpha \cdot (m+n)$ operations in transforming input data to the desired matrices – but a GPU can perform `p` of these in parallel if the GPU has `p` vector processors available. In modern GPU architectures, `p` is typically more than 2,000. The `DT_op` in GPU-assisted approach is estimated as $\texttt{DT\_op} \approx \frac{\alpha \cdot (m+n)}{p}$. Notice that the GPU-based approach needs to move raw data in Equation 3.2, TCUDB still needs to evaluate the summation of `DM_op` and `DT_op` to determine the most appropriate data transformation method.

Computation cost Finally, the dimensions of the transformed input matrices also determine the TCU computation time. Using the number of records, the number of distinct

values and the most compact data type derived from the feasibility test, TCUDB can estimate the required device memory and the density of input matrices for the operator. Based on the estimation, TCUDB can potentially take three different approaches in performing an operator.

(1) If all inputs and outputs fit within the device memory, TCUDB simply needs to copy all inputs into the device memory and invokes the matrix multiplication function once.

(2) In case the working set size exceeds the available device memory, TCUDB's query plan will need to apply the blocked and pipeline matrix multiplication algorithm [130, 279] to move parts of input and output data as well as perform matrix multiplications block-by-block. (Section 3.4.2)

(3) If the densities of input matrices are lower than a certain threshold (an architecture-dependent value), TCUDB will use sparse matrix multiplications instead. (Section 3.4.2)

Since each pair of values in input matrices requires 2 operations for multiplication and accumulation, the computation time in the simplest case where all input matrices fit in the device memory can be estimated by

$$\texttt{CT\_op} \approx MNK \times \frac{2}{\texttt{peak\_TCU\_TFLOPS}} \tag{3.3}$$

where `peak_TCU_TFLOPS` is the TCUs' peak number of floating-point operations per second (FLOPS). If the query results in inputs larger than device memory, TCUDB still leverages Equation 3.3 to estimate the cost but replaces `peak_TCU_TFLOPS` with the measured FLOPS from the blocked/pipelined matrix multiplications. For the cases where input matrices are sparse, TCUDB estimates the computation costs not only using the FLOPS from our sparse matrix multiplication implementation but also multiplying the cost by the density of inputs.

The final cost estimation is then the summation of the above three terms $\texttt{DT\_op} + \texttt{DM\_op} + \texttt{CT\_op}$. TCUDB then compares this estimated cost with the estimated cost of the other CPU/GPU-based operators to decide whether to use TCUs. TCUDB obtain the most up-to-date estimations for $\texttt{Bandwidth}_{\texttt{GPU/host}}$ and `peak_TCU_TFLOPS` by checking the execution time of previous queries.

Note that there can be more than one TCU-accelerated plan because the system can choose a higher or lower-precision data type, which can change the decision of whether to perform transformation operator within the GPU or not.

Handling large datasets.

Due to the limited device memory capacity (e.g., 24 GBs in our case), the input matrices of TCUDB's operators cannot fit in the GPU's device memory if the datasets are extremely large and dense. Once TCUDB catches such a case during the feasibility test, TCUDB will consider applying a blocked matrix multiplication algorithm for the corresponding query operators. The blocked matrix multiplication algorithm works by fetching a submatrix from the system main memory as a multiplicand, gradually fetching other same-sized submatrices as the multiplier, and aggregating the result to the corresponding submatrix in the result matrices.

TCUDB's implementation of blocked matrix multiplication extends MSplitGEMM [279] to support blocked matrix multiplications using TCUs. Both TCUDB's implementation and MSplitGEMM exploit pipeline parallelism by creating multiple streams in fetching input submatrices, performing matrix multiplication and accumulation, and writing back results simultaneously. TCUDB's implementation uses TCUs for matrix multiplication and accu-

mulation instead of conventional GPU cores. During the periodical microbenchmark tests, TCUDB also performs a series of tests to figure out the optimal size of submatrices that balances the latency of each stage in the pipeline to maximize the computation throughput. The measured throughput using these optimal parameters will also be used as the metrics for evaluating the costs of large and dense inputs in Section 3.4.2.

Handling sparse matrices.

Due to the current capability of TCU hardware in handling sparse matrices, conventional TCU operators that assume dense matrices as their inputs may not always outperform a GPU plan when the input matrices to a TCU-accelerated operator are very sparse. Therefore, TCUDB implements a TCU-accelerated sparse matrix multiplication (TCU-SpMM) operator that

- transforms an input into a compressed sparse row matrix format (CSR)
- partitions an input matrix into 16×16 submatrices,
- skips submatrices containing all 0s,
- multiplies the rest using TCUs and accumulates results [268].

By doing so, the TCU-SpMM operator can still leverage TCU's computation power but on a much smaller number of submatrices pairs when the input matrices are large and sparse.

To determine whether a TCU-SpMM-based plan should replace the dense multiplication plan, TCUDB needs to estimate the cost similar to the regular cases with dense matrices. We estimate the total cost by multiplying the estimated dense operator cost by

the inputs' densities. In addition, the TCU-SpMM-based operator requires scanning inputs to construct/partition a matrix and filter those all-0-submatrices. TCUDB estimates this part of the cost with a simple linear function with respect to the input size.

Finally, the query optimizer of TCUDB still needs to evaluate plans using the GPU-based HashJoin cost model [267], in particular sparse matrix multiplication on conventional CUDA cores to determine whether a TCU-SpMM-based plan is more efficient.

3.5 Experimental Results

Leveraging TCUs' capabilities in optimizing matrix algebra, TCUDB delivers up to 14× speedup over a conventional GPU-based DB engine for the sample queries that Section 3.3 describes. Inspired by the result, we experimented with TCUDB in real-world application query workloads with inputs as large as 24 GBs. In summary, TCUDB achieves up to 7.52× speedup in matrix multiplications, up to 3.96× speedup for analytic queries in the star schema benchmark, up to 288× speedup in entity matching queries, and up to 4.22× speedup for the core of the PageRank algorithm. The comparison of TCUDB performance on different GPU architectures also reveals the strong potential of TCU-accelerated DB engines in the future.

3.5.1 Experimental Methodology

We conducted experiments on a machine with an Intel Core i7-7700K processor, 32 GB DDR4 DRAM. The processor contains 4 cores and each processor core runs at 4.2 GHz by default. The GPU in our experiments is an NVIDIA GeForce RTX 3090 GPU based

on Ampere architecture. This GPU contains 24 GB GDDR6X device memory and 328 Tensor Cores and attaches to a PCIe 3.0 x16 slot. The TCU-accelerated operator library in TCUDB is implemented using a NVIDIA CUDA Toolkit 11.2. The system runs a Linux 4.15.0 kernel with the NVIDIA driver version in 460.32.03. We compared TCUDB with a state-of-the-art GPU execution engine for warehouse-style queries, YDB [267] and a pure CPU-based execution engine, MonetDB [21], as reference designs.

3.5.2 Microbenchmark

To allow query optimizers to select the right query plans, the database engine must obtain samples of executing workloads using TCU-accelerated operations. Upon installing TCUDB in the system or when the system detected any change in hardware configurations, TCUDB will perform a one-time sampling process that runs a set of microbenchmark workloads to collect critical timing information for query optimizations.

During the sampling process, TCUDB will execute three main queries, Q1, Q3 and Q4 from Section 3.3, with various-sized, random-generated input datasets. TCUDB does not evaluate Q2 and Q5 as they are essentially combinations of other queries. The sampling process also helps us to classify the cases where TCUDB is superior to the conventional GPU-accelerated engine and identify the source of performance gain/loss in TCUDB. With large system main memory and aggressive file system caching by operating systems as well as the underlying high-performance NVMe SSD, we have not observed significant disk load time in each DB engine's initialization phase.

As MonetDB is a full-fledged system, we excluded the additional steps/overheads by measuring only the time to execute the physical plan for a fair comparison. (We use the

"–timer=performance" option and disable the resulting output to report the runtime part only.)

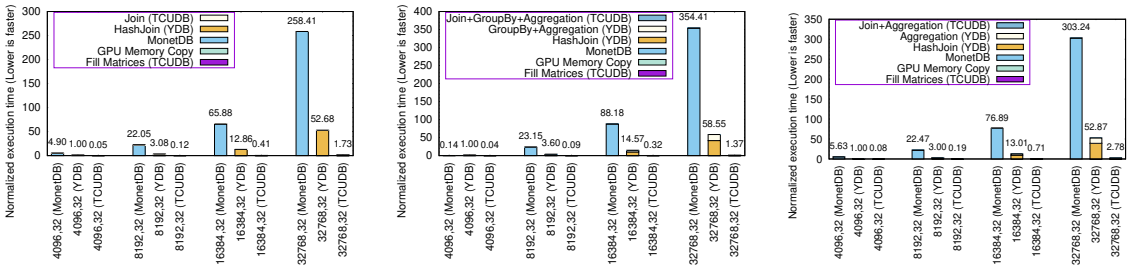


Figure 3.7: The relative execution time of running (a) Q1, (b) Q3, and (c) Q4 with various number of records and 32 distinct values in the target attribute on TCUDB, YDB, and MonetDB.

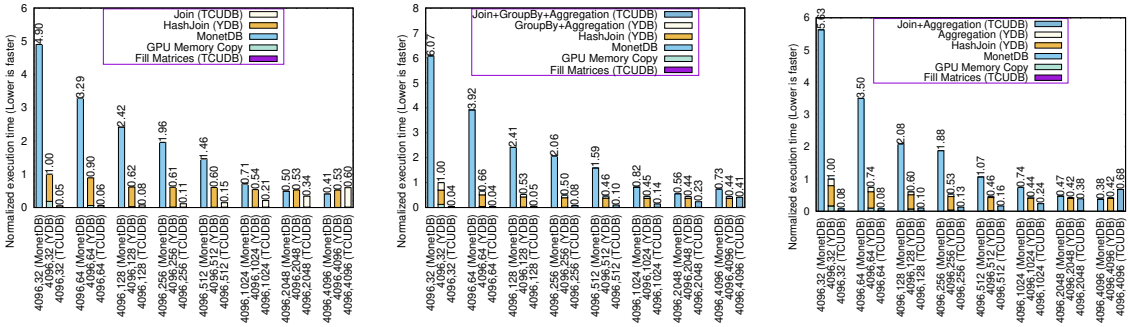


Figure 3.8: The relative execution time of running (a) Q1, (b) Q3, and (c) Q4 with 4096 records and various distinct values in the target attribute on TCUDB, YDB, and MonetDB.

Figure 3.7 and Figure 3.8 present a subset of microbenchmark results from the sampling process on the default testbed described in Section 3.5.1. We label the x-axis

of each sample in this figure with two parts in the configuration. The first part is the parameters for the query, M, K and N, that represent the sizes of the input matrices for each evaluated operator where one matrix has the dimension of $M \times K$ and the other is $K \times N$. To save space, we only present the cases when $M = N$ and label each configuration with their values of M and K as M, K in these figures. The second part is the DB engine (i.e, TCUDB, YDB, or MonetDB). The vertical axis in each figure shows the aggregated execution time in each step of running these queries, normalized to the total time when running the same query using YDB, the conventional GPU-accelerated engine, with $M = N = 4096$ and $K = 32$.

Figure 3.7(a) shows the performance of Q1 for TCUDB, YDB and MonetDB from input sizes 4096 to 32768. Both TCUDB and YDB significantly outperform MonetDB for this query. TCUDB outperforms YDB in most configurations. The advantage of TCUDB is especially significant when datasets grow. TCUDB outperforms YDB by 14× for the case of (32768, 32) and 9.3× for (16384, 32), but only 1.18× for (4096,32). Observing the breakdown of execution time in Figure 3.7(a), we found the major speedup comes from the significant reduction of computation time from the TCU-accelerated join operator, despite the additional overhead in filling and transforming datasets into the desired matrices for TCUDB.

Figure 3.8(a) varies the number of distinct values that affect the sparsity of input matrices in Q1 for TCUDB's join operator. As the number of distinct values becomes larger, the performance advantage of TCUDB's join operator over YDB and MonetDB begins to shrink. Because the sizes of one dimension of both input matrices for the TCUDB join op-

erator in Q1 depends on the number of distinct values from the chosen attribute to perform matching, matching on an attribute with more distinct values will lead to computation on larger but sparse matrices. In contrast, YDB's and MonetDB's `HashJoin` algorithm produces smaller vectors as the chance (i.e., total number) of records sharing a single value reduces if the number of distinct values increases. Therefore, even though YDB's and MonetDB's `HashJoin` operator needs to work on more pairs of vectors, each pair of vectors have smaller dimensions. However, TCUDB's join operator still outperforms YDB and MonetDB in all cases until the number of distinct values reaches 4096. This profiling result suggests that TCUDB select a GPU-hash-join-based or sparse-matrix-based implementation if the density of input matrices is below 0.04% on our testbed.

Figure 3.7(b) presents the performance of running Q3 using TCUDB, YDB and MonetDB. Q3 evaluates the group-by and aggregations over join query. Unlike the conventional GPU-accelerated DB engine where group-by and aggregations are separate operations after the hash join, TCUDB can implement the whole Q3 using just one matrix multiplication. As a result, the execution time of using TCUDB of executing Q3 remains similar to executing Q1 when the input parameters are the same. However, YDB or MonetDB always have to perform the additional group-by operations and leads to a longer execution time than performing Q1 for the same inputs. Therefore, the performance advantage of TCUDB becomes more significant for Q3. For (32768, 32), TCUDB can outperform YDB by 45×.

When we increase the number of distinct values as in Figure 3.8(b), TCUDB becomes less advantageous, similar to the phenomenon in Q1. However, as TCUDB still uses single-matrix-multiplication-based Join/Aggregation/GroupBy operation to perform oper-

ations where YDB or MonetDB needs multiple-step HashJoin and GroupBy/Aggregation operators, TCUDB still outperforms YDB and MonetDB in all cases.

Figure 3.7(c) presents the relative execution time of Q4 on TCUDB, YDB and MonetDB. YDB and MonetDB will perform Q4 using *HashJoin* and then an aggregate query but without a group-by operator. Therefore, the overall execution time in each configuration of YDB and MonetDB is less than Q3 because of the elimination of group-by operator. However, again, TCUDB still implements this operator using single matrix multiplication on the transformed input matrices. Therefore, TCUDB achieves 19× speedup for (32768, 32).

As in Q1 and Q3, TCUDB becomes less advantageous when we increase the number of distinct values as in Figure 3.8(c). Because the amount of operations in YDB and MonetDB for Q4 is fewer than Q3, we still see TCUDB falls short when the number of distinct reaches 4096 and suggest an alternative plan for cases where input matrix densities are below 0.04%.

3.5.3 Analytic queries: Star Schema Benchmark

We evaluate the performance of TCUDB on the popular Star Schema Benchmark (SSB) [182], a benchmark suite modeling the data warehouse workloads. SSB is widely used in benchmarking analytic engines due to its realistic modeling of data warehousing workloads. The database form a star schema consisting of one fact table (`lineorder`) and four dimension tables (`supplier, customer, date` and `part`) connected to the fact table by foreign keys.

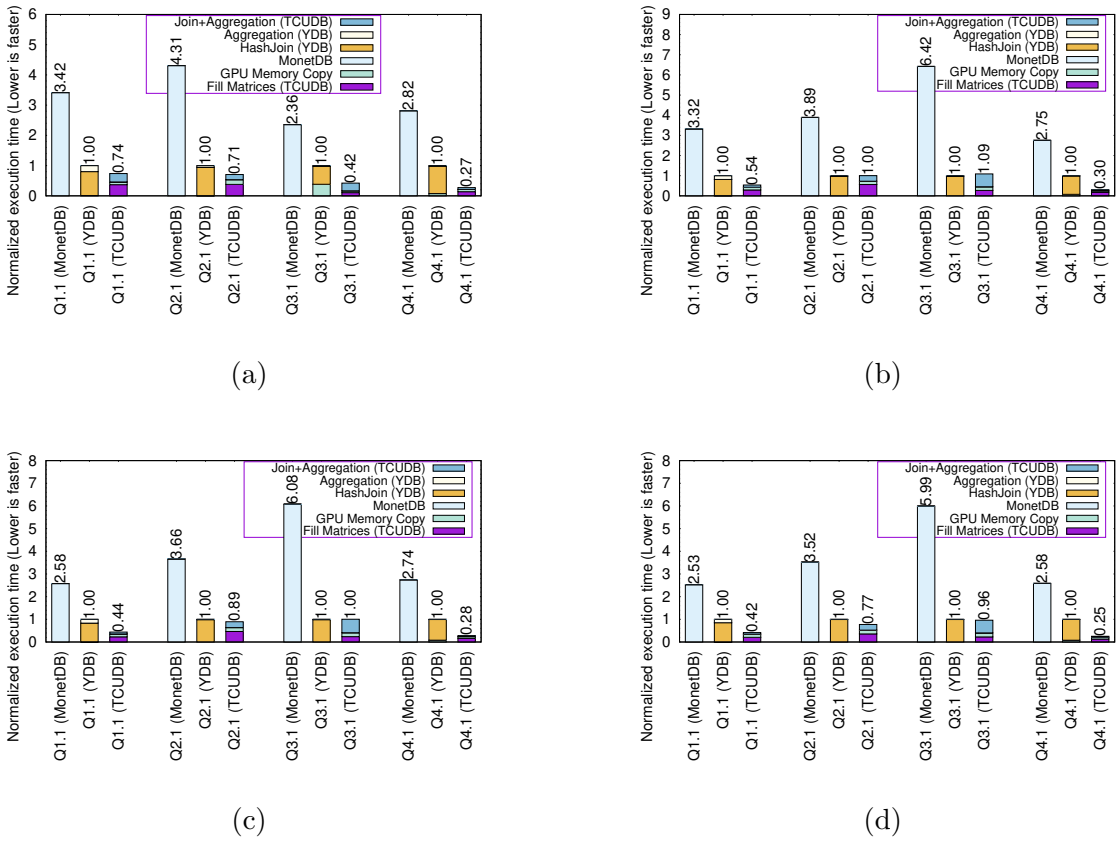


Figure 3.9: The relative runtime of star schema benchmark on TCUDB compared to MonetDB and YDB running the same query as the baseline with scaling factor (a) 1, (b) 2, (c) 4, and (d) 8.

The benchmark provides 13 queries in 4 flights. TCUDB supports all the 13 SSB queries. Figure 3.9 compares the performance of TCUDB, YDB and MonetDB in running SSB queries with scaling factors varying from 1 to 8 resulting in data sizes from 0.7GB to 5.6GB.

Figure 3.9 summarizes the results. TCUDB outperforms both YDB and MonetDB in all evaluated SSB workloads with up to 3.96× speedup when running Q4.1 with scaling factor as 8. Even with the worst performing SSB Q3.1, TCUDB still maintains the same

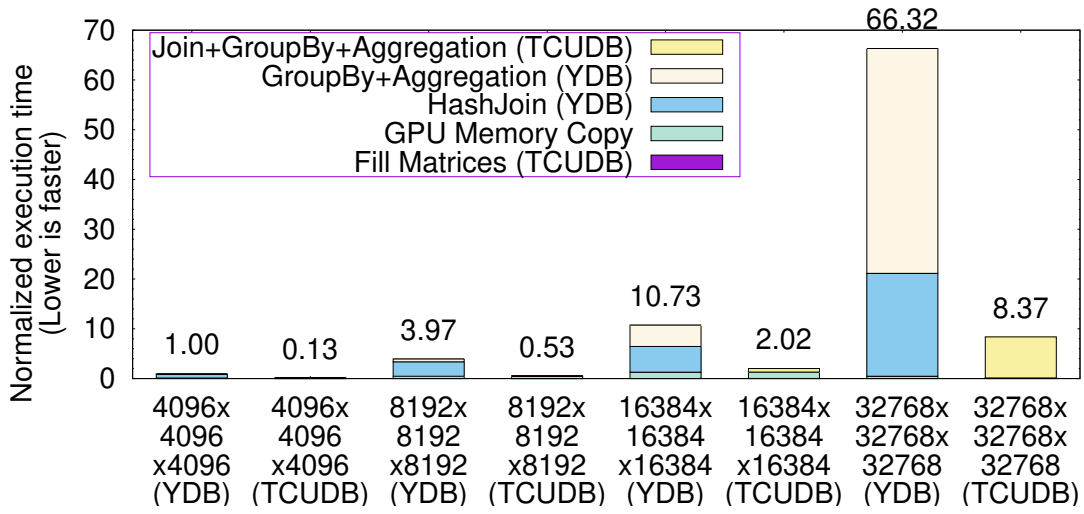


Figure 3.10: The relative execution time and breakdown of matrix multiplication query on TCUDB and YDB.

level of performance as YDB. These promising results show that TCUDB has the potentials of being integrated into real-world analytic engines.

3.5.4 Case studies: matrix multiplication, entity matching, and PageRank

In addition to individual operators, we also evaluated three representative use cases, matrix multiplication, entity matching and PageRank to demonstrate TCUDB's capabilities in handling intensive operations and large datasets.

Matrix Multiplication

Matrix multiplication was once considered inefficient for relational databases. With the help of hardware-accelerated matrix multiplications, TCUDB can make queries con-

taining complex linear algebra operations more efficient. We use a query in Figure 3.5 to demonstrate this use case. We create two tables A and B where each record in both tables has three attributes (row_num, col_num, val) as the input. We generate the synthetic dataset according to this schema with input matrices of dimensions up to 32768×32768 and data volume up to 24 GB, approximately 2.14 billion records.

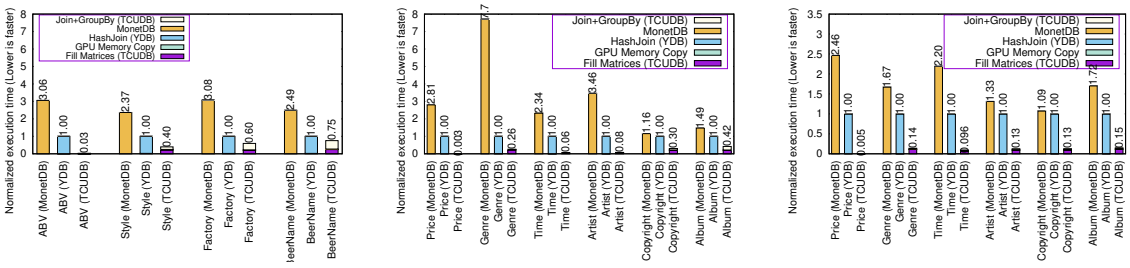
Figure 3.10 presents the relative execution time and breakdown of performing matrix multiplication on TCUDB and YDB, using YDB with each table containing 4096×4096 records as the baseline. We did not include MonetDB's result in these Figures as MonetDB cannot finish these queries within a reasonable amount of time and present MonetDB's results in Figure 3.10 would render the results of TCUDB and YDB invisible. When the dataset contains fewer than 16384×16384 records, the input matrices that TCUDB creates for the TCU's `Join` + `Aggregation` + `GroupBy` operator completely fit in the GPU's device memory. TCUDB consistently outperforms YDB and delivers up to 7.51× speedup. When the dataset contains 32768×32768 records for each table, TCUDB must partition the input matrices into submatrices, use the block algorithm, and pipeline the swapping in/out of submatrices to perform the Join/Aggregation/GroupBy operator. TCUDB still performs multiplication and aggregation of submatrices using TCUs. Even with the overhead of data exchanges in the blocked Join/Aggregation/GroupBy operator, TCUDB is still able to outperform YDB by 7.92× for the case of 32768×32768 records for each table. As datasets fit in the system's main memory as well as the operating system's aggressive caching and the help of high-speed NVMe SSD, the data load time from storage is relatively insignificant in these experiments. The data movement (`cudaMemcpy`) time is the most timing criti-

	2048 ×2048 ×2048	4096 ×4096 ×4096	8192 ×8192 ×8192	16384 ×16384 ×16384	32768 ×32768 ×32768
$x = 0, 1$	0	0	0	0	0
$-2^7 \leq x < 2^7$	0	0	0.00076%	0.00076%	0.00076%
$-2^{15} \leq x < 2^{15}$	0.00114%	0.00450%	0.00908%	0.00908%	0.00908%
$-2^{31} \leq x < 2^{31}$	0.00122%	0.00451%	0.00909%	0.00909%	0.00909%

Table 3.1: The mean absolute percentage error rates (MAPE) of matrix multiplication queries with various value ranges.

cal stage for TCUDB. However, the amount of time is comparable to TCUDB and YDB because both engines only transfer the required data to the device memory. The most time-consuming parts for YDB are `HashJoin` and `GroupBy` operations because code using conventional CUDA cores needs to iterate tables row by row. YDB spends up to 14× (in the case of 16384×16384 records in each table) more execution time in `HashJoin` and `GroupBy` than TCUDB's single Join/Aggregation/GroupBy operator.

Due to the limited 16-bit precision of TCUs, they cannot generate 100% accurate results in some cases. Table 3.1 shows the mean absolute percentage error (MAPE) rates in performing matrix multiplication queries. In the cases where the values are only 0s and 1s – similar to the cases of Q1 and Q2, the generated TCUDB operations can always produce accurate outputs. Therefore, the result implies that TCUDB never leads to incorrect outcomes for sub-queries like Q1 and Q2. When we enlarge the value ranges, we start to see errors in results, but with very limited imprecision – even in the worst case, the MAPE is lower than 0.01%. We believe this error rate is acceptable in most cases. This level of data



(a) (b) (c)

Figure 3.11: The relative runtime of the EM-blocking queries on TCUDB using the default deepmatcher datasets (a) BeerAdvo-RateBeer (b) iTunes-Amazon and (c) scaled iTunes-Amazon, compared to MonetDB and YDB running the same query as the baseline.

error does not cause any inexact query results for the entity matching or the microbenchmark workloads. For numerical analysis such as SSB, the result values can have minor error rates typically less than 0.001% for cases with input values larger than 2^{15} or matrices with a dimension larger than 8192 due to the 16-bit representation. However, the error rate is very insignificant and never results in misplacement of rankings and orderings of the query results.

Entity Matching

Entity matching (EM), also known as entity resolution, fuzzy join, and record linkage, searches records correspond to the same real-world entities from different data sources [57, 54, 39, 127]. A key component of EM is *blocking* [127, 68, 188]. Given two tables of entity records, the goal of blocking is to apply matching heuristics to quickly generate candidate pairs of records that are likely to be real matches, which are later processed by a more accurate pairwise classifier (aka the *matcher*). Scalability is the main challenge of

blocking as the heuristics are typically natural join conditions (e.g., selecting products with the same brand) that often produce large join results. Therefore, we expect that TCUDB can provide significant performance gain for this EM workload.

To validate this hypothesis, we evaluate TCUDB's performance on two real EM datasets BeerAdvo-RateBeer and iTunes-Amazon from the Deepmatcher benchmark [169]. The BeerAdvo-RateBeer dataset contains two tables, where one of them contains 3,777 rows and the other contains 2,670 rows, from different sources. Each table has the same table schema with five attributes {`ID, BEER_NAME, FACTORY, STYLE, ABV`}. Table 3.2 reveals the number of distinct values of each attribute, which acts as one matrix dimension for TCUDB when performing join operation. We evaluate the following query on BeerAdvo-RateBeer dataset to perform blocking:

Attribute	ABV	Style	Factory	BeerName
#distinct values	20	71	3678	6228

Table 3.2: Distinct values in BeerAdvo-RateBeer dataset.

Attribute	Price	Genre	Time	Artist	Copyright	Album
#distinct values	12	813	908	2418	3197	6004
#distinct values (scaled)	25	1614	1208	6420	8199	11005

Table 3.3: Distinct values in iTunes-Amazon dataset.

```
-- EM-blocking query for BeerAdvo-RateBeer dataset:
SELECT TABLE_A.ID, TABLE_A.BEER_NAME,
       TABLE_B.ID, TABLE_B.BEER_NAME
FROM TABLE_A, TABLE_B
WHERE TABLE_A.ABV = TABLE_B.ABV; -- attributes may vary
```

The iTunes-Amazon dataset contains two tables, where one of them has 6,907 rows and the other has 55,923 rows, from iTunes and Amazon music. Both tables share the same table schema with seven attributes `ID`, `PRICE`, `GENRE`, `TIME`, `ARTIST`, `COPYRIGHT`, and `ALBUM`. Table 3.3 shows the number of distinct values for each attribute in the iTunes-Amazon dataset. We perform the following query on the iTunes-Amazon dataset for blocking:

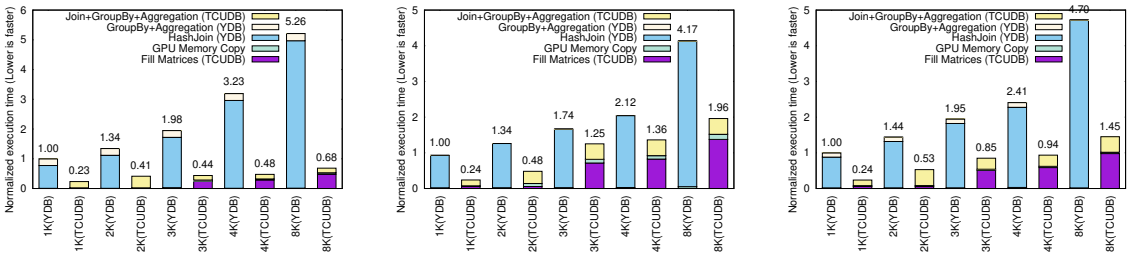
```
-- EM-blocking query for iTunes-Amazon dataset:
SELECT TABLE_A.ID, TABLE_A.SONG,
       TABLE_B.ID, TABLE_B.SONG
FROM TABLE_A, TABLE_B
WHERE TABLE_A.ARTIST = TABLE_B.ARTIST; -- attributes may vary
```

Figure 3.11 presents the result of running the above EM-blocking queries on the two datasets and different attributes. As the execution time varies significantly among different queries, we use YDB running the same query as the baseline and show the relative execution time. TCUDB outperforms YDB in most cases, achieving a maximum speedup of 288× among our experiments.

TCUDB is especially effective when the number of distinct values is small. For the BeerAdvo-RateBeer dataset in Figure 3.11(a), TCUDB is at most 33× faster than YDB when searching for matches on the ABV attribute where there are only 20 distinct values. For the iTunes dataset in Figure 3.11(b), TCUDB further shows 288× speedup over YDB when performing entity matchings on the Price attribute that only has 12 distinct values. When the number of distinct values becomes larger, the performance advantage of TCUDB's operators relying on dense matrix operations over YDB starts to shrink, for the reason we have described in Section 3.5.2. However, as TCUDB uses TCU-spMM in these cases, TCUDB still outperforms YDB and MonetDB in all cases.

Scaling up. To demonstrate the ability of TCUDB and the query optimizer in dealing with larger EM datasets, we synthesized an iTunes-Amazon dataset by randomly duplicating each input table's entry values. The resulting dataset contains 111,846 records in the larger input source and 13,814 in the smaller one. The #distinct values (scaled) show the resulting distinct values in each attribute field of this synthetic dataset.

Figure 3.11(c) shows the relative execution time of TCUDB, compared with YDB running the same query. TCUDB still outperforms YDB in most cases, by up to 216× when performing matching on the price field. When TCUDB performs the query on artist,



(a) (b) (c)

Figure 3.12: The relative execution time of executing PageRank queries (a) Q1, (b) Q2, and (c) Q3 on TCUDB, using YDB running the same query as the baseline. Each value equals the actual query time divided by YDB's runtime on the 1k table.

album and copyright fields, the query optimizer detects that these cases contain way too many distinct values and the pure TCU operator cannot efficiently process the query since the input matrices are sparse. Therefore, TCUDB uses a TCU-SpMM operator for query processing and achieves more than 6.67× and 7.8× speedup on Copyright and Album, respectively, over YDB that essentially performs sparse matrix multiplications using CUDA cores.

PageRank

To demonstrate TCUDB's ability in processing graph-related queries as well as data analytics, we also evaluate TCUDB in performing the PageRank algorithm. PageRank algorithm consists of three steps: (1) computing the out-degree of each node, (2) initializing the value of each node, and finally, (3) calculating the PageRank iteratively. The whole PageRank algorithm can be implemented as the following three queries:

```
-- PR Q1: compute out-degree
SELECT NODE.ID,
       COUNT(EDGE.SRC)
FROM NODE, EDGE
WHERE NODE.ID = EDGE.SRC
GROUP BY NODE.ID;
```

```
-- PR Q2: initialize values
SELECT NODE.ID,
       (1-@alpha)/@num_node as rank
FROM NODE, OUTDEGREE
WHERE NODE.ID = OUTDEGREE.ID;
-- @alpha is 0.85 by default
```

```
-- PR Q3: calculate the PageRank score
SELECT
      SUM(@alpha * PAGERANK.rank / OUTDEGREE.DEGREE)
      + (1-@alpha)/@num_node
FROM PAGERANK, OUTDEGREE
WHERE PAGERANK.ID = OUTDEGREE.ID;
-- @alpha is 0.85 by default
```

#Nodes	1024	2048	3072	4096	8192	16384	32768
#Edges	2058	4152	6280	8450	17444	37106	82070

Table 3.4: Reduced graph information.

Among these three queries, PR Q1 represents step 1, PR Q2 represents step 2 and PR Q3 represents step 3. A complete run of the PageRank algorithm will invoke PR Q1 and PR Q2 once and execute PR Q3 several times until the PageRank scores converge or reach the maximal number of iterations.

We used the Pennsylvania road network dataset from SNAP [135] that contains 1.08M nodes and 1.54M edges as the input dataset. Evaluated TCUDB under different sizes of graphs, we created a subset of the original graph for our experiments using the most popular N nodes and preserving the connectivity of selected nodes in the original graph. Table 3.4 describes the characteristics of the resulting graphs. Figure 3.12 illustrates the relative execution time and the breakdown of latency in each system component for all three queries. We normalized the execution time to run the same query using the graph with 1K nodes on YDB.

Though the computation of out-degree using PR Q1 is a one-pass task (Figure 3.12(a)), TCUDB's pure TCU Join/Aggregation/Groupby operator still has advantages when the graph is small, by up to 3.6× speedup with 1K graph. For graphs with more than 3K nodes, TCUDB selects TCU-SpMM to exercise the Join/Aggregation/Groupby operator due to the low density in their adjacency matrices. Compared with a pure TCU Join/Aggregation/Groupby operator, a TCU-SpMM-based operator spends more time in creating operator inputs. However, as the TCU-SpMM-based operator skips submatrices with all

0s, TCU-SpMM significantly reduces the computation time on matrix multiplications and allows TCUDB to outperform YDB that essentially performs sparse matrix operations on CUDA cores by up to 7.69×.

PR Q2 is also a one-time process in the PageRank algorithm but requires additional arithmetic to initialize the values for PR Q3. Figure 3.12(b) shows that TCUDB consistently performs better than YDB. with speedup ranging from 1.40× to 4.18×. Similar to Q1, TCUDB uses a dense TCU operator for graphs smaller than 2K and uses TCU-SpMM's Join/Aggregation/Groupby to exercise queries for larger graphs.

Figure 3.12(c) shows the performance of TCUDB and YDB in performing PR Q3, the core of the PageRank algorithm that the algorithm executes multiple times until values converge. In our experiments, we performed PR Q3 for 50 iterations for each configuration. For PR Q3, TCUDB's Join/Aggregation/Group operator improves the execution time of arithmetic calculations over the multi-step process in YDB. TCUDB is 4.22× faster than YDB with 1K nodes in the graph. Even with graphs containing 8K nodes, TCUDB still outperforms YDB by 3.24×, as TCU-SpMM's Join/Aggregation/Groupby skips submatrices containing all 0s.

3.5.5 Comparison with Graph Database Systems

TCUDB demonstrates the potential of using relational database engines to analyze datasets that are originally graphs through case studies on PageRank. On the other hand, graph database systems provide more natural representations and storage layouts to serve the same purpose. To investigate the strength and the implications of TCUDB in the future

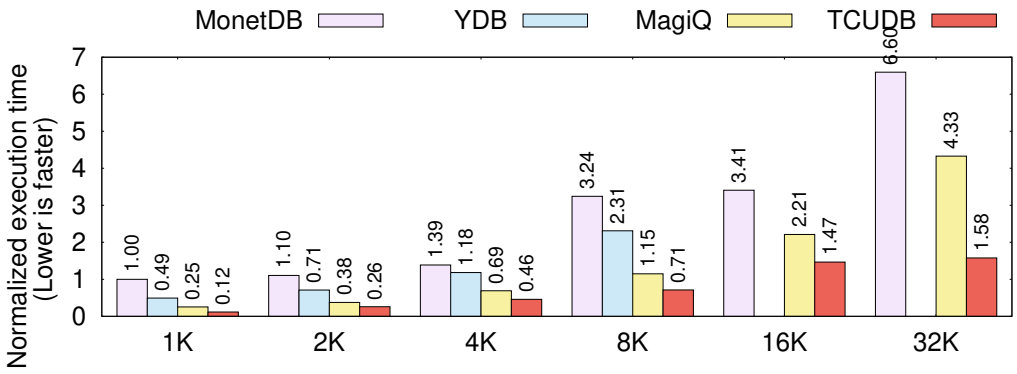


Figure 3.13: The relative latency of the core join and aggregation operation when running PageRank Q3 in MonetDB, YDB, MAGiQ, and TCUDB.

advancement of graph database systems, this section compares the performance of TCUDB on the PageRank algorithm with the state-of-the-art graph query engine MAGiQ [102]. In contrast to the table-style storage that relational database systems and TCUDB use, MAGiQ's backend storage is organized as 2-dimensional key-value pairs, typically already in some sparse matrix formats. MAGiQ translates the queries described by SPARQL into a set of GraphBLAS [43] calls on these sparse matrices.

We use the same SNAP dataset as in Section 3.5.4 to evaluate the PageRank performance of MAGiQ with GPU and TCUDB. Figure 3.13 compares the performance of MAGiQ and TCUDB with MonetDB and YDB as references. However, the released version of YDB can only support these queries with datasets containing at most 8,192 nodes. Due to the large overhead of retrieving sparse matrices in MAGiQ compared to other counterparts, we only present the latency of the core join and aggregation operations in each experiment. The presented numbers are PageRank Q3's performance on the sub-sampled graphs listed in Table 3.4. MAGiQ outperforms YDB, the pure GPU query engine on relational databases,

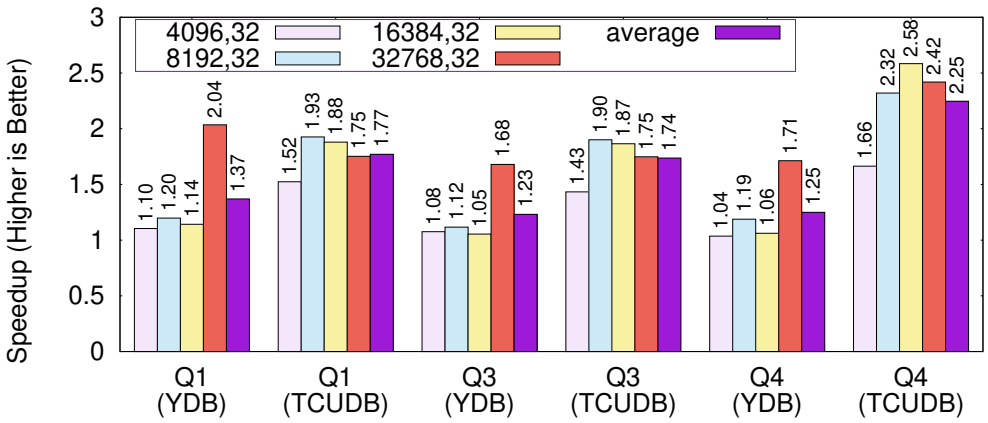


Figure 3.14: The microbenchmark speedup of using RTX 3090 over RTX 2080 for Q1, Q3, Q4 on TCUDB and YDB. Each value equals RTX 2080 time divided by RTX 3090 time.

in all cases, demonstrating that a customized graph database engine does provide a more efficient platform for graph analytics on the same architecture. Meanwhile, TCUDB outperforms MAGiQ in all evaluated cases. The main reason is that TCUs allow TCUDB to more efficiently exercise these queries than GraphBLAS that uses only conventional GPU cores at this moment. We observed that the difference is more significant as the graph becomes larger and more sparse. These results help us generate two insights. First, with TCUs, graph analytics can be efficient with existing relational databases. Second, graph databases can also be more efficient if their backends can leverage TCUs as TCUDB does.

3.5.6 TCUDB on different GPU architectures

To investigate the performance scaling on different GPU architectures and their implications to the design of the TCU-accelerated DB engine, we perform experiments on NVIDIA's 2080, which uses an earlier Turing GPU architecture with the last generation

TCU available.

Figure 3.14 compares the performance of microbenchmarks on the same queries Q1, Q3, Q4 mentioned in Section 3.5.2 using both YDB and TCUDB on RTX 3090 GPU and RTX 2080 GPU. The baseline ran the same query using the same DB engine on RTX 2080. We observed that TCUDB performs better generation-over-generation – when using RTX 3090 TCUDB achieved an average speedup of 1.77× on Q1, 1.74× on Q3 and 2.25× on Q4, but YDB only achieved 1.37× on Q1, 1.23× on Q3 and 1.25× on Q4. It is worth noting that RTX 3090 contains only 328 Tensor Cores compared to 368 Tensor Cores in RTX 2080. On the other hand, the RTX 3090 has 10496 conventional CUDA GPU cores for vector processing while RTX 2080 only has 2944 of them. The results reveals that the performance scaling of Tensor Cores in newer generations of GPU architectures is stronger than conventional vector processing cores, given that RTX 3090 has fewer Tensor Cores, 3.4× more CUDA cores, but TCUDB's speedup is more significant on RTX 3090. This result also indicates applications, including DB engines, with a larger portion relying on TCUs will expect to receive more performance gains when new GPU architectures are used.

3.6 Related Work

Hardware-accelerated DB's. Integrating advanced hardware accelerators into database systems has been an active line of research for the past few decades. Commonly considered accelerators include GPUs [80, 242, 255, 267, 25, 246, 256, 136, 191, 266, 24, 40, 67, 219] and FPGAs [170, 248, 154, 186, 60]. Optimization techniques have been proposed for database operators including Select [226], Join [90, 91, 84, 225], Sort [79] and Group-by

Aggregate [116]. In particular, to support star schema queries, YDB [267] implements these operators into a data warehousing engine, which we used as a baseline for TCUDB. GPUs have also been incorporated into industrial DB engines such as OmnisciDB [181], Kinetica [125], and BlazingSQL [18].

With GPUs reducing the computation time but the increasing volume of datasets, the data movement overhead becomes more significant to the degree that DB engines must be aware [194, 23]. Several GPUDB systems incorporate GPU RDMA techniques [176, 10, 121, 236, 270, 145] to directly access data on the storage devices [272, 136, 37] or efficiently exchange data among multiple GPUs [151], bypassing the host system's main memory. This paper is orthogonal but will receive significant benefit from this line of research projects. To fundamentally address the data movement overhead, DB systems can push down the computation of query processing into existing or additional hardware logic to offload part of the computation instead of using computing resources on the host system [52, 51, 112, 245, 128, 244, 105]. However, due to the power and hardware budget of memory/storage devices, the computing resources near data locations are typically limited. For the cases studied in this paper, DB systems still have to rely on host computing resources (i.e., GPUs, TCUs, FPGAs and TPUs) to efficiently perform the received queries. With modern matrix processors need to partition matrix data and accept reduced precision values, DB system like this paper can still leveage near data processing models to reduce precisions [98] or reshape data [146] if the processing power in storage devices is permitted.

Matrix processors in relational databases. To the best of our knowledge, TCUDB is the first database system that fully leverages Tensor Core Units (TCUs) as matrix proces-

sors to accelerate compute-intensive database queries. Prior work [42] leverages TCUs for scan/reduction operators by mapping scan/reduction into matrix-vector products. However, [42] only treats TCUs as wider vector processors leveraging TCU's fused operations that can perform multiplications and accumulations in a single operation. In contrast, TCUDB transforms queries into matrix-matrix operations so that it can fully utilize TCUs' nature as matrix processors. Prior work [96] investigated the feasibility of accelerating relational queries using Google Cloud's closed-architecture TPU platform and proprietary version of TensorFlow. However, due to limitations of the platform, [96] only accelerates vector-based operators such as reduced sum. Its implementation can only support single-table queries (called Dimension Join in [96]). On the other hand, TCUDB can support a wide range of queries include two-way natural joins by leveraging TCUs for matrix operations.

Join processing as matrix multiplication. A key technical contribution of TCUDB is to cast the join operator as dense matrix multiplication. While being unconventional due to the high theoretical computational complexity, this idea was explored in [11] and more recently in [45]. In particular, [45] proposed a fast join algorithm that combines worst-case optimal join algorithms [175] and fast matrix multiplication. The authors also provide a CPU-based implementation highlighting performance gain from the highly-optimized linear algebra framework such as Intel MKL [243]. The implementation achieves up to 50× performance improvement compared to baselines. In TCUDB, we further push this trend by leveraging NVIDIA's TCUs that are specialized for tensor processing, which commonly appears in deep learning workloads to achieve up to 288× performance gain.

Graph queries as matrix operators. Processing queries as matrix operators have also

been considered in the context of graph databases. In particular, MAGiQ [102] accelerates SPARQL queries on RDF graphs by translating queries into sparse matrix linear algebra programs. We have discussed the key differences between TCUDB and MAGiQ in Section 3.5.5. Our experiment results also show that integrating TCUDB's strategy of executing those matrix operators in TCUs can be an interesting optimization opportunity for graph query engines like MAGiQ.

Advanced in-database analytics. To accommodate the exponential growth in data science and machine learning applications, a recent line of work [93, 4, 56, 150, 53, 99, 230, 26] focuses on supporting advanced analytics queries that involve linear algebra (LA) operators. TCUDB shares the goal of LevelHeaded [4] in identifying the worst-case optimal join (WCOJ) [175] or LaraDB's rule-based translation between relational queries and parallel LA queries, but TCUDB additionally provides the capability of translating (parts of) the query to TCU-accelerated matrix multiplication operator(s) and different sets of opportunities from the orders of magnitude speedup by TCUs in such operations. TCUDB also offers a better system architecture by making TCU-accelerated operators as integral parts of the DB engine and thus incurs zero system overhead in processing TCU-accelerated queries. In contrast, query analyzers like AIDA [56] that rely on external parallel libraries from different language frameworks from the query engine always lead to redundant memory copies that are especially significant in our use cases. Compared with proposals relying on SQL extensions that introduce data type labels (e.g., vector and matrix) to support LA queries [150] or new query languages [26], TCUDB does not require any change to the SQL.

Entity Matching and PageRank. A major challenge in EM [57, 54, 39, 127] is in the

blocking phase [127, 68, 188] to reduce the number of candidate pairs to be matched by heuristics specified as natural joins. Our case study demonstrates that TCUDB delivers over 300× speedup for blocking queries compared to a GPU-accelerated `HashJoin` implementation. This indicates the potential of building scalable EM systems with TCUDB as the backend.

PageRank is a graph-based ranking algorithm with applications from web searches to basic science (see [73] for a survey). PageRank is also commonly used in benchmarks of graph databases [166, 174, 49]. While there has been an effort to accelerate PageRank (and other graph analytic queries) using GPUs [257, 207, 222], to our knowledge, TCUDB is the first to attempt to accelerate PageRank using TCUs.

3.7 Conclusion

We propose, implement and evaluate TCUDB, an efficient database query engine with TCUs, an emerging type of AI/ML hardware accelerator presented in modern GPU architectures. We identify query patterns that match TCUs' acceleration model. Through solving technical difficulties such as remapping inputs and limited precision, the resulting TCUDB shows ours achieves up to 288× speedup against the baseline GPU-accelerated DB engine. The performance gain of TCUDB over conventional GPU-based DB engines indicates a strong performance scaling in new GPU architectures. For future work, we plan to extend TCUDB by exploring more potential workloads and addressing the complex query optimization problem with multiple accelerators of different types.

Chapter 4

Assessing Hardware Effectiveness for AI Applications

We have seen a huge surge in personal assistant applications, from basic chatbots to advanced AI virtual assistants that go beyond what we ever imagined. The shift from traditional personal computers to personal mobile devices in the human-computer interface has introduced complexities in the data processing pipeline for personal assistant applications, leading to changes in various aspects of the data management model.

The conventional model of handling these data management pipelines, data-center machine learning (DCML), heavily relies on data centers with powerful processors, GPUs, or hardware accelerators like TPUs [110, 108, 106] to perform the majority of computation of personal assistant applications. The device that attaches to an end user is only responsible for collecting inputs and requests as well as presenting the response from data centers.

Despite the short latency of performing core ML tasks (e.g., inference) using data center servers, DCML model suffers from the overhead of exchanging data over the network.

Beneficial from the integration of GPUs and accelerators (e.g., edgeTPUs and Apple's Neural Engines) into modern mobile devices or Internet of Things (IoTs) open up the avenue for on-device machine learning (ODML) that performs critical AI/ML functions on mobile or IoT devices without too much intervention from the data center. By removing the data exchange overhead, ODML may still offer competitive performance despite the lower computation capability on user devices. However, for workload that require more computation or memory resources, ODML still struggles to provide acceptable user experiences.

In addition to the two extremes of data processing models, we also explore the potential of another data management model, multi-device machine learning (MDML). Instead of heavily relying on either the extreme of user devices or data center servers, MDML promotes the collaboration of heterogeneous devices in accomplishing tasks in emerging AI/ML-assisted/centered applications.

Challenges. However, the choice of data management models, design, and development of AI/ML-assisted/centered applications must tackle the following challenges. (1) Trade-offs. Efficiently using heterogeneous devices in an application leads to trade-offs in every design decision. For example, data center servers can deliver short latency in inference, making user devices cheaper and smaller but increasing the end-to-end latency, device power/energy consumption, and privacy concerns. (2) Quality of service. Many personal assistant applications interact with end-users directly. Therefore, these applications must deliver an expected response within a specific latency. Some devices (e.g., low-precision hardware

accelerators) and mechanisms (e.g., approximate computing) can deliver supreme performance but fail on quality. (3) Cost. The operational costs (e.g., electricity consumption and carbon footprints) and cost of ownership will affect an application's sustainability and long-term success. An energy-inefficient or capital-heavy approach may not necessarily be a feasible solution. (4) Developing code on heterogeneous devices. As personal assistant applications require multiple devices, where each device is highly specialized with distinct processor/accelerator architectures, the developer will need to optimize the implementation on different devices separately. In addition, the developed code is not portable between heterogeneous devices.

4.1 Overview of PAMLB

We present Personal Assistant Multi-device Machine Learning Benchmark (PAMLB) to address challenges in developing data management pipelines on emerging personal assistant applications. In contrast to conventional benchmark suites that often focus solely on queries executed within a single device, PAMLB targets the comprehensive data management pipeline, including operations that may extend across multiple devices or platforms. PAMLB leverages a device-agnostic abstraction, DAQL, that can easily describe the interaction among different data management pipeline elements. The abstraction also allows PAMLB to support theoretical analyses encompassing performance, energy, and data movements. Furthermore, PAMLB encompasses a wide range of applications, covering the majority of data processing pipelines typical for personal assistant applications. The PAMLB application set enables developers to assess the impact of optimization or deployment of a

module to devices with analogous applications prior to code development. The variety of PAMLB's data management pipelines will enable the developer to quickly map an emerging application's data management process to one in PAMLB, saving the evaluation time of data processing models of new applications.

Insights learned from PAMLB. With PAMLB's abstraction and availability on various types of devices, we evaluated PAMLB's workloads in settings covering the three data processing models, DCML, ODML, and MDML. We identified the following insights through the evaluations of real systems and projections with measured numbers and PAMLB's abstraction. (1) With the size of AI/ML models growing, DCML is more advantageous in latency despite increasing data movements, but with appropriate allocation of tasks, MDML can consistently deliver the service at the desired level. (2) ODML is feasible for workloads with low arithmetic intensities but unlikely to work efficiently for applications that need large models. (3) The performance of datacenter servers does not always translate to user experience. (4) With the current progress in hardware advancements, the optimizations on models matter more. (5) Disregarding performance, ODML can be the most eco-friendly model with recent advancements in device technologies. (6) ODML can also be the least eco-friendly if the device is too slow. (7) Considering the service levels, MDML is the most eco-friendly and cost-effective model. (8) Optimizing the performance of PAMLB still needs intensive investigations. Based on the learned insights, PAMLB also explored the potential of an optimizer on PAMLB workloads to avoid the process of exhaustive design space exploration.

PAMLB makes the following contributions:

- We propose a methodology to create PAMLB, a collection of personal assistant workloads across various representative domains, including video query, visual query, natural language query, geolocation-based/spatial query, and question-answering system. To the best of our knowledge, PAMLB is the first benchmark considering the multi-device interaction from a holistic system design point of view for personal assistant applications.
- We identify MDML as a competitive data management model for personal assistant applications that adds 11% user-perceived latency while saves 41% energy on average.
- Among our key findings, we highlight two significant insights:

  (1) Despite a more than 4× improvement in compute capability (TOPS) for the upgraded hardware, the performance of applications, in terms of latency, improved by only 11% due to limited memory bandwidth enhancements.

  (2) The selection of an appropriate machine learning model has a greater impact on end-to-end latency than upgrading the computing device. Adopting a specialized model over a generalized one resulted in a more than 3× speedup.

- We evaluate the potential of a simple yet effective method to guide the model selection on various workloads across multiple devices.
- We have made all our evaluation scripts, benchmark specification representation, and

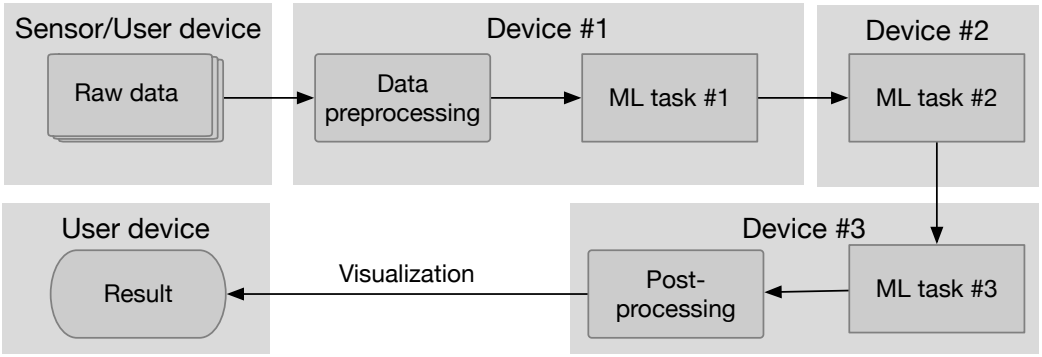


Figure 4.1: Typical machine learning application pipeline in MDML data processing model.

analysis available online. In this age of rapidly evolving information and technology, a benchmark that comprehensively captures the behaviors of widely discussed AI applications can contribute to future research as community resources.

4.2 Background and Motivation

This section describes the background of different data processing models in the multi-device setting, and their motivations given the recent rapidly evolving ML/DL technologies and hardware. By describing the applications with our proposed specification language DAQL, we can evaluate a set of representative workloads systematically and analyze the trade-offs in terms of cost, energy consumption, and hardware computing capability.

Table 4.1: Data processing models comparison.

Metrics \Data processing models	**ODML**	**DCML**	**MDML (ours)**
Total energy consumption	Low	High	Medium
Server cost	Low	High	Medium
User cost	High	Low	Medium

4.2.1 Data Processing Models

Data-center Machine Learning (DCML)

Data-center inference processing model [199, 88, 70, 86] indicates the practice of running machine learning models on cloud servers that are equipped with high-performance hardware, such as high-end GPUs or tensor processing units (TPUs). DCML enables high-performance inference and faster response time. Besides, DCML provides good scalability in which it can handle a large number of concurrent inference requests and thus a a suitable solution for applications with high traffic and demand.

While DCML offers considerable computational power and accessibility, it also has some drawbacks. (1) Privacy. Sending data to external servers for inference might pose privacy issues, especially when dealing with sensitive or personal information. However, as privacy concerns are beyond the scope of this work, we focus on data transfer time as a component of response time in our machine learning pipeline. (2) Cost for high throughput. For high-throughput applications, it will incur significant costs because of the computational resources used. (3) Network reliance. Since DCML relies on network communication between the user device and the server, which introduce latency in obtaining results that is

critical for real-time applications. Moreover, a stable internet connection is mandatory for data-center inference to get continuous service.

On-device Machine Learning (ODML)

ODML refers to the process of running a complete machine learning pipeline directly on a user device, such as mobile phones, tablets, or IoT devices, without relying on the cloud or the remote server. This data management model is opposed to the conventional data-center-based model, where a user device is used to issue a request and receive the results without data processing and computation. Representative ODML frameworks [50, 149, 253, 85, 134, 86] make efficient use of computing resources on the device to obtain an acceptable performance. ODML preserves several advantages that are hard for other solutions to compete with. (1) Privacy. Because data stays on the user's device, there is no need to transfer sensitive data to third-party servers, which greatly reduces privacy concerns. (2) Latency. On-device inference minimizes the delay in obtaining results because there is no network communication. (3) Availability. Devices can perform inference without an internet connection, allowing continuous functionality.

However, ODML also faces some challenges, as the less powerful on-device computational resource and memory can limit the performance level of the applications or result in a higher cost to the user. Considering the current user devices' specifications, some AI/ML-assisted applications need to be highly optimized or quantized to run effectively on resource-constrained devices, which incurs extra engineering efforts.

Multi-device Machine Learning (MDML)

MDML refers to the data processing model where computation tasks can be distributed to more than one device. This is the typical paradigm for emerging multimodal ML tasks (e.g., visual question answering) that require inference calls of multiple models in different modalities such as vision and language. Figure 4.1 shows the typical machine learning pipeline in MDML scenario. The raw data can come from either the sensor or the user device, and device#1 (e.g., mobile devices) conducts data preprocessing and light-weight ML task. The compute-intensive task (e.g., ML task#2) can be handed over to device#2 (e.g., the server machine). Next, the intermediate results can be propagated to device#3 (e.g., edge devices) to perform ML task#3 if any, otherwise, results can be directly sent back to the user's device for rendering after post-processing.

Compare with other data processing models such as the ODML model and DCML model, MDML model can process the light-weight task without fully relying on the server and preserve the opportunity to handle the private data locally. Flores et al., Gao et al., and Wen et al. [64, 71, 249] exploit the multi-device edge computing by organizing a group of devices for the AI inference task, which is also a subset of the MDML we discussed in this work. Because edge devices by far can not deliver satisfying latency for AI/ML applications, running applications with a cluster of edge devices may cancel out the benefit of using these low-power devices for energy saving. Pavlo et al. [192] compares the data processing pipelines of two paradigms (MapReduce and parallel SQL DBMS) on large-scale data in terms of performance and development complexity. While MDML serves as a hybrid model of DCML and ODML to balance user demands including end-to-end latency,

costs, and energy consumption. Table 4.1 shows a high-level comparison of three models. ODML, while offering low total energy consumption and server costs, imposes higher user cost due to the necessity for better hardware to maintain the performance. From a latency perspective, DCML by all means offers the best performance, but we have to consider the trade-offs on energy efficiency and costs. MDML serves as a hybrid option by leveraging part of the computation to the power-efficient device, thereby relieving server pressure and saving energy compared to DCML. MDML preserves the opportunity to perform data preprocessing before transferring data to high-performance devices for compute-intensive tasks.

4.2.2 PAMLB: Addressing Changes of Data Management Pipeline

Personal Assistants. In this paper, we present a benchmark, PAMLB, for evaluating the three aforementioned processing models: DCML, ODML, and MDML. In particular, we choose AI-based personal assistants as the main target domain. Personal assistants are ideal candidates for evaluating these data processing models because they inherently operate across a wide array of devices (i.e., smart phones and data centers) and require real-time, efficient, and accurate responses. The multi-device nature demands the deployment of machine learning models with different modalities and resource requirements across devices of varying capacities.

What does PAMLB cover? Running ML applications on heterogeneous devices introduces several system design decisions, including the trade-offs of task allocation among different hardware, computing resource management, and application interfaces across platforms. For example, approaches to the heterogeneous system design from the task scheduling

perspective [148, 143, 269, 8] dynamically place tasks on devices based on their computing capabilities and network conditions. On the other hand, Zhou et al. and Risso et al. [276, 205] fully exploit the parallelism of computing resources on the device to maximize the performance of heterogeneous computing. In summary, PAMLB addresses the following challenges for MDML:

Response Time

For applications with real-time interaction requirements, response time is critical. We need to deliver a certain degree of results to users to fulfill the demand. Compression strategies or quantization techniques [41, 65, 124] are applied as optimizations to achieve reasonable results within a specified time frame and reduce memory footprint.

Energy Efficiency

Beyond performance, energy efficiency is closely linked to both costs and long-term environmental sustainability. Addressing energy consumption in emerging AI-based personal assistant applications presents another challenge. This information allows us to choose various hardware configurations to attain desirable outcomes with reduced energy consumption.

Unifying Device Abstractions

Another challenge for PAMLB is addressing the various interfaces across multiple devices. Given that different platforms have their own operating systems and instruction sets, the ML pipeline may not run smoothly using the MDML data management model

without modifications. The proposed PAMLB should offer a unified abstraction to distribute the workload among different devices and to estimate the data exchange overhead between them.

Being Implementation-agnostic

The diversity and representativeness of AI/ML applications pose challenges that machine learning benchmarks should address. While there can be hundreds of implementations for a single ML task, evaluating them with a consistent standard is difficult. For example, many different implementations exist for object detection tasks [200, 144, 202, 92], and each delivers strong performance under different scenarios. An ideal PAMLB should be flexible enough to capture all these characteristics, enabling better analysis of potential bottlenecks within the ML pipeline.

4.2.3 Why Do We Need a New Benchmark?

Currently, there is no benchmark for MDML use cases. In a heterogeneous computing environment, one ML task can be implemented in a number of different ways. We lack a general representation for ML applications that is implementation-independent and can characterize each adjacent machine learning component individually.

Most current ML benchmarks and AI competition platforms (e.g., Kaggle, DrivenData, Numerai) focus on model accuracy by providing reference implementations as baselines for specific tasks or by encouraging participation through dedicated leaderboards. Another set of benchmarks emphasizes basic operations crucial to DL, such as the fully-connected layer (FC), convolution layer (Conv), and recurrent neural network (RNN), aim-

ing to evaluate performance for select target workloads. However, as the complexity of tasks (e.g., multimodal tasks [75, 81, 19, 221]) increases along with user demands and evolving technologies, establishing a standard for each specialized task in this swiftly changing era becomes nearly impossible. For instance, large language models (LLMs) [184, 234, 6] have emerged, captivating people worldwide with their impressive capabilities. Yet, cataloging and comparing the plethora of LLMs developed by various companies and research labs is a daunting task. On the other hand, Liang et al. [140] investigate the capabilities and limitations of multimodal models in this multimedia era by providing an automated machine learning pipeline and standardized implementations of core approaches. This demands programmers and researchers to continually develop and update metrics in response to the rapid emergence of new models. In essence, handcrafting evaluation metrics for each model or task is almost unfeasible as the state-of-the-art (SOTA) models evolve daily.

Observing the drawbacks of adapting to continuously updated model implementations, we believe that a benchmark should provide a high-level abstraction to capture the characteristics of the task and bypass specific model implementations. Furthermore, instead of focusing solely on model accuracy, other metrics, such as latency and energy consumption, are crucial for both users and developers. For instance, responsiveness (i.e., end-to-end latency) affects user experience and satisfaction, while energy consumption reflects both the carbon footprint and electric bills. An effective benchmark should offer insights to system designers and application developers, guiding them to allocate computation to the optimal device or hardware. This ensures the best use of resources and maximizes user experience.

Table 4.2: Operators in MDML specification language.

Operator	**Category**	**Meaning**
TRANSFORM	Preprocessing	Text-based data preprocessing, e.g., text parsing, text transformation, no-ops.
EXTRACT	Preprocessing	Image/video preprocessing operations, e.g., rasterization, voxelization, vectorization for images.
PREDICT	Inference	Perform inference.
CONCAT	Postprocessing	Concatenate a sequence of tokens.
POST_PROCESS	Postprocessing	Any kind of post-processing, e.g., clean up noise, deduplicate, no-ops.

4.3 Evaluating personal assistant applications on multiple devices

We introduce an abstraction that is independent of devices, machine learning models, and programming languages. Alongside this, we present a set of workloads representative of personal assistant applications. These are aimed at addressing the challenges of evaluating data processing models when executing these workloads across different hardware and programming platforms. This section provides an overview of the proposed abstraction and the associated workloads.

4.3.1 Device Agnostic Query Language (DAQL)

Modern AI-assisted applications typically involve data paths that encompass heterogeneous types of devices. To circumvent dependencies on system-level implementations, we adopt declarative language inspired by BigQuery ML [77] and the Structured Query Language (SQL). To capture the full picture of data interactions and computation, we introduce

Table 4.3: Summary of PAMLB.

TASKS	INPUTS	COVERAGE	TIME REQUIREMENT
VQImg [81]	Video, Image, Natural language[81]	Object detection, Object tracking, Video retrieval [17]	The shorter is better
VQNL [81]	Video, Natural language[81]	Natural language processing ,Natural language reasoning ,Video retrieval	The shorter is better
Text-to-image Generation [240]	Natural language [241]	Natural language processing ,Natural language comprehension ,Image processing	The shorter is better
VMF [221]	Sensor data, Map data [59]	Image processing, Signal processing, Spatial computing	Must be shorter than 25 ms [274, 277]
Question Answering Bot [275]	Natural language [198]	Large language models, Natural language processing, Generative AI	The shorter is better
Bilingual Voice Assistant	Audio, Natural language [198]	Speech recognition, Natural language processing, Generative AI	The shorter is better
Recommender [254]	User-item interactions [254]	Recommendation system, Data mining	The shorter is better

DAQL as an extension to SQL. DAQL appropriately abstracts our target applications and scenarios without necessitating extensive programming knowledge of AI/ML frameworks.

Drawing from the principles of SQL, DAQL represents workloads as a series of queries. Each query can encompass multiple operators, each with its associated clauses. These clauses specify the precise data processing commands, conditions, and the input/output data pertinent to the corresponding operator. Table 4.2 summarizes the operators that cater to all use cases discussed in the current set of workloads in this paper.

Using a text-to-image application (e.g., Microsoft's Bing Image Creator or Google's Imagen [209]), we will explain how DAQL describes a workload. Figure 4.2 illustrates the workflow of the text-to-image application [103]. A text-to-image application receives an

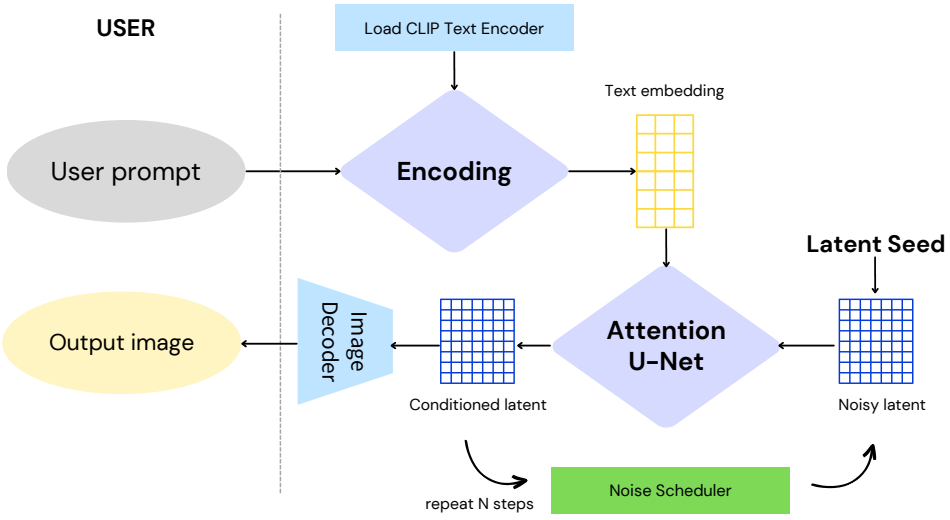


Figure 4.2: Simplified text-to-image generation workflow.

end-user request in natural language sentences, like "Can you draw a picture with a bear riding a bike? The bike is black. The bear is wearing sunglasses". The application then parses and encodes these input sentences into embeddings. Finally, using one or more generative AI models, the application produces an image that visually corresponds to the provided text description.

Figure 4.3 shows the DAQL query that covers the complete workflow of the text-to-image application. The `DECLARE` clause at the first line defines the request from the end-user input. Then, the `LOAD MODEL` clause at line 3 loads a pre-trained model called `CLIP` [197] as `encoder`. The later `SELECT` clause invokes a `PREDICT` operator that generates embeddings into `text_embeddings` using the model loaded as `encoder`. The `LOAD MODEL` clause at line 6 would load another pre-trained model, `stable_diffusion_model`, as a `predictor` into the application. The syntax also allows the user to optionally set the parameters of the

```
1  DECLARE prompt STRING DEFAULT 'Can you draw a picture with a
       bear riding a bike? The bike is black. The bear is wearing
       sunglasses';
2
3  LOAD MODEL `CLIP` AS encoder;
4  SELECT PREDICT(MODEL encoder, prompt) AS text_embeddings;
5
6  LOAD MODEL `stable_diffusion_model`
7  OPTIONS (
8    input_seq_length=128,
9    language='en',
10   latent_noise='path_to_latent_noise_data',
11 ) AS predictor;
12
13 /* Perform inference to generate image with given prompt */
14 SELECT PREDICT(MODEL predictor, text_embeddings) AS
       generated_image;
```

Figure 4.3: The DAQL query for the text-to-image application.

loading model `OPTIONS` statements. Finally, the `SELECT` clause uses the embeddings that the `encoder` generated as the input, and the `PREDICT` operator invokes the `predictor` to perform inference on the selected embeddings. If the workload loads an appropriate model in the `LOAD MODEL` clause at line 6, the query would finally generate an image based on the user's request.

Beyond the capability of describing the complete control flow and dataflow of the workload, each clause of the query in Figure 4.3 is entirely independent of devices and programming frameworks. The actual implementation of the query can implement each

clause on a different device and communicate the input/output through memory copies or network links. The `LOAD MODEL` clause describes which model to load, but does not enforce the implementation of the model. If more efficient models emerge, the evaluator can replace the file in `LOAD MODEL` clauses as long as the input/output format is compatible or similar. The evaluator may use the query to estimate the performance of applications that the current set does not cover if the new workload can be reduced to an existing question. For example, suppose the evaluator wants to gauge the design of an application that generates a video from a user question. In that case, the evaluator only needs to replace the model loaded at line 6 without significantly changing the query.

4.3.2 Benchmark Applications

We have collected seven representative personal assistant workloads to form our benchmark, PAMLB. These applications not only represent real-world use cases suited for the workload, but they also showcase the adaptability of their data processing pipelines for emerging applications. Table 4.3 summarizes the workloads in PAMLB, including details on referenced models, input datasets, application domains, and key performance metrics.

These workloads share several characteristics. First, at least one of their inputs and one of their outputs are in human-perceptible formats. Over half of the workloads accept multiple types of inputs. Second, response generation spans various application domains using traditional execution models and abstractions. Lastly, these applications feature multi-stage data processing pipelines, highlighting the potential for distributing stages across different devices.

```
1  /* Data preprocessing */
2  TRANSFORM('annotation.json') AS annotations;
3  EXTRACT('demo.mp4') AS clips;
4
5  LOAD MODEL `object_detection_model`
6  OPTIONS (model_type='SiamRCNN') AS detector;
7
8  LOAD MODEL `object_tracking_model`
9  OPTIONS (model_type='kys') AS tracker;
10
11 /* Perform VQImg inference */
12 SELECT
13   (bbox, score) AS response_track
14 FROM
15   PREDICT(MODEL tracker, (
16   SELECT
17     bbox -- detector's output
18   FROM
19     PREDICT(MODEL detector, (
20     SELECT
21       /* inputs for detector */
22       'visual_crop', 'query_frame', 'clip_frames'
23     FROM clips, annotations))
24   ));
```

Figure 4.4: The DAQL query for the VQImg application.

Aside from the text-to-image workload, which we discussed in Section 4.3.1, we will provide brief descriptions of the remaining six workloads in PAMLB and their corresponding DAQL queries in the subsequent paragraphs.

Visual Query with Image (VQImg) VQImg accepts an input image from the user, such as a photo from a camera, and localizes this image within another input: a video stream. In response, VQImg provides the timestamp of the most recent appearance of the user's input in the video, accompanied by objects encased in 2D bounding boxes. For generating

```
1  /* Data preprocessing */
2  TRANSFORM('annotation.json') AS word_embeddings;
3  EXTRACT('demo.mp4') AS visual_features;
4
5  LOAD MODEL 'multimodal'
6  /* model_type=[TRANSFORMER, RNN, LSTM] */
7  OPTIONS (model_type='TRANSFORMER') AS predictor;
8
9  /* Perform VQNL inference */
10 SELECT
11   ('clip_uid', 'annot_uid', 'query_idx', 'predicted_times') AS response_track
12 FROM
13   PREDICT(MODEL predictor,
14     (word_embeddings, visual_features))
15 LIMIT K;
```

Figure 4.5: The DAQL query for the VQNL application.

this response, VQImg relies on modules designed for object detection, object tracking, and video retrieval. Figure 4.4 displays the DAQL representation of VQImg.

Visual Query with Natural Language (VQNL) VQNL receives a user query in natural language (e.g., "Where is my key?") and performs the corresponding search in an input video stream. The response provides the timeframe where the answer is either derivable or visible. Despite the core visual query component being similar to VQImg, VQNL's data processing pipeline is applicable to all applications that integrate visual understanding and language comprehension. Figure 4.5 shows VQNL's DAQL representation.

Vehicle Motion Forecasting (VMF) VMF receives sensor data, camera images, the current geographical location, and a map as inputs and predicts the vehicle's trajectory by considering the surrounding environment, road maps, and observed agents such as vehicles,

```
1  /*
2   *  Assume single trajectory prediction.
3   *  raw_data=[sensor_data, map_data]
4   */
5  EXTRACT(raw_data) AS input_features;
6
7  LOAD MODEL 'motion_pred_model'
8  OPTIONS (model_type='TRANSFORMER') AS predictor;
9
10 /* Perform motion prediction */
11 SELECT
12 ('timestamps', 'track_id', 'future_coord_offsets') AS pred_trajectory
13 FROM PREDICT(MODEL predictor, input_features);
14
15 /* Convert coord from agent space to world space */
16 POST_PROCESS(pred_trajectory) AS displacement_world_space;
```

Figure 4.6: The DAQL query for VMF.

cyclists, and pedestrians. The workload requires feature extractions from input data and an appropriate model to predict the vehicle's trajectory [97]. Figure 4.6 depicts the workflow of VMF. VMF has a critical timing constraint that the end-to-end latency of each request in the workload must finish within 25 ms [274, 277].

Question Answering Bot (QABot) QABot serves as a representative workload wherein users pose questions in natural language to the application, and in return, the application provides responses in the same form. In-production examples of QABot include ChatGPT [184], Llama-2 [234], and Vicuna [33]. A typical Transformers-based architecture of QABot consists of two modules: an encoder and a decoder. Given user input, the encoder first translates the question into latent vector representations. Subsequently, the decoder module produces responses in human-readable formats (e.g., natural languages) based on these

```
DECLARE question STRING DEFAULT 'Who is the author for The Little Prince?';

LOAD MODEL `qa_model`
OPTIONS (
   model_type='TRANSFORMER',
   input_seq_length=128,
   language='en'
) AS qa_model;

/* Encode the question */
SELECT PREDICT(MODEL qa_model, question) AS encoded_question
FROM question;

/* Perform inference to generate the answer */
SELECT PREDICT(MODEL qa_model, encoded_question) AS predicted_answer
FROM encoded_question;

/* Decode the answer */
SELECT CONCAT(predicted_answer, '_') AS answer;
```

Figure 4.7: The DAQL query for QABot.

encoded representations. More recent LLMs often adopt a decoder-only architecture in which the encoder and decoder share the same model parameters. Figure 4.7 illustrates the QABot process.

Bilingual Voice Assistant (BVA) BVA is a workload in which we intentionally made the scenario more complex by extending the QABot to accept inputs and produce outputs in different languages. Figure 4.8 displays the workflow diagram of our synthetic voice assistant application. Besides demonstrating the flexibility and extensibility of PAMLB, BVA embodies a potential scenario that caters to the diverse language requirements of users in today's multilingual and cross-cultural environments, which current applications fail to ad-

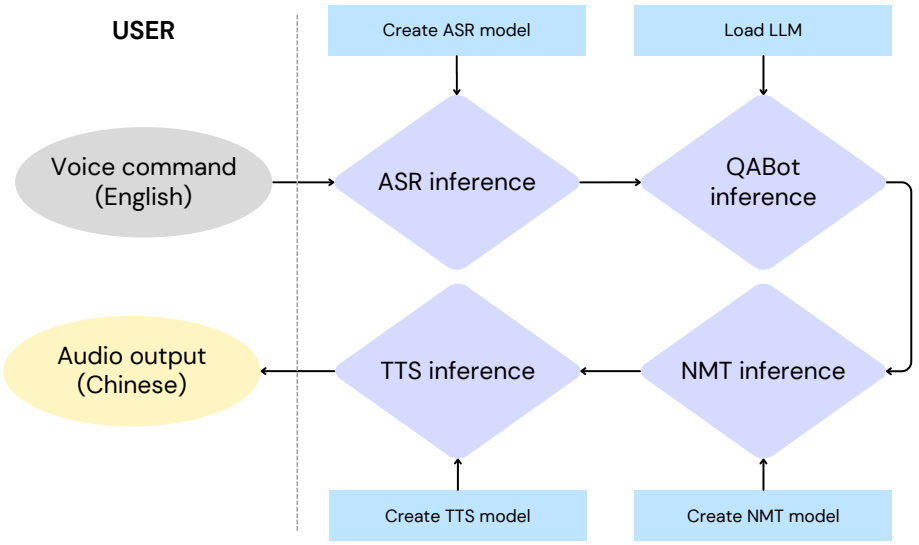


Figure 4.8: Simplified voice assistant workflow diagram.

dress. In BVA, a user issues a voice command or poses a question in one language (e.g., English), and the virtual voice assistant responds or performs tasks in another language (e.g., Chinese). The data processing involves four primary components: automatic speech recognition (ASR), language comprehension, language translation, and text-to-speech (TTS). Figure 4.9 illustrates the DAQL expression of BVA.

Recommender Recommendation systems [142] are among the most critical data center workloads and serve as the primary profit engines for cloud service companies. These systems offer personalized content and product suggestions based on a user's preferences and behavior. Initially, the system preprocesses the gathered user data (e.g., locations, clicks, and cursor movements) to extract meaningful features. These features then serve as input for one or more models.

Table 4.4: Key characteristics of the experimental platforms.

	CPU	GPU	GPU TOPS
Datacenter Server (Default)	16-core Intel Raptor Lake	RTX 3090	285
Datacenter Server (Upgraded)	16-core Intel Raptor Lake	RTX 4090	1321
User Device (Default) (Default)	6-core ARM Cortex-A78AE	1024-core NVIDIA Ampere	70
User Device (Downgraded)	4-core ARM Cortex-A57	128-core NVIDIA Maxwell	0.472

The recommendation system processes the model outputs, ranking and filtering the inference results, to deliver the final recommendations. Figure 4.10 illustrates the structure of the recommendation system.

4.4 Evaluation Platforms

PAMLB aims to evaluate and project the performance of personal assistant workloads on potential data processing models. Therefore, we use machines that either resemble the computational power of mobile devices or are custom-built to mimic data-center computers.

4.4.1 Hardware Configurations

Table 4.4 summarizes the key machine configurations that affect the workload performance of the hardware devices we evaluated in this paper. Each custom-built server, emulating a data center computer, contains an Intel Raptor Lake processor with 16 physical

cores, 24 threads, and a maximum frequency of 5.2 GHz. The server has 128 GB of main memory. We also equipped each server with an NVIDIA GPU. The default server features an RTX 3090 GPU based on the Ampere architecture, delivering up to 285 Tensor TFLOPs. To project future server technology improvements, we introduced an upgraded version of the server equipped with an RTX 4090 GPU based on the Ada Lovelace architecture, which delivers up to 1,321 Tensor TFLOPs. This upgraded server retains the original CPU since most workloads now run their core computation modules on GPUs.

For the default user device, we use NVIDIA's Jetson Orin NX series [178]. Jetson Orin is powered by a 6-core ARM Cortex-A78AE processor and features a scaled-down version of the NVIDIA GPU using the Ampere architecture, capable of 70 Tensor TFLOPs. In terms of CPU/GPU performance, Jetson Orin is highly competitive and is potentially more capable than most high-end mobile phones. Given that many existing mobile devices use proprietary hardware and operating systems, Jetson Orin's system offers greater flexibility in programming frameworks and software components. To project performance variations in mobile device configurations, we also tested a less powerful version using the Jetson Nano. The Jetson Nano houses a Quad-core ARM A57 processor and a 128-core Maxwell GPU, delivering up to just 0.5 Tensor TFLOPs. Nevertheless, the Jetson Nano's system configuration is representative of most low-end or IoT devices, allowing us to evaluate the performance of execution models on such devices. Communication between our user devices and servers occurs via WiFi 6 wireless links.

4.4.2 Software Systems

PAMLB and DAQL do not require specific versions or implementations of operators. They also impose no restrictions on where the workload performs operations or where the system stores the data. Despite their version differences, all the devices we evaluated in this paper run on Ubuntu-based operating systems. All operators in PAMLB have implementations in PyTorch with GPU or Tensor Core accelerated functions, except for preprocessing operators, which are implemented on the CPU. We have composed a Python program for each workload that runs on the user device to direct the application's control flow. The data center servers continuously run services that listen for requests from user devices and invoke optimized code to execute the corresponding operators.

It is important to note that when evaluating MDML models, it is often necessary to reconfigure the control flow of the driver program to allocate computing operations to different devices, such as mobile units or data centers. This reconfiguration can be time-consuming, especially if each setting requires manual reprogramming of the driver program. To address this issue, we adopt a straightforward emulation-based approach that allows for the quick estimation of MDML latencies and energy consumption by assessing their component-wise performance in mobile-only ODML settings or data center-only DCML settings.

4.5 Results

Running PAMLB on various platforms with various configurations in ODML, DCML, and MDML models, we found several insights that may guide the development and the system design of future personal assistant applications.

4.5.1 User-Perceived Latency

User-perceived latency, the end-to-end latency from when the user issues a request and when the user receives the results, is the most relevant performance metric to user experiences and the default latency this paper measures. Figure 4.11(a)-(g) presents the user-perceived latency in absolute numbers of PAMLB applications with default server and default user device settings. Figure 4.11(a)-(g) also breaks down the latency into three components: the execution time on the user device (OD time), the execution time on a datacenter server (DC time), and the time the computing resources on both devices are idle for data exchanging (DX time). Figure 4.11(h) summarizes results in relative latency. Parsing the results from user-perceived latency, we learned the following.

1. Data centers still play essential roles in personal assistants using modern technologies

Despite the advances in mobile device technologies and our user device offers up to 70 TOPS peak computation throughput, DCML and MDML that fully or partially rely on datacenters still offer the most competitive user-perceived latency in general. MDML provides a feasible alternative if the application has sensitive data or computation that cannot leverage the data center as each workload's optimal MDML configuration only adds 11% in user-perceived latency.

In contrast, modern technologies still fall short of supporting ODML well in general. Besides VQNL and Recommender where Jetson Orin shows some strengths, the same device can barely support the complete workflow of VQImg and VMF but fail on others due to memory constraints.

2. Opportunities of ODML: workloads with relatively low arithmetic intensity

Comparing the two cases, VQNL and Recommender, that ODML performs well with other workloads, we observed that ODML can deliver great user experiences on both workloads due to their low arithmetic intensities. The arithmetic intensity stands for the computation of each byte of data throughout the application.

Both VQNL and VQImg accept video streams as inputs, and both workloads idle for 0.05 seconds to wait for the first batch of video streams before the data process pipeline can start on servers. However, as VQImg requires the application to thoroughly scan all video clips and return all the occurrences of clips matching each querying image, the arithmetic intensity is significantly higher than VQNL. VQNL, in contrast, the high arithmetic intensity computation falls on the relatively small amount of visual input to figure out the target object but only requires the report if the video contains the target through a simple object recognition model. Therefore, the increased computation time on the device does not outweigh the data exchange overhead on the network in VQNL. Similarly, the Recommender's arithmetic intensity is only 4 per byte, and therefore, the data exchange plays a significant role in the latency and renders the recommendation system's kernel computation insignificant.

3. Mismatching between computation throughputs and latency

As PAMLB's design is hardware-independent, we evaluate these workloads using various combinations of machines described in Section 4.4. Figure 4.12(a)-(g) demonstrates user-perceived latency on the same set of experiments as Figure 4.11(a)-(g) using the upgraded server. Despite more than 4× theoretical TOPS that RTX 4090 offers over RTX 3090, the upgraded server achieves an 11% performance gain. The mismatching between performance gain and the computation power improvement comes from the limited memory bandwidth improvement on RTX 4090. RTX 4090's memory bandwidth is only 8% more than RTX 3090. As a result, models with large memory footprints cannot fully utilize the 1.6× more SIMD cores to achieve the claimed performance.

4. Model matters more than the upgrade of computing devices As PAMLB and DAQL is agnostic to hardware and software implementations, PAMLB allows the evaluator to easily assess the performance improvements on different implementations of the software components. Leveraging such flexibility, Figure 4.14 shows the effect of replacing the neural machine translation (NMT) (Line $19-24$) and text-to-speech (TTS) (Line $26-30$) from generalized (Ge) models to specialized (Sp) models using `LOAD MODEL` operators in BVA. We can achieve a 3.65× speedup in BVA, more significantly than simply replacing the hardware. However, the general models (NMT_{Ge} and TTS_{Ge}) still provide versatility (e.g., multilingual capabilities) across a range of scenarios. For personal assistant applications, the model choice offers an intriguing point for debate: the decision between smaller, more specialized models and larger, more generalized models. It prompts considerations of cost, computational efficiency, and meeting specific user needs in various contexts.

4.5.2 Energy-efficiency and Cost

The awareness of carbon-footprint, green energy, and most importantly, the cost of sustaining a service makes energy-efficiency the most important metric besides user-perceived latency. We derived several insights on the energy efficiency and cost for personal assistants using PAMLB in various settings.

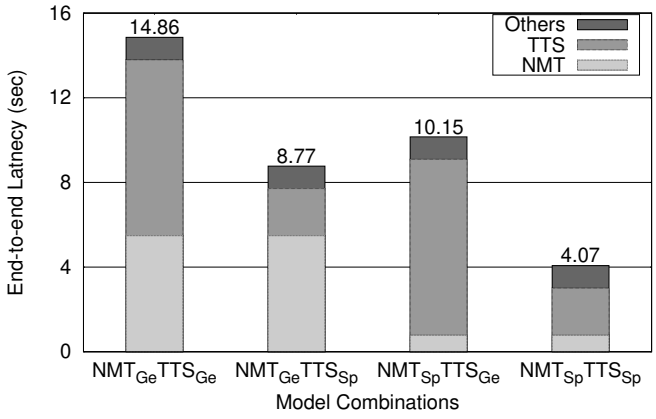


Figure 4.14: BVA with specialized/generalized NMT models.

5. ODML is the most carboon-footprint friendly – if device is powerful enongh but not hurt user experience

Figure 4.13 shows the total energy consumption, including the data center and device sides, when running workloads using various models on different machine configurations. ODML on our default device is always the best for total energy consumption if ODML can finish the workload. However, the user-perceived latency may not always be pleasant.

6. ODML can be the least carboon-footprint friendly – if device is too slow

A fallacy in system design is believing devices with lower power can lead to more energy efficiency. The downgraded user device, in our experience, consumes the lowest peak power among all devices at only 10 W. In contrast, the default user device consumes 15 W.

Table 4.5: Annual cost estimation of AI-driven applications (in USD).

Application \Data Management Model	**DCML (USD)**	**MDML (USD)**
VQImg	3.50E+05	2.57E+05
VQNL	3.27E+05	1.48E+05
Neural Machine Translation	9.77E+04	8.05E+04
Text-to-image Generation	2.57E+05	2.16E+05
Vehicle Motion Forecasting	3.50E+05	2.57E+05
QABot	4.44E+05	3.86E+05

However, observing the efficiency of ODML using the downgraded device in Figure 4.13, ODML on the downgraded user device always consumes more energy than the default user device. This configuration sometimes surpasses the energy consumption of MDML or even ODML.

7. MDML offers the best balance between user experience and data-center cost

MDML delivers performance comparable to DCML while consuming less than half the energy of DCML in 4 out of 7 workloads. Therefore, MDML offers the best balance between user experience and data-center cost. In addition, MDML has a strong benefit in reducing the operation cost of services while maintaining the quality of services. MDML reduces such cost by 41% compared with the DCML model. Throughout our experiments, we continuously measured the power consumption of our workloads on each machine to estimate the energy consumption.

We estimated the annual cost for most applications using hypothetical daily usage and electricity rates, providing a better understanding of energy consumption. Table 4.5 presents our estimates by converting the energy measurements from our experiments (ex-

Table 4.6: The mapping between DAQL line numbers and software components in various applications.

Application	**Description of Software Components in Each Application**				
	A	**B**	**C**	**D**	**E**
VQImg	2-3	5-6, 8-9	20-23	19	15
VQNL	2-3	5-7	10-15		
Text-to-image	3	4	6-11	14	
VMF	5	7-8	11-13	16	
QABot	3-8	11-12	3-8	15-16	19
BVA	4-10, 33-34	12-17, 37-38	19-24, 41-42	26-30, 45-48	
Recommender	5	7-8	11-14	15	

pressed in kilojoules, kJ) into kilowatt-hours, multiplying them by the number of requests per year, and applying the electric rates based on PG&E's charge (as of July 1, 2023).

While some applications, like Google Translate [76] with 200 million daily active users as of 2016, and the recently sensational ChatGPT chatbot, enjoy large user bases, not every application we discussed enjoys such popularity. To avoid overestimating costs with unrealistic numbers, we assumed each application receives 2 million daily requests in our estimation, similar to the emerging text-to-image applications[185]. The cost estimation table provides evidence that the MDML data management model effectively conserves energy and cost by offloading lightweight computing tasks to energy-efficient devices.

As each workload in PAMLB has multiple operators or stages, each workload has a rich design space for distributing the data processing pipeline elements in the MDML model. Using the insights learned from the previous section, we identified the arithmetic intensity

Table 4.7: The feasible data processing stages distribution in the MDML model.

	MDML		MDML2		MDML3		MDML4		MDML5		MDML6		MDML7	
	Device	Server	Device	Server	Device	Server	Device	Server	Device	Server	Device	Server	Device	Server
VQImg	A, B, E	B, C, D	B, C, D	A, B, E	A, B, C, D	B, E	B, E	A, B, C, D	A	B, C, D, E	B, C, D, E	A		
VQNL	A	B, C	B, C	A										
Text-to-image	A, B	C, D												
VMF	A, B, C	D	A	B, C, D	A, D	B, C	D	A, B, C	B, C, D	A	B, C	A, D		
QABot	A, B	C, D, E												
BVA	C, D	A, B	C	A, B, D	D	A, B, C	A, C, D	B	A, C	B, D	A, D	B, C	A	B, C, D
Recommender	B, C, D	A	B, C	A, D	D	A, B, C	A	B, C, D	A, D	B, C	A, B, C	D		

of software components as the critical factor determining the effectiveness of MDML and the decisions among ODML, MDML and DCML. Table 4.6 lists all stages in each workload and their corresponding lines in each workload's DAQL. Table 4.7 describes all feasible distributions of these stages in each workload.

4.5.3 Design Space Exploration of MDML

As each workload in PAMLB has multiple operators or stages, each workload has a rich design space in distributing the data processing pipeline elements in the MDML model. Table 4.6 lists all stages in each workload and their corresponding lines in each workload's DAQL. Table 4.7 describes all feasible distributions of these stages in each workload.

8. There is no clear guideline in finding the optimal MDML configurations. Figure 4.15 presents the complete design space exploration on all possible MDML configurations using the default server and the default user device. Depending on the degree of optimization in the implementation of each operator (e.g., some implementations can leverage on device accelerator to mitigate the performance gap), the performance difference between the device and server also varies. Due to the hard-to-predict device-server performance ratio and the trade-off between data exchange and the slow-down of computation performance, there is no clear rule in deriving the optimal MDML configuration. For example, in Recommender (Figure 4.15(f)), MDML4 and MDML6 have similar performance. However, MDML4 only uses the device to perform one stage, but MDML6 uses the device to perform 3 out of 4 stages. We believe the MDML model would require a runtime component similar to query optimizers and engines in modern database systems to adjust the distribution of operators dynamically.

4.6 Workload Optimizer on PAMLB: A Case Study

Using the first insight learned from Section 4.5 indicating the correlation between arithmetic intensity (AI) and the effectiveness of ODML, DCML, and MDML as well as the last insight regarding the complexity of optimizing MDML, we explored an idea of using the abstraction of DAQL and the arithmetic intensities of software components to project the most optimal model and the best MDML configuration.

Figure 4.16 illustrates our methodology. We assume the authoring language, potentially domain-specific, supports an abstraction similar to DAQL and provides a runtime. The runtime system can collect the hardware capability, analyze the arithmetic intensity of packages implementing corresponding software components, and retrieve the input size from metadata associated with file descriptors or input streams on peripherals.

The runtime system can estimate the effectiveness of executing each software component on any potential hardware from a straightforward formula using the arithmetic intensity, input size, hardware specifications as

1. $Execution\ Time(ET) = \dfrac{AI \times Input\ Size}{Hardware\ Throughput}$

2. $Energy\ Consumption(E) = ET \times Hardware\ Power\ Consumption$

To verify the potential of a runtime optimizer of personal assistants, the current prototype leverages publicly available arithmetic intensity numbers of software components PAMLB uses from [122, 74, 273, 247, 107] during the compilation stage.

Our methodology always effectively predicts the selection of DCML or ODML for latency or energy efficiency. We applied estimation formulas on each software component

that Table 4.6 shows and examined all valid configurations of MDML in Table 4.7. However, our estimation on MDML model does not always select the most optimal configuration. This limitation comes from the formula's heavy reliance on input size, where the later stages of the pipeline, characterized by relatively more minor intermediate results as input size, contribute less to the estimated execution time. This discrepancy prevents the effectiveness of our fine-grained estimation approach from guiding MDML configuration choices accurately.

4.7 Related Work

We have already reviewed data management paradiams for ML and ML benchmarks in Section 4.2 and MDML applications in Section 4.3. Here we surveyed: (1) techniques for acclerating on-device inference and (2) DB systems for machine learning.

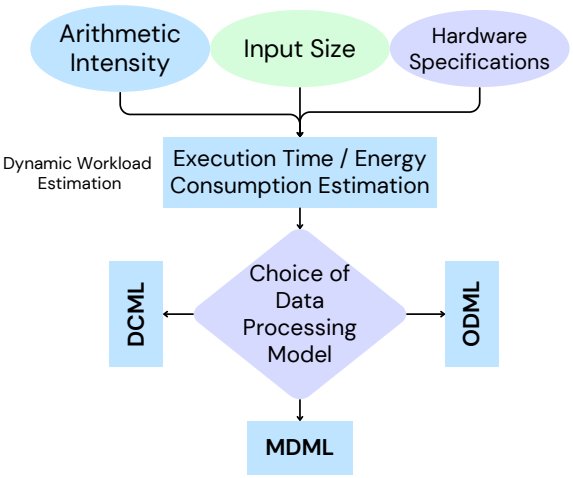


Figure 4.16: Dynamic workload estimation flowchart.

Accelerating on-device inference. More recently, the emergence of large, powerful deep-learning models such as GPT-4 [184], Llama [234, 235], and Stable Diffusion [206] has sparked intense research interest in enabling rapid on-device inference of such models. Quantization is a primary technique that allows large models to fit within the memory constraints of local devices by compressing 16-bit models to execute inference locally at just 8-

or 4-bits [46, 47, 66]. Notably, the most recent advancements, QLoRa [48] and OPTQ [66], have made on-device inference of substantial language models feasible.

Another significant technique for on-device inference is knowledge distillation [94] (for a comprehensive survey, see [78]), which involves transferring the knowledge acquired by a large deep neural network to a much smaller model. For instance, Alpaca [229] and Vicuna [33] are two recent successful instances where knowledge from the powerful large language model GPT-4 [184] was transferred to smaller architectures with 7B or 13B parameters.

DB systems for ML. While DB systems were traditionally optimized primarily for transaction processing and analytical querying, there's a growing need to support ML workloads. In-database machine learning has become a focal point to reduce the extensive data transmission between storage and computation. Examples of such systems include Amazon Redshift [13], BlazingSQL [179], and Google Cloud BigQuery [62], among others. As ML workflows increase in complexity, systems like MLbase [129], MLib [161], and SageMaker [141] aim to simplify the ML pipeline by offering users high-level abstractions and automating numerous underlying tasks, such as feature extraction and model selection. For deep learning, Nautilus [171] supports deep transfer learning (DTL) with multi-query optimization; Cerebro [172] introduces a novel data system for deep model selection, leveraging task- and data-parallelism; HiveMind [173] enhances multi-model deep learning workloads using multi-model operator fusion.

4.8 Conclusion

In conclusion, we present PAMLB and assesses PAMLB and DAQL across diverse data processing models and workloads. PAMLB encompasses a comprehensive suite of personal assistant workloads across various domains, while DAQL facilitates the evaluation of intricate multimodal operations. Moreover, DAQL offers optimization insights for system designers with its generic and descriptive approach. Through rigorous evaluations, MDML emerges as a formidable data management model for personal assistant applications, distinguishing itself in latency, service quality, and cost-effectiveness. MDML's performance is commendable, being only 11% slower than DCML, yet it achieves superior energy efficiency, conserving 41% more energy compared to DCML. As the realm of AI and personal assistant applications continues to advance, the insights furnished by PAMLB will undoubtedly constitute a precious cornerstone for subsequent research endeavors.

```
/* Assume the voice_command is an audio format */
DECLARE voice_command STRING DEFAULT 'How_is_the_weather_in_California?';

LOAD MODEL `asr_model`
OPTIONS (
  model_type='TRANSFORMER',
  input_audio_col='audio_input',
  output_col='transcription',
  enable_timestamps BOOLEAN,
  enable_word_level BOOLEAN) AS asr_model;

LOAD MODEL `qa_model`
OPTIONS (
  model_type='TRANSFORMER',
  input_seq_length=128,
  language='en'
) AS qa_model;

LOAD MODEL `nmt_model`
OPTIONS (
  model_type='TRANSFORMER',
  input_seq_length=128,
  language='en'
) AS nmt_model;

LOAD MODEL `tts_model`
OPTIONS (
  text_input_col='text_input',
  audio_output_col='audio_output'
) AS tts_model;

/* ASR: Convert speech to text */
SELECT PREDICT(MODEL asr_model, voice_command) AS transcribed_text
FROM voice_command;

/* QABot: Ask a question based on transcribed text */
SELECT PREDICT(MODEL qa_model, transcribed_text) AS answer
FROM transcribed_text;

/* NMT: Translate the question into another language */
SELECT PREDICT(MODEL nmt_model, answer) AS translated_answer
FROM answer;

/* TTS: Generate speech from the translated answer */
SELECT PREDICT(MODEL tts_model, translated_answer) AS voice_response
FROM translated_answer;

SELECT POST_PROCESS(voice_response) AS final_response;
```

Figure 4.9: The DAQL query for BVA.

```
/*
 *  Assume dataset contains:
 *  raw_data=[user_id, item_id, ratings, genre]
 */
TRANSFORM(raw_data) AS processed_data;

LOAD MODEL 'rec_model'
OPTIONS(model_type='matrix_factorization') AS rec_model;

/* Make predictions */
SELECT user_id, item_id, predicted_score
FROM PREDICT(MODEL rec_model,
  (SELECT user_id, item_id, ratings, genre FROM processed_data)
) AS recommendations
LIMIT K;
```

Figure 4.10: The DAQL query for Recommender.

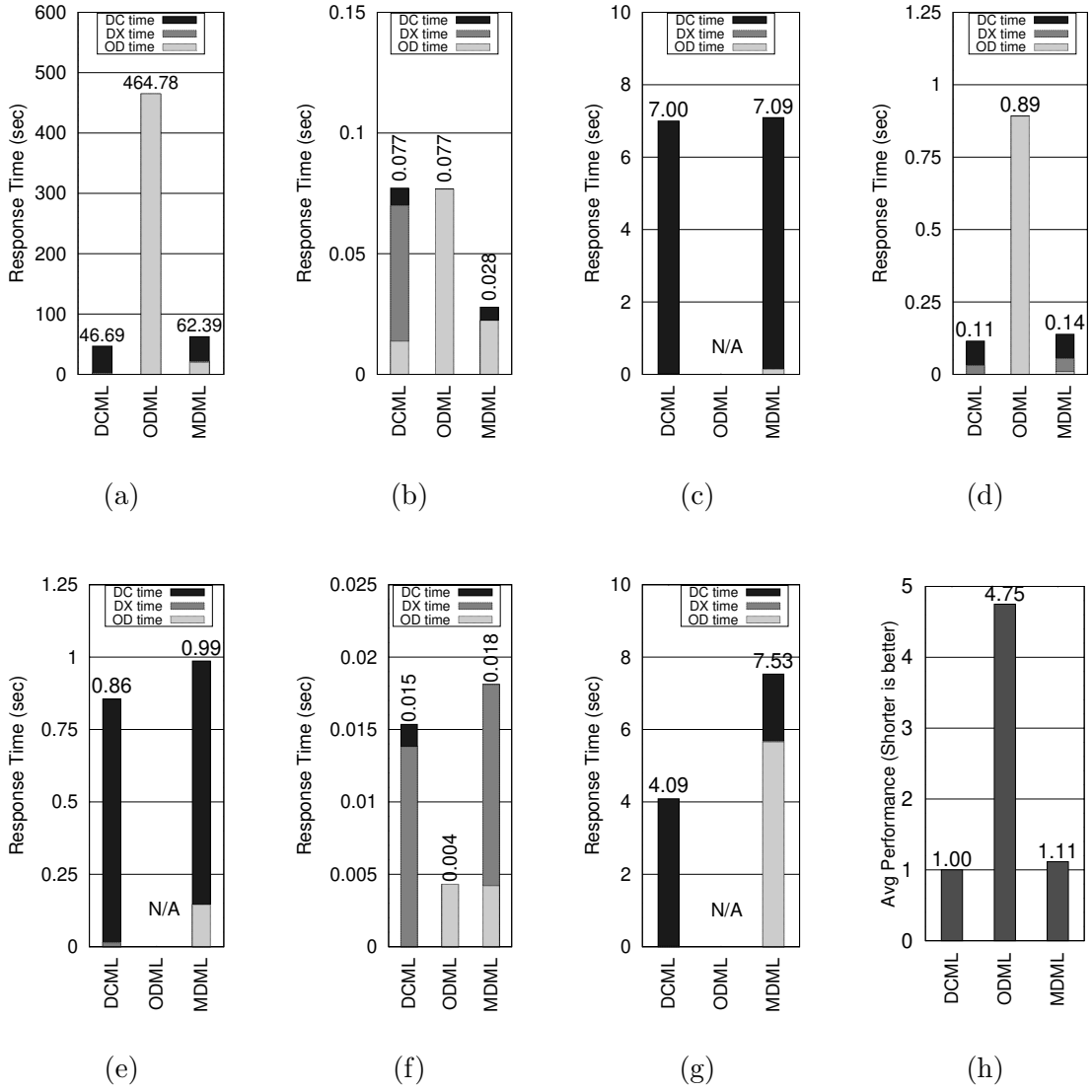


Figure 4.11: Response time for (a) VQImg, (b) VQNL, (c) Text-to-image, (d) VMF, (e) QABot, (f) Recommender, (g) BVA applications, and (h) Average user-perceived latency across all applications.

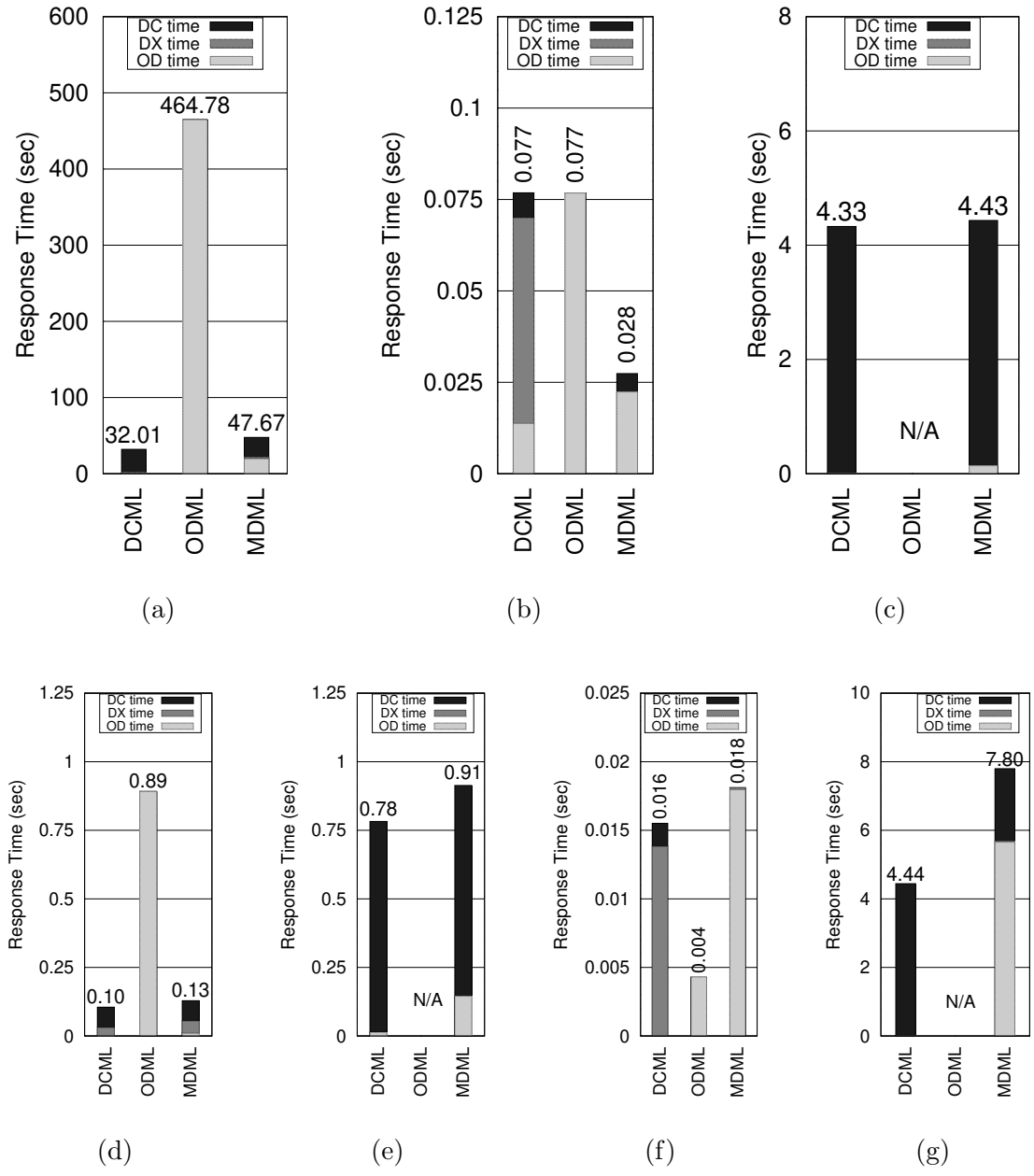


Figure 4.12: Response time of upgraded server for (a) VQImg, (b) VQNL, (c) Text-to-image, (d) VMF, (e) QABot, (f) Recommender, and (g) BVA applications.

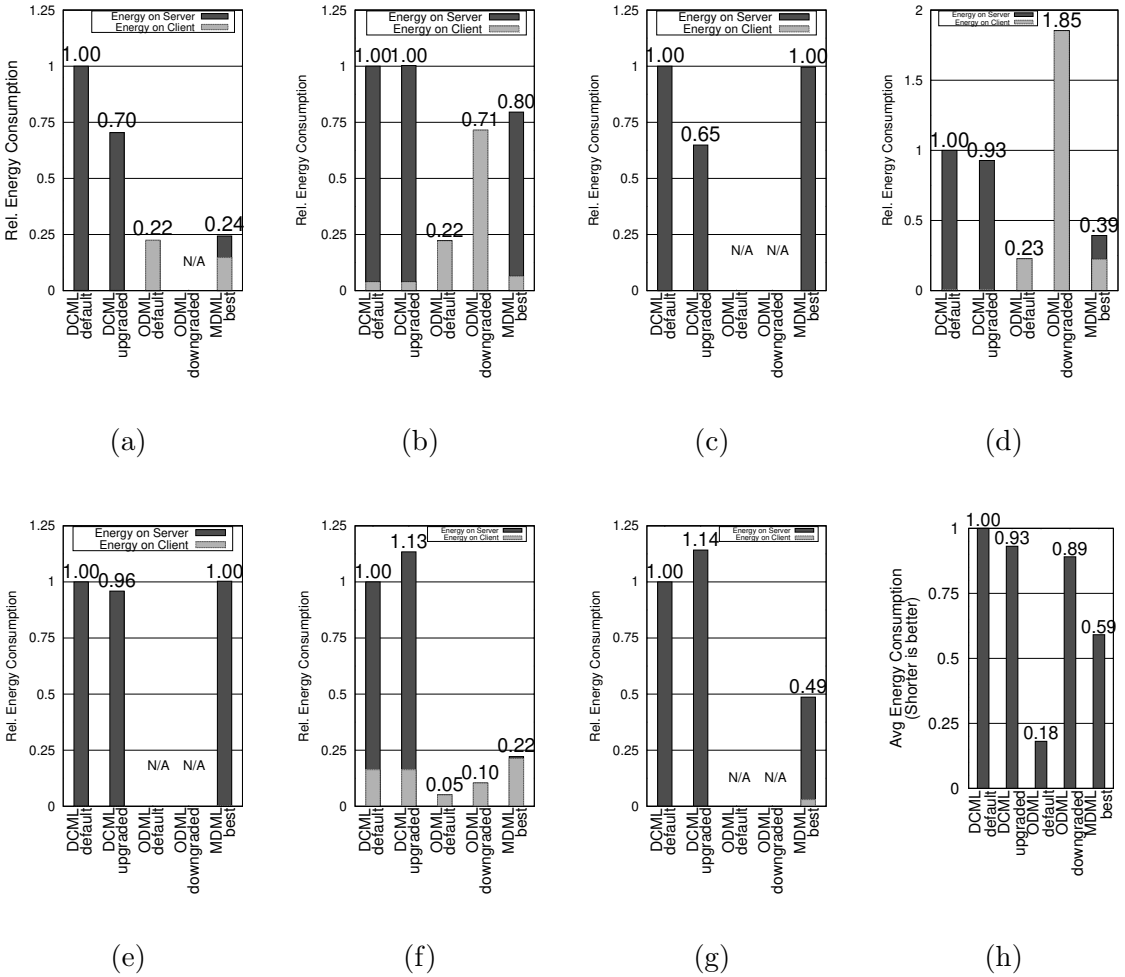


Figure 4.13: The relative energy consumption for (a) VQImg, (b) VQNL, (c) Text-to-image, (d) VMF, (e) QABot, (f) Recommender, (g) BVA applications, and (h) Average energy consumption across all applications.

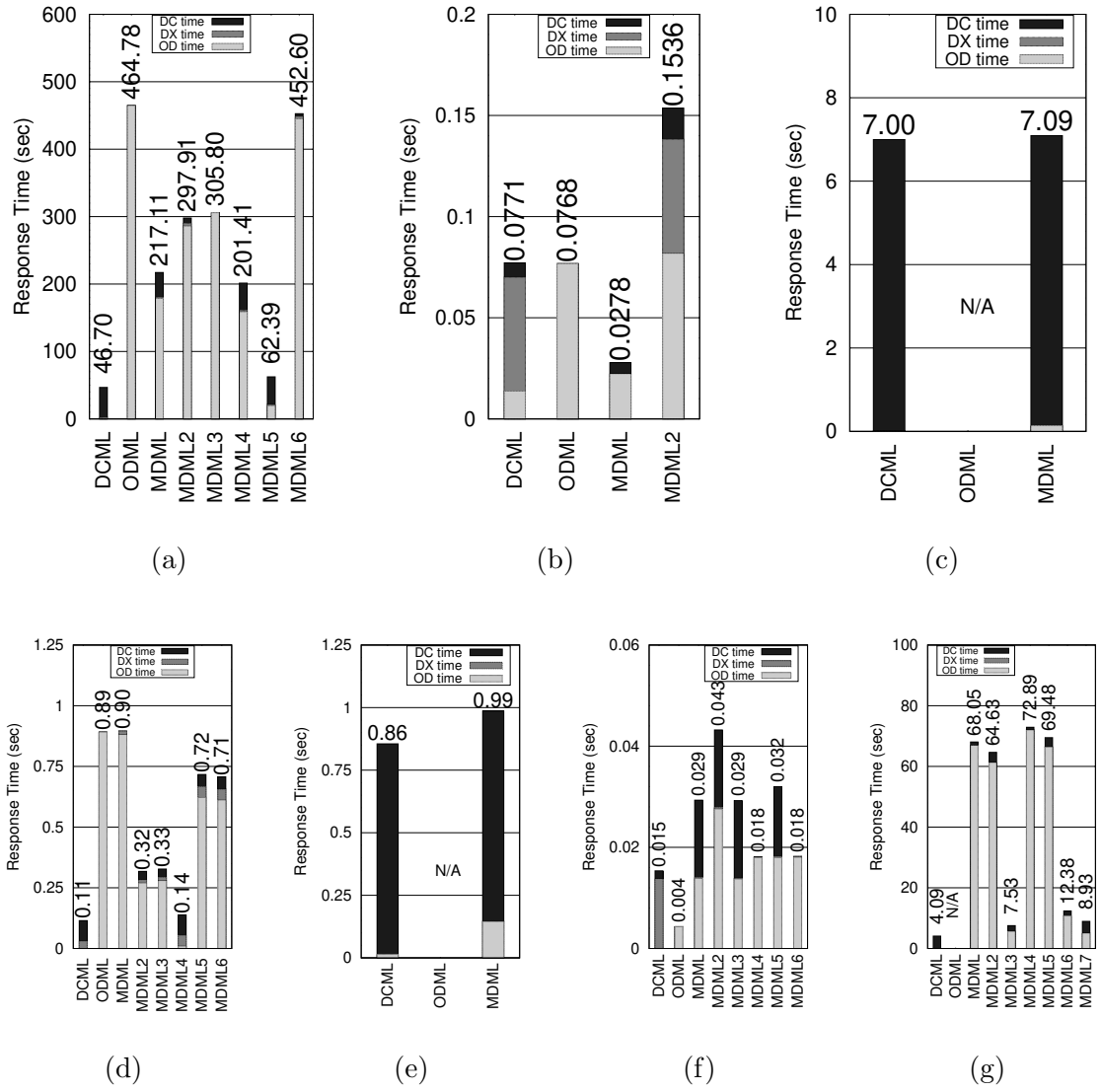


Figure 4.15: User-perceived latency with multiple MDML configurations using default server and default user device for (a) VQImg, (b) VQNL, (c) Text-to-image, (d) VMF, (e) QABot, (f) Recommender, and (g) BVA applications.

Chapter 5

Conclusions

The advancement of technology is closely related to the demand for real-world applications. Artificial Intelligence and generative AI have attracted a lot of attention in recent years, followed by the rise of hardware accelerators to provide more powerful and faster computation while taking into account energy savings. However, whether we have fully explored these AI/ML hardware accelerators in the process is a topic worth researching. This dissertation addresses this research question through the following works:

Firstly, VS, mentioned in Chapter 2, explores the data supply pipeline to ensure a smooth and efficient delivery of data to powerful computing units for optimal performance. VS reduces unnecessary data through in-storage processing thus mitigating data traffic and minimizing data transformation overhead via a hardware-software co-designed approach. By dynamically adjusting data resolutions and performing quality control within the storage device, VS enhances overall system performance, delivering an average speedup of 1.52× compared to conventional approximate computing frameworks.

Secondly, TCUDB, mentioned in Chapter 3, explores the portability of hardware accelerators, particularly whether domains outside of AI/ML can benefit from these technologies. By leveraging Tensor Cores, TCUDB implements an efficient database query engine that accelerates relational database queries. Through revisiting application algorithms and data layout for emerging hardware accelerators, TCUDB achieves up to 288× speedup compared to the baseline GPU-accelerated DB engine. It shows compelling use cases, including matrix multiplication, entity matching, and PageRank, demonstrating its effectiveness and potential in various domains.

Lastly, PAMLB, mentioned in Chapter 4, investigates whether current hardware is sufficient for evolving AI-assisted applications or if additional accelerators are needed. It considers the multi-device interaction from a holistic system design perspective for personal assistant applications. By leveraging DAQL, workloads are represented as a series of queries, providing a systematic way to assess performance across diverse data management models. Key findings include: (1) Despite a more than 4× improvement in compute capability (TOPS) with upgraded hardware, application performance improved by only 11% in terms of latency due to limited memory bandwidth enhancements. (2) The choice of machine learning model has a greater impact on end-to-end latency than upgrading the computing device, with specialized models providing over a 3× speedup compared to generalized ones. Overall, MDML was identified as a competitive data management model for personal assistant applications, adding only 11% user-perceived latency while saving 41% energy compared to DCML. PAMLB serves as a valuable foundation for future research in this domain.

Besides these proposed works, we believe there are more research directions worth diving into, such as efficient data representation for the system to consume, a compiler for intelligently distributing tasks across multiple devices/platforms, fine-tuning compute kernels for each ML adjacent component, and so on. Summarizing the above research directions, I believe we will be able to build a better user experience by maximizing the use of these hardware accelerators.

Bibliography

[1] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[2] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD*, pages 967–980. ACM, 2008.

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

[4] Christopher Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. Levelheaded: A unified engine for business intelligence and linear algebra querying. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 449–460. IEEE, 2018.

[5] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–492, Feb 2017.

[6] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems*, 35:23716–23736, 2022.

[7] C. Alvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computers*, 54(7):922–927, July 2005.

[8] Moustafa Alzantot, Yingnan Wang, Zhengshuang Ren, and Mani B. Srivastava. Rstensorflow: Gpu enabled tensorflow for deep learning on commodity android devices. In

Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications, EMDL '17, page 7–12, New York, NY, USA, 2017. Association for Computing Machinery.

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

[10] AMD Inc. AMD FirePro DirectGMA. `http://developer.amd.com/community/blog/2014/09/08/amd-firepro-gpus-directgma/`, 2014.

[11] Rasmus Resen Amossen and Rasmus Pagh. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory*, pages 121–126. Association for Computing Machinery, 2009.

[12] Apple Inc. Apple M1. `https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/`, 11 2020.

[13] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. Amazon redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2205–2217, 2022.

[14] B. He, M. Lu, K. Yang, R. Fang, N. Govindaraju, Q. Luo, and P. Sander. Gpudb source code., 2013.

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

[16] Peter Bakkum and Kevin Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103. Association for Computing Machinery, 2010.

[17] Ravi Bansal and Sandip Chakraborty. Visual content based video retrieval on natural language queries. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 212–219, New York, NY, USA, 2019. Association for Computing Machinery.

[18] BlazingSQL Inc. BlazingDB. `https://blazingsql.com`, 2015.

[19] Paulo Blikstein and Marcelo Worsley. Multimodal learning analytics and education data mining: Using computational technologies to measure complex learning tasks. *Journal of Learning Analytics*, 3(2):220–238, 2016.

[20] 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.

[21] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[22] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. Probability type inference for flexible approximate programming. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 470–487, New York, NY, USA, 2015. ACM.

[23] Sebastian Breβ, Henning Funke, and Jens Teubner. Robust query processing in co-processor-accelerated databases. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1891–1906, 2016.

[24] Sebastian Breβ, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal*, 27(6):797–822, December 2018.

[25] Sebastian Breß and Gunter Saake. Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms. *Proc. VLDB Endow.*, 6(12):1398–1403, 2013.

[26] Robert Brijder, Floris Geerts, Jan Van Den Bussche, and Timmy Weerwag. On the expressive power of query languages for matrices. *ACM Trans. Database Syst.*, 44(4), October 2019.

[27] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[28] M. Burtscher and P. Ratanaworabhan. Fpc: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers*, 58(1):18–31, Jan 2009.

[29] Nil Goksel Canbek and Mehmet Emin Mutlu. On the track of artificial intelligence: Learning with intelligent personal assistants. *Journal of Human Sciences*, 13(1):592–601, 2016.

[30] 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.

[31] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. *CoRR*, abs/1603.02754, 2016.

[32] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(8):1196–1208, Aug 2010.

[33] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023.

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

[35] H. Cho, L. Leem, and S. Mitra. Ersa: Error resilient system architecture for probabilistic applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(4):546–558, April 2012.

[36] I. Stephen Choi and Yang-Suk Kee. Energy efficient scale-in clusters with in-storage processing for big-data analytics. In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, pages 265–273, New York, NY, USA, 2015. ACM.

[37] W. G. Choi, D. Kim, H. Roh, and S. Park. Ourrocks: offloading disk scan directly to gpu in write-optimized database system. *IEEE Transactions on Computers*, pages 1–1, 2020.

[38] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

[39] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. Entity resolution in the web of data. *Synthesis Lectures on the Semantic Web*, 5(3):1–122, 2015.

[40] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. Hardware-conscious query processing in gpu-accelerated analytical engines. In *Proceesings of the 9th Biennial Conference on Innovative Data Systems Research*, number CONF, 2019.

[41] Insoo Chung, Byeongwook Kim, Yoonjung Choi, Se Jung Kwon, Yongkweon Jeon, Baeseong Park, Sangha Kim, and Dongsoo Lee. Extremely low bit transformer quantization for on-device neural machine translation, 2020.

[42] 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. Association for Computing Machinery, 2019.

[43] Tim Davis, Michel Pelletier, and Scott Kolodziej. Graphblas standard. `https://github.com/GraphBLAS`, 2017.

[44] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 497–508, New York, NY, USA, 2010. ACM.

[45] Shaleen Deep, Xiao Hu, and Paraschos Koutris. Fast join project query evaluation using matrix multiplication. In *SIGMOD*, pages 1213–1223. Association for Computing Machinery, 2020.

[46] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3.int8(): 8-bit matrix multiplication for transformers at scale. In *NeurIPS*, 2022.

[47] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. In *ICLR*. OpenReview.net, 2022.

[48] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *CoRR*, abs/2305.14314, 2023.

[49] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E Lee. Aggregation support for modern graph analytics in tigergraph. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 377–392, 2020.

[50] Sauptik Dhar, Junyao Guo, Jiayi Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. A survey of on-device machine learning: An algorithms and learning theory perspective. *ACM Transactions on Internet of Things*, 2(3):1–49, 2021.

[51] 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.

[52] Jaeyoung Do and Jignesh M. Patel. Join processing for flash ssds: Remembering past lessons. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 1–8, 2009.

[53] Oksana Dolmatova, Nikolaus Augsten, and Michael H Böhlen. A relational matrix algebra and its implementation in a column store. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2573–2587, 2020.

[54] Xin Luna Dong and Divesh Srivastava. Big data integration. In *2013 IEEE 29th international conference on data engineering (ICDE)*, pages 1245–1248. IEEE, 2013.

[55] 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.

[56] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. AIDA: Abstraction for advanced in-database analytics. *PVLDB*, 11(11):1400–1413, 2018.

[57] Ahmed K Elmagarmid, Panagiotis G Ipeirotis, and Vassilios S Verykios. Duplicate record detection: A survey. *IEEE Transactions on knowledge and data engineering*, 19(1):1–16, 2006.

[58] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 449–460, Washington, DC, USA, 2012. IEEE Computer Society.

[59] Scott Ettinger, Shuyang Cheng, Benjamin Caine, Chenxi Liu, Hang Zhao, Sabeek Pradhan, Yuning Chai, Ben Sapp, Charles R. Qi, Yin Zhou, Zoey Yang, Aur'elien Chouard, Pei Sun, Jiquan Ngiam, Vijay Vasudevan, Alexander McCauley, Jonathon Shlens, and Dragomir Anguelov. Large scale interactive motion forecasting for autonomous driving: The waymo open motion dataset. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 9710–9719, October 2021.

[60] Jian Fang, Yvo TB Mulder, Jan Hidders, Jinho Lee, and H Peter Hofstee. In-memory database acceleration on fpgas: a survey. *The VLDB Journal*, 29(1):33–59, 2020.

[61] Y. Fang, H. Li, and X. Li. A fault criticality evaluation framework of digital systems for error tolerant video applications. In *2011 Asian Test Symposium*, pages 329–334, Nov 2011.

[62] Sérgio Fernandes and Jorge Bernardino. What is bigquery? In *Proceedings of the 19th International Database Engineering & Applications Symposium*, pages 202–203, 2015.

[63] 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.

[64] Huber Flores, Petteri Nurmi, and Pan Hui. Ai on the move: From on-device to on-multi-device. In *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 310–315. IEEE, 2019.

[65] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

[66] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. OPTQ: accurate quantization for generative pre-trained transformers. In *ICLR*. OpenReview.net, 2023.

[67] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1603–1618, 2018.

[68] Luca Gagliardelli, Giovanni Simonini, Domenico Beneventano, and Sonia Bergamaschi. SparkER: Scaling entity resolution in spark. In *EDBT 2019: 22nd International Conference on Extending Database Technology*, 2019.

[69] S. Ganapathy, A. Teman, R. Giterman, A. Burg, and G. Karakonstantis. Approximate computing with unreliable dynamic memories. In *2015 IEEE 13th International New Circuits and Systems Conference (NEWCAS)*, pages 1–4, June 2015.

[70] Jim Gao. Machine learning applications for data center optimization. 2014.

[71] Zhipeng Gao, Shan Sun, Yinghan Zhang, Zijia Mo, and Chen Zhao. Edgesp: Scalable multi-device parallel dnn inference on heterogeneous edge clusters. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 317–333. Springer, 2021.

[72] Pedram Ghodsnia. An in-gpu-memory column-oriented database for processing analytical workloads. pages 54–59, 01 2012.

[73] David F Gleich. Pagerank beyond the web. *siam REVIEW*, 57(3):321–363, 2015.

[74] Alicia Golden, Samuel Hsia, Fei Sun, Bilge Acun, Basil Hosmer, Yejin Lee, Zachary DeVito, Jeff Johnson, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. Generative ai beyond llms: System implications of multi-modal generation, 2023.

[75] Google LLC. MUM: A new AI milestone for understanding information. `https://blog.google/products/search/introducing-mum/`, 2021. Accessed: June 27, 2024.

[76] Google LLC. Ten years of Google Translate. `https://blog.google/products/translate/ten-years-of-google-translate/`, 4 2016.

[77] Google LLC. BigQuery ML. `https://cloud.google.com/bigquery-ml/docs`, 7 2018.

[78] Jianping Gou, Baosheng Yu, Stephen John Maybank, and Dacheng Tao. Knowledge distillation: A survey. *CoRR*, abs/2006.05525, 2020.

[79] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, pages 325–336. ACM, 2006.

[80] Naga K Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *SIGMOD*, pages 215–226. ACM, 2004.

[81] Kristen Grauman, Andrew Westbury, Eugene Byrne, Zachary Chavis, Antonino Furnari, Rohit Girdhar, Jackson Hamburger, Hao Jiang, Miao Liu, Xingyu Liu, Miguel Martin, Tushar Nagarajan, Ilija Radosavovic, Santhosh Kumar Ramakrishnan, Fiona Ryan, Jayant Sharma, Michael Wray, Mengmeng Xu, Eric Zhongcong Xu, Chen Zhao, Siddhant Bansal, Dhruv Batra, Vincent Cartillier, Sean Crane, Tien Do, Morrie Doulaty, Akshay Erapalli, Christoph Feichtenhofer, Adriano Fragomeni, Qichen Fu, Abrham Gebreselasie, Cristina González, James Hillis, Xuhua Huang, Yifei Huang, Wenqi Jia, Weslie Khoo, Jáchym Kolář, Satwik Kottur, Anurag Kumar, Federico Landini, Chao Li, Yanghao Li, Zhenqiang Li, Karttikeya Mangalam, Raghava Modhugu, Jonathan Munro, Tullie Murrell, Takumi Nishiyasu, Will Price, Paola Ruiz, Merey Ramazanova, Leda Sari, Kiran Somasundaram, Audrey Southerland, Yusuke Sugano, Ruijie Tao, Minh Vo, Yuchen Wang, Xindi Wu, Takuma Yagi, Ziwei Zhao, Yunyi Zhu, Pablo Arbeláez, David Crandall, Dima Damen, Giovanni Maria Farinella, Christian Fuegen, Bernard Ghanem, Vamsi Krishna Ithapu, C. V. Jawahar, Hanbyul Joo, Kris Kitani, Haizhou Li, Richard Newcombe, Aude Oliva, Hyun Soo Park, James M. Rehg, Yoichi Sato, Jianbo Shi, Mike Zheng Shou, Antonio Torralba, Lorenzo Torresani, Mingfei Yan, and Jitendra Malik. Ego4D: Around the World in 3,000 Hours of Egocentric Video. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 18995–19012, June 2022.

[82] L.M. Grupp, A.M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-42, pages 24 –33, 12 2009.

[83] 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.

[84] C. Guo and H. Chen. In-memory join algorithms on gpus for large-data. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1060–1067, 2019.

[85] Peizhen Guo, Bo Hu, and Wenjun Hu. Mistify: Automating {DNN} model porting for {On-Device} inference at the edge. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 705–719, 2021.

[86] Tian Guo. Cloud-based or on-device: An empirical study of mobile deep inference. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 184–190. IEEE, 2018.

[87] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy. Impact: Imprecise adders for low-power approximate computing. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 409–414, Aug 2011.

[88] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, 2018.

[89] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34, 2009.

[90] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524, 2008.

[91] Jiong He, Mian Lu, and Bingsheng He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *VLDB*, 6(10):889–900, 2013.

[92] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn, 2018.

[93] Joseph M Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. The madlib analytics library. *Proceedings of the VLDB Endowment*, 5(12), 2012.

[94] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531, 2015.

[95] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 199–212. ACM, 2011.

[96] Pedro Holanda and Hannes Mühleisen. Relational queries with a tensor processing unit. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. Association for Computing Machinery, 2019.

[97] Yihan Hu, Jiazhi Yang, Li Chen, Keyu Li, Chonghao Sima, Xizhou Zhu, Siqi Chai, Senyao Du, Tianwei Lin, Wenhai Wang, et al. Planning-oriented autonomous driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 17853–17862, 2023.

[98] 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 (Best Paper Honorable Mention), 2019.

[99] Dylan Hutchison, Bill Howe, and Dan Suciu. LaraDB: A minimalist kernel for linear and relational algebra computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, BeyondMR'17, 2017.

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

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

[102] Fuad Jamour, Ibrahim Abdelaziz, Yuanzhao Chen, and Panos Kalnis. Matrix algebra framework for portable, scalable and efficient query engines for rdf graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19. Association for Computing Machinery, 2019.

[103] Dawid Stachowiak Jarosław Kochanowicz, Maciej Domagała and Krzysztof Dziedzic. Diffusion models in practice. `https://deepsense.ai/diffusion-models-in-practice-part-1-the-tools-of-the-trade/`, 2023.

[104] Djordje Jevdjic, Karin Strauss, Luis Ceze, and Henrique S. Malvar. Approximate storage of compressed and encrypted videos. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 361–373, New York, NY, USA, 2017. ACM.

[105] Yanqin Jin, Hung-Wei Tseng, Steven Swanson, and Yannis Papakonstantinou. KAML: A Flexible, High-Performance Key-Value SSD. In *23th International Symposium on High Performance Computer Architecture*, HPCA 2017, 2017.

[106] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten lessons from three generations shaped google's tpuv4i : Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2021.

[107] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings, 2023.

[108] Norman P Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A Domain-specific Supercomputer for Training Deep Neural Networks. *Communications of the ACM*, 63(7):67–78, 2020.

[109] 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.

[110] 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, 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, and Jonathan Ross. In-Datacenter Performance Analysis of a Tensor Processing Unit. 2017.

[111] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 23:1–23:12, New York, NY, USA, 2016. ACM.

[112] 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.

[113] A. B. Kahng and S. Kang. Accuracy-configurable adder for approximate arithmetic designs. In *DAC Design Automation Conference 2012*, pages 820–825, June 2012.

[114] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. GPU join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 55–62, 2012.

[115] 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.

[116] Tomas Karnagel, René Müller, and Guy M Lohman. Optimizing gpu-accelerated group-by and aggregation. *ADMS@ VLDB*, 8:20, 2015.

[117] Z. M. Kedem, V. J. Mooney, K. K. Muntimadugu, and K. V. Palem. An approach to energy-error tradeoffs in approximate ripple carry adders. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 211–216, Aug 2011.

[118] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27(3):42–52, September 1998.

[119] D. S. Khudia and S. Mahlke. Harnessing soft computations for low-budget fault tolerance. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–330, Dec 2014.

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

[121] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Amir Wated, Emmett Witchel, and Mark Silberstein. GPUnet: Networking abstractions for gpu programs. In *OSDI*, pages 6–8, 2014.

[122] Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W. Mahoney, Sophia Shao, and Amir Gholami. Full stack optimization of transformer inference. In *Architecture and System Support for Transformer Models (ASSYST @ISCA 2023)*, 2023.

[123] Y. Kim, S. Behroozi, V. Raghunathan, and A. Raghunathan. Axserbus: A quality-configurable approximate serial bus for energy-efficient sensing. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, July 2017.

[124] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. μLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[125] Kinetica DB Inc. Kinetica. `https://www.kinetica.com/`, 2016.

[126] 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.

[127] Pradap Konda, Sanjib Das, Paul Suganthan GC, AnHai Doan, Adel Ardalan, Jeffrey R Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, et al. Magellan: Toward building entity matching management systems. *Proceedings of the VLDB Endowment*, 9(12):1197–1208, 2016.

[128] 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.

[129] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. Mlbase: A distributed machine-learning system. In *Cidr*, volume 1, pages 2–1, 2013.

[130] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74, 1991.

[131] 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*, PLDI '16, pages 161–176, New York, NY, USA, 2016. ACM.

[132] I. Lazaridis and S. Mehrotra. Approximate selection queries over imprecise data. In *Proceedings. 20th International Conference on Data Engineering*, pages 140–151, April 2004.

[133] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[134] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. On-device neural net inference with mobile gpus, 2019.

[135] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[136] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *PVLDB*, 9(14):1647–1658, 2016.

[137] Xuanhua Li and Donald Yeung. Application-level correctness and its impact on fault tolerance. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 181–192, Washington, DC, USA, 2007. IEEE Computer Society.

[138] Youjie Li, Jongse Park, Mohammad Alian, Yifan Yuan, Zheng Qu, Peitian Pan, Ren Wang, Alexander Gerhard Schwing, Hadi Esmaeilzadeh, and Nam Sung Kim. A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks. In *51th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 2018, 2018.

[139] J. Liang, J. Han, and F. Lombardi. New metrics for the reliability of approximate and probabilistic adders. *IEEE Transactions on Computers*, 62(9):1760–1771, Sept 2013.

[140] Paul Pu Liang, Yiwei Lyu, Xiang Fan, Zetian Wu, Yun Cheng, Jason Wu, Leslie Yufan Chen, Peter Wu, Michelle A Lee, Yuke Zhu, et al. Multibench: Multiscale benchmarks for multimodal representation learning. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[141] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, et al. Elastic machine learning algorithms in amazon sagemaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 731–737, 2020.

[142] Greg Linden, Brent Smith, and Jeremy York. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*, 7(1):76–80, 2003.

[143] Kaiyang Liu, Jun Peng, Heng Li, Xiaoyong Zhang, and Weirong Liu. Multi-device task offloading with time-constraints for energy efficiency in mobile cloud computing. *Future Generation Computer Systems*, 64:1–14, 2016.

[144] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single shot MultiBox detector. In *Computer Vision – ECCV 2016*, pages 21–37. Springer International Publishing, 2016.

[145] 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. IEEE, 2016.

[146] Yu-Chia Liu and Hung-Wei Tseng. NDS: N-Dimensional Storage. In *54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 2021 (Best Paper Nomination), 2021.

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

[148] Andre Luckow, Kartik Rattan, and Shantenu Jha. Exploring task placement for edge-to-cloud applications using emulation. In *2021 IEEE 5th International Conference on Fog and Edge Computing (ICFEC)*, pages 79–83. IEEE, 2021.

[149] Camillo Lugaresi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, Esha Uboweja, Michael Hays, Fan Zhang, Chuo-Ling Chang, Ming Guang Yong, Juhyun Lee, Wan-Teh Chang, Wei Hua, Manfred Georg, and Matthias Grundmann. Mediapipe: A framework for building perception pipelines. *CoRR*, abs/1906.08172, 2019.

[150] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable linear algebra on a relational database system. *IEEE Transactions on Knowledge and Data Engineering*, 31(7):1224–1238, 2019.

[151] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Pump up the volume: Processing large data on gpus with fast interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1633–1649, 2020.

[152] 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.

[153] D. Mahajan, A. Yazdanbaksh, J. Park, B. Thwaites, and H. Esmaeilzadeh. Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 66–77, June 2016.

[154] Divya Mahajan, Joon Kyung Kim, Jacob Sacks, Adel Ardalan, Arun Kumar, and Hadi Esmaeilzadeh. In-rdbms hardware acceleration of advanced analytics. *PVLDB*, 11(11), 2018.

[155] 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.

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

[157] Elman Mansimov, Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Generating images from captions with attention. *arXiv preprint arXiv:1511.02793*, 2015.

[158] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

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

[160] Jiayuan Meng, S. Chakradhar, and A. Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, May 2009.

[161] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The journal of machine learning research*, 17(1):1235–1241, 2016.

[162] Jin Miao, Ku He, Andreas Gerstlauer, and Michael Orshansky. Modeling and synthesis of quality-energy optimal approximate adders. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '12, pages 728–735, New York, NY, USA, 2012. ACM.

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

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

[165] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 25–34, May 2010.

[166] Ioannis Mitliagkas, Michael Borokhovich, Alexandros G Dimakis, and Constantine Caramanis. Frogwild! fast pagerank approximations on graph engines. *Proc. VLDB Endow.*, 8(8):874–885, 4 2015.

[167] T. Moreau, A. Sampson, and L. Ceze. Approximate computing: Making mobile systems more efficient. *IEEE Pervasive Computing*, 14(2):9–13, Apr 2015.

[168] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. Snnap: Approximate computing on programmable socs via neural acceleration. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture, HPCA 2015*, pages 603–614. Institute of Electrical and Electronics Engineers Inc., 3 2015.

[169] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. Deep learning for entity matching: A design space exploration. In *SIGMOD*, pages 19–34. Association for Computing Machinery, 2018.

[170] Rene Mueller and Jens Teubner. FPGA: What's in it for a database? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 999–1004. ACM, 2009.

[171] Supun Nakandala and Arun Kumar. Nautilus: An optimized system for deep transfer learning over evolving training datasets. In *Proceedings of the 2022 International Conference on Management of Data*, pages 506–520, 2022.

[172] Supun Nakandala, Yuhao Zhang, and Arun Kumar. Cerebro: A data system for optimized deep learning model selection. *Proceedings of the VLDB Endowment*, 13(12):2159–2173, 2020.

[173] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. Accelerating deep learning workloads through efficient multi-model execution. In *NeurIPS Workshop on Systems for Machine Learning*, volume 20, 2018.

[174] Mark Needham and Amy E Hodler. *Graph Algorithms: Practical Examples in Apache Spark and Neo4j*. O'Reilly Media, 2019.

[175] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

[176] NVIDIA. GPUDirect RDMA. `https://developer.nvidia.com/gpudirect`, 2017.

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

[178] NVIDIA Corp. NVIDIA Jetson Orin NX Series Data Sheet. `https://developer.nvidia.com/downloads/jetson-orin-nx-series-data-sheet`, 4 2022.

[179] Alexander Ocsa. Sql for gpu data frames in rapids accelerating end-to-end data science workflows using gpus. In *LatinX in AI Research at ICML 2019*, 2019.

[180] H. Omar, M. Ahmad, and O. Khan. Graphtuner: An input dependence aware loop perforation scheme for efficient execution of approximated graph algorithms. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 201–208, Nov 2017.

[181] OmniSci Inc. Open Source Analytical Database & SQL Engine. `https://www.omnisci.com/platform/omniscidb`, 2018.

[182] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In *Performance evaluation and benchmarking*, pages 237–252, 2009.

[183] OpenAI. AI and Compute. `https://openai.com/index/ai-and-compute/`, 2020. Accessed: 2024-06-12.

[184] OpenAI. GPT-4 Technical Report, 2023.

[185] OpenAI Inc. Dall·e. `https://openai.com/blog/dall-e-now-available-without-waitlist`, 9 2022.

[186] Muhsen Owaida, Gustavo Alonso, Laura Fogliarini, Anthony Hock-Koon, and Pierre-Etienne Melet. Lowering the latency of data processing pipelines through fpga based hardware acceleration. *Proc. VLDB Endow.*, 13(1):71–85, 2019.

[187] Daniele Jahier Pagliari, Enrico Macii, and Massimo Poncino. Approximate energy-efficient encoding for serial interfaces. *ACM Trans. Des. Autom. Electron. Syst.*, 22(4):64:1–64:25, May 2017.

[188] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. Blocking and filtering techniques for entity resolution: A survey. *ACM Computing Surveys (CSUR)*, 53(2):1–42, 2020.

[189] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*. 2019.

[190] 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.

[191] Johns Paul, Jiong He, and Bingsheng He. GPL: A GPU-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1935–1950, 2016.

[192] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178, 2009.

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

[194] Steven Pelley, Thomas F Wenisch, Brian T Gold, and Bill Bridge. Storage management in the nvram era. *Proceedings of the VLDB Endowment*, 7(2):121–132, 2013.

[195] Colin Perciva. Matching with Mismatches and Assorted Applications. 2006.

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

[197] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.

[198] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad, 2018.

[199] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. MLPerf Inference Benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459, 2020.

[200] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.

[201] Scott Reed, Zeynep Akata, Xinchen Yan, Lajanugen Logeswaran, Bernt Schiele, and Honglak Lee. Generative adversarial text to image synthesis. In *International conference on machine learning*, pages 1060–1069. PMLR, 2016.

[202] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2016.

[203] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, June 2001.

[204] Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. Monitoring and debugging the quality of results in approximate programs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 399–411, New York, NY, USA, 2015. ACM.

[205] Matteo Risso, Alessio Burrello, Giuseppe Maria Sarda, Luca Benini, Enrico Macii, Massimo Poncino, Marian Verhelst, and Daniele Jahier Pagliari. Precision-aware latency and energy balancing on multi-accelerator platforms for dnn inference. *arXiv preprint arXiv:2306.05060*, 2023.

[206] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10684–10695, June 2022.

[207] Arnon Rungsawang and Bundit Manaskasemsak. Fast pagerank computation on a gpu cluster. In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 450–456. IEEE, 2012.

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

[209] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily Denton, Seyed Kamyar Seyed Ghasemipour, Burcu Karagol Ayan, S. Sara Mahdavi, Rapha Gontijo Lopes, Tim Salimans, Jonathan Ho, David J Fleet, and Mohammad Norouzi. Photorealistic text-to-image diffusion models with deep language understanding, 2022.

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

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

[212] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.

[213] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 25–36, New York, NY, USA, 2013. ACM.

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

[215] Kaz Sato, Cliff Young, and David Patterson. An in-depth look at google's first tensor processing unit (TPU). *Google Cloud Big Data and Machine Learning Blog*, 12, 2017.

[216] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[217] 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.

[218] Muhammad Shafique, Waqas Ahmad, Rehan Hafiz, and Jörg Henkel. A low latency generic accuracy configurable adder. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 86:1–86:6, New York, NY, USA, 2015. ACM.

[219] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. A study of the fundamental performance characteristics of gpus and cpus for database analytics. In *SIGMOD*, pages 1617–1632, 2020.

[220] Yashvardhan Sharma and Sahil Gupta. Deep learning approaches for question answering system. *Procedia computer science*, 132:785–794, 2018.

[221] Shaoshuai Shi, Li Jiang, Dengxin Dai, and Bernt Schiele. Motion Transformer with Global Intention Localization and Local Movement Refinement, 2023.

[222] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on gpus: A survey. *ACM Computing Surveys (CSUR)*, 50(6):1–35, 2018.

[223] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.

[224] Patrice Simard, Bernard Victorri, Yann LeCun, and John Denker. Tangent prop - a formalism for specifying selected invariances in an adaptive network. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 895–903. Morgan-Kaufmann, 1992.

[225] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hardware-conscious hash-joins on gpus. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 698–709, 2019.

[226] Evangelia A Sitaridi and Kenneth A Ross. Optimizing select conditions on gpus. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, pages 1–8, 2013.

[227] Arun Subramaniyan and Reetuparna Das. Parallel automata processor. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 600–612, New York, NY, USA, 2017. ACM.

[228] Xin Sui, Andrew Lenharth, Donald S. Fussell, and Keshav Pingali. Proactive control of approximate programs. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 607–621, New York, NY, USA, 2016. ACM.

[229] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`, 2023.

[230] Anthony Thomas and Arun Kumar. A comparative evaluation of systems for scalable linear algebra-based analytics. *Proc. VLDB Endow.*, 11(13):2168–2182, 2018.

[231] 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, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[232] Jonathan Ying Fai Tong, David Nagle, and Rob. A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Trans. Very Large Scale Integr. Syst.*, 8(3):273–285, June 2000.

[233] 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.

[234] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models, 2023.

[235] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[236] Hung-Wei Tseng, Yang Liu, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. Gullfoss: Accelerating and simplifying data movement among heterogeneous computing and storage resources. Technical report, UCSD Technical Report, 2015.

[237] 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.

[238] A. K. Verma, P. Brisk, and P. Ienne. Variable latency speculative addition: A new paradigm for arithmetic circuit design. In *2008 Design, Automation and Test in Europe*, pages 1250–1255, March 2008.

[239] P. Volk, D. Habich, and W. Lehner. GPU-Based speculative query processing for database operations. In *ADMS@VLDB*, 2010.

[240] Patrick von Platen, Suraj Patil, Anton Lozhkov, Pedro Cuenca, Nathan Lambert, Kashif Rasul, Mishig Davaadorj, and Thomas Wolf. Diffusers: State-of-the-art diffusion models. `https://github.com/huggingface/diffusers`, 2022.

[241] C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie. Caltech-ucsd birds-200-2011 (cub-200-2011). Technical Report CNS-TR-2011-001, California Institute of Technology, 2011.

[242] Slawomir Walkowiak, Konrad Wawruch, Marita Nowotka, Lukasz Ligowski, and Witold Rudnicki. Exploring utilisation of gpu for database applications. *Procedia Computer Science*, 1(1):505–513, 2010.

[243] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer, 2014.

[244] Jianguo Wang, Chunbin Lin, Ruining He, Moojin Chae, Yannis Papakonstantinou, and Steven Swanson. Milc: Inverted list compression in memory. *Proc. VLDB Endow.*, 10(8), 4 2017.

[245] Jianguo Wang, Dongchul Park, Yannis Papakonstantinou, and Steven Swanson. SSD in-storage computing for search engines. *IEEE Transactions on Computers*, 2016.

[246] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. Concurrent analytical query processing with gpus. *VLDB*, 7(11):1011–1022, 7 2014.

[247] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. A systematic methodology for analysis of deep learning hardware and software platforms. In *The 3rd Conference on Machine Learning and Systems (MLSys)*, 2020.

[248] Zeke Wang, Huiyan Cheah, Johns Paul, Bingsheng He, and Wei Zhang. Accelerating database query processing on opencl-based fpgas. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 274–274. ACM, 2016.

[249] Dingzhu Wen, Peixi Liu, Guangxu Zhu, Yuanming Shi, Jie Xu, Yonina C Eldar, and Shuguang Cui. Task-oriented sensing, computation, and communication integration for multi-device edge ai. *arXiv preprint arXiv:2207.00969*, 2022.

[250] Zeyi Wen, Jiashuai Shi, Bingsheng He, Qinbin Li, and Jian Chen. ThunderSVM: A fast SVM library on GPUs and CPUs. *To appear in arxiv*, 2018.

[251] Zeyi Wen, Jiashuai Shi, Bingsheng He, Qinbin Li, and Jian Chen. ThunderGBM: Fast GBDTs and random forests on GPUs. *To appear in arXiv*, 2019.

[252] 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.

[253] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. Machine Learning at Facebook: Understanding Inference at the Edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344, 2019.

[254] Fangzhao Wu, Ying Qiao, Jiun-Hung Chen, Chuhan Wu, Tao Qi, Jianxun Lian, Danyang Liu, Xing Xie, Jianfeng Gao, Winnie Wu, et al. Mind: A large-scale dataset for news recommendation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 3597–3606, 2020.

[255] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *MICRO*, pages 107–118. IEEE Computer Society, 2012.

[256] Haicheng Wu, D. Zinn, M. Aref, and S. Yalamanchili. Multipredicate join algorithms for accelerating relational graph processing on gpus. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 09 2014.

[257] Tianji Wu, Bo Wang, Yi Shan, Feng Yan, Yu Wang, and Ningyi Xu. Efficient PageRank and SpMV computation on AMD GPUs. In *2010 39th International Conference on Parallel Processing*, pages 81–89. IEEE, 2010.

[258] Yann Collet. Zstandard - Fast real-time compression algorithm. `https://github.com/facebook/zstd/releases/`, 2018.

[259] 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.

[260] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. Neural acceleration for gpu throughput processors. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 482–493, New York, NY, USA, 2015. ACM.

[261] Mao Ye, Peifeng Yin, and Wang-Chien Lee. Location Recommendation for Location-Based Social Networks. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, page 458–461. Association for Computing Machinery, 2010.

[262] Rong Ye, Ting Wang, Feng Yuan, Rakesh Kumar, and Qiang Xu. On reconfiguration-oriented approximate adder design and its application. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, pages 48–54, Piscataway, NJ, USA, 2013. IEEE Press.

[263] Thomas Y. Yeh, Glenn Reinman, Sanjay J. Patel, and Petros Faloutsos. Fool me twice: Exploring and exploiting error tolerance in physics-based animation. *ACM Trans. Graph.*, 29(1):5:1–5:11, December 2009.

[264] Yong Ho Song. The OpenSSD Project. `http://www.openssd-project.org/wiki/The_OpenSSD_Project`, 2017.

[265] Chien-Chih Yu and Hsiao-ping Chang. Personalized location-based recommendation services for tour planning in mobile tourism applications. In *E-Commerce and Web Technologies: 10th International Conference, EC-Web 2009, Linz, Austria, September 1-4, 2009. Proceedings 10*, pages 38–49. Springer, 2009.

[266] Y. Yuan, M. F. Salmi, Y. Huai, K. Wang, R. Lee, and X. Zhang. Spark-GPU: An accelerated in-memory data processing engine on clusters. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 273–283, 2016.

[267] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The Yin and Yang of processing data warehousing queries on GPU devices. *VLDB*, 6(10):817–828, 2013.

[268] Orestis Zachariadis, Nitin Satpute, Juan Góumez-Luna, and Joaquqín Olivares. Accelerating sparse matrix-matrix multiplication with GPU Tensor Cores. *Computers and Electrical Engineering*, 88:106848, 2020.

[269] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Transactions on Networking*, 29(2):595–608, 2020.

[270] Jie Zhang, David Donofrio, John Shalf, Mahmut T Kandemir, and Myoungsoo Jung. NVMMU: A non-volatile memory management unit for heterogeneous gpu-ssd architectures. In *PACT*, pages 13–24. IEEE, 2015.

[271] 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.

[272] Kai Zhang, Feng Chen, Xiaoning Ding, Yin Huai, Rubao Lee, Tian Luo, Kaibo Wang, Yuan Yuan, and Xiaodong Zhang. Hetero-DB: Next generation high-performance database systems by best utilizing heterogeneous computing and storage resources. *Journal of Computer Science and Technology*, 30(4):657–678, 2015.

[273] Wei Zhang, Wei Wei, Lingjie Xu, Lingling Jin, and Cheng Li. Ai matrix: A deep learning benchmark for alibaba data centers. *arXiv preprint arXiv:1909.10562*, 2019.

[274] Zhejun Zhang, Alexander Liniger, Christos Sakaridis, Fisher Yu, and Luc Van Gool. Real-time motion prediction via heterogeneous polyline transformer with relative pose encoding. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[275] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

[276] Li Zhou, Mohammad Hossein Samavatian, Anys Bacha, Saikat Majumdar, and Radu Teodorescu. Adaptive parallel execution of deep neural networks on heterogeneous edge devices. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, SEC '19, page 195–208, New York, NY, USA, 2019. Association for Computing Machinery.

[277] Zikang Zhou, Luyao Ye, Jianping Wang, Kui Wu, and Kejie Lu. Hivt: Hierarchical vector transformer for multi-agent motion prediction. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8813–8823, 2022.

[278] Ning Zhu, W. L. Goh, and K. S. Yeo. An enhanced low-power high-speed adder for error-tolerant application. In *Proceedings of the 2009 12th International Symposium on Integrated Circuits*, pages 69–72, Dec 2009.

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