dpKernels: Harvesting DPU Compute Resources for Data-path Efficiency in Cloud Data Processing

Jiasheng Hu
University of Toronto
Toronto, Ontario, Canada
jasonhu@cs.toronto.edu

Kaiwen Zheng
University of Toronto
Toronto, Ontario, Canada
helloworld.zheng@mail.utoronto.ca

Anna Li
University of Toronto
Toronto, Ontario, Canada
annaz.li@mail.utoronto.ca

Sidharth Sankhe
University of Toronto
Toronto, Ontario, Canada
sankhe@cs.toronto.edu

Philip A. Bernstein
Microsoft Research
Redmond, Washington, USA
philbe@microsoft.com

Qizhen Zhang
University of Toronto
Toronto, Ontario, Canada
qz@cs.toronto.edu

ABSTRACT

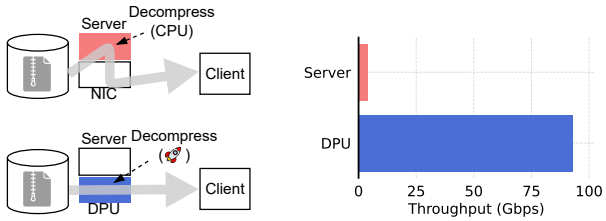
Data processing units, or DPUs, are equipped with hardware accelerators for compute-intensive data path tasks. Although DPUs' SoC cores are wimpier than the host's, hardware accelerators are typically orders of magnitude faster than CPUs. Harvesting DPU hardware accelerators for database systems could significantly increase throughput and save host CPU cycles. However, due to the heterogeneity of DPUs' hardware configurations and performance characteristics, it is challenging to offer a unified and portable solution for cloud data processing systems to harvest the compute resources on DPUs across generations and vendors. Additionally, due to DPU resource constraints, offloaded compute tasks need to be carefully optimized and scheduled to achieve high efficiency and avoid performance regression. To address these challenges, we introduce two levels of abstraction: dpKernels, which are unified, efficient, and portable primitives that abstract DPU compute resources (i.e., hardware accelerators and SoC cores) for cloud data systems, and dpManager, an onboard management framework that abstracts specific DPU platforms for dpKernels to deliver their promises with optimized, scheduled, and cross-platform executions. The benefits of our proposal have been validated by various workloads, systems, and DPU hardware.

PVLDB Reference Format:
Jiasheng Hu, Kaiwen Zheng, Anna Li, Sidharth Sankhe, Philip A. Bernstein, and Qizhen Zhang. dpKernels: Harvesting DPU Compute Resources. PVLDB, 19(10): 2563 - 2576, 2026.
doi:10.14778/3828612.3828615

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/fardatalab/dpKernels.

1 INTRODUCTION

Due to hardware resource disaggregation [15, 22, 71, 74, 77, 80–82], cloud data-intensive systems, such as databases and KV services, have been progressively modularized—the components in these systems are distributed in separate locations and are loosely connected by high-speed networks [69]. On one hand, higher degrees of distribution and disaggregation facilitate scalability, elasticity, and availability, which are crucial properties of cloud data systems. On the other hand, the critical execution paths where data moves have been extended from local buses (that connect storage, memory, and compute) to the data center network, and thus cloud data processing is increasingly bottlenecked by network communication [67, 83].



(a) Data paths (server vs. DPU) **(b) Throughput comparison**

Figure 1: Data path efficiency with servers vs. DPUs.

Data-path efficiency over the network can be characterized by the speed of the network and the execution performance of compute tasks in the data path. The speed of the network (i.e., latency and bandwidth) determines how fast data is moved from the source to the destination given a fixed volume. Data center network fabrics have drastically improved over the years (e.g., advanced network topology [2, 33] and transport protocols [18, 26, 40]), currently providing 100s Gbps point-to-point throughput [56]. Placing more compute tasks on the data path is becoming a common design paradigm for two purposes: (1) certain computation can reduce data volumes moved across the network, e.g., compute pushdown to the storage backend for near data processing [20, 42, 75, 76, 78], and (2) specific tasks are required by scale-out system design, e.g., data encryption for secure data transfers [10, 62] and decompression for serving reads from compressed storage [1, 36]. Compared to network speed, the performance of executing compute-intensive tasks is increasingly the bottleneck of data-path efficiency due to the decline in the scaling of general-purpose computing hardware [66].

Figure 1a (top) shows an example of the data path for data serving with DEFLATE-compressed blocks on a traditional server (experimental setup is detailed in Section 8): blocks are decompressed on the server before identifying the requested data and sending

Proceedings of the VLDB Endowment, Vol. 19, No. 10 ISSN 2150-8097.
doi:10.14778/3828612.3828615

it back to the client. Although the network interface card (NIC) provides 100 Gbps bandwidth, single-block decompression speed on the server becomes the bottleneck—only 4.1 Gbps even when parallelization is enabled with multi-threading and SIMD.

In response, new hardware solutions have been proposed to accelerate compute-intensive tasks on the data path, such as GPUs [12, 38, 60], FPGAs [16], and specialized instructions [21, 30, 39, 73]. For example, HippogriffDB [38] stores data in a compressed format and uses GPUs to decompress it before running OLAP queries, thus improving effective I/O bandwidth. GOLAP [12] also places a GPU in the data path of OLAP query execution and uses it to decompress and prune data at the line rate. Chiosa et al. [16] proposed an FPGA architecture that compresses and decrypts data on the I/O path of SAP HANA. Parquet-Select [39] and several other recent proposals [21, 30, 73] exploit BMI or SIMD instructions to parallelize the scan and pruning operations over column databases. Although these approaches have improved the performance of the target data-path tasks, they suffer from high cost (e.g., GPU solutions), programming complexity (e.g., FPGA solutions), and generalization to other data-path tasks (all these solutions).

In this paper, we propose to optimize the data-path efficiency over the data center network using Data Processing Units (DPUs), a family of data center programmable NICs (SmartNICs) designed to optimize I/O activities over the network. Compared to aforementioned hardware, DPUs are generally programmable with their SoC (System-on-a-Chip) cores, target low-cost high-efficiency data-path optimizations, are currently in mass production [6, 29, 32, 45, 52], and have been deployed in data centers to replace traditional NICs [3, 4, 13, 23, 55]. DPUs are equipped with a variety of hardware accelerators that achieve exceedingly high performance and energy efficiency for compute-intensive operations on the network or storage data paths. For example, NVIDIA BlueField-3 (BF-3) [53] has a decompression accelerator. We can directly stream data from SSDs into the accelerator (Figure 1a) and achieve line-rate decompression at 92.9 Gbps (Figure 1b).

Challenge 1: Low-level and vendor-specific access methods to accelerators. Vendor-provided SDKs for programming DPUs are low-level. Constructing an application workflow to utilize specific accelerators requires significant engineering effort. For example, to use the decompression accelerator on BF-3, one needs to use NVIDIA's Datacenter-On-a-Chip (DOCA) SDK [54] to first create a DOCA device that represents the decompression accelerator, a DOCA task that specifies the decompression operation, two buffers from DOCA buffer inventory as the input (compressed data) and output (decompressed data) of the task, and finally a DOCA context to reference the task. The task can then be submitted to the device and polled in DOCA's progress engine with the context. This tedious process is repeated when implementing another task, the SDK evolves, or switching to a different DPU. Data Plane Development Kit (DPDK) [65] supported by some DPUs is vendor-agnostic, but it targets packet processing and remains low-level to data systems.

Challenge 2: Diversity of hardware across generations and vendors. Detailed configuration and availability of hardware accelerators varies between DPUs from different vendors. Specifically, BF-3 has accelerators for decompression, RegEx matching, and AES encryption. In comparison, AMD Pensando [6] equips a different set of accelerators for compression, decompression, and encryption. In fact, every DPU has a unique configuration of hardware for its target applications, e.g., Intel IPU [29], Marvell OCTEON [45], and Amazon Nitro [4]. Even with the same vendor, DPUs across generations can have different accelerators. For instance, compared to BF-2, BF-3 removes the compression accelerator but adds new accelerators for erasure coding. This diversity makes it impossible to directly migrate the code from one DPU to another.

Challenge 3: Inefficient data flows on DPUs. Although hardware accelerators can achieve high compute efficiency, general resources such as CPU cores and memory on DPUs are wimpier than their counterparts on servers [43, 79]. This implies that if not careful, executing a workflow that moves data between the original data path and the hardware accelerators can become the roadblock to high efficiency. We will show how the performance of existing data structures deteriorates on DPUs.

To overcome Challenge 1, we develop dpKernels, the first level of abstraction on DPUs that provides a unified and declarative interface for data systems to offload compute-intensive tasks to hardware accelerators *by simply declaring the functionality and providing arguments*. Each dpKernel maps a specific functionality to the corresponding hardware accelerator on the DPU. For instance, to decompress a data block with the decompression accelerator, one can invoke `dpk_decompress` and specify the addresses of the input and output buffers—hardware-specific and vendor-specific complexities are concealed from data system developers. dpKernels are registered to and managed by dpManager, which prepares the execution environment for dpKernels, accepts invocation requests from applications, schedules execution on proper hardware, and notifies the callers when kernels [1] are completed.

To port dpKernels across different DPUs and overcome Challenge 2, dpManager introduces the second level of abstraction that exposes the support required by kernel execution to a DPU platform. Service providers supply to dpManager a set of functions that entail the hardware characteristics and SDK intricacy of the target DPU, such as the initialization of the appropriate hardware accelerator for each dpKernel and allocation of memory that is directly accessible by accelerators. To cope with varying availability of specific hardware accelerators, a backup CPU implementation is provided for each dpKernel. Therefore, if the corresponding hardware accelerator is not available on the target DPU, each dpKernel can still be automatically executed using the SoC cores. Finally, dpManager maintains a catalog that stores the characteristics and configuration of the target DPU to adapt kernel execution and optimizations. These two levels of abstraction enable ease of programming and "offload once, run anywhere" portability for executing compute-intensive tasks in cloud data systems on DPUs.

To optimize data flows and fully exploit hardware performance when executing dpKernels on DPUs (Challenge 3), we develop:

- A memory management module that allows applications to allocate *zero-copy* buffers from dpManager. These buffers are directly accessible by hardware accelerators.
- A fast lockless ring buffer tailored for DPUs that eliminates any inefficiency in exchanging messages between applications and dpManager.

[1] We interchangeably use kernels and dpKernels in this paper.

- Kernel execution optimizations that pipeline, coalesce, and parallelize dpKernels. These optimizations are selectively applied based on the nature of the kernels.
- Dynamic scheduling that automatically balances requests between the hardware accelerators and SoC cores (when enabled) to prevent head-of-line blocking for small requests.

In summary, this paper presents dpKernels and dpManager, two levels of DPU abstraction that enable easy and portable offloading of compute-intensive tasks in cloud data systems to improve data-path efficiency. It includes techniques we developed to optimize the performance of on-DPU data flows and kernel execution. We implemented a prototype of the system and ported it to real DPUs. The benefits of adopting dpKernels in cloud data processing have been extensively evaluated with an analytical database service and an in-memory key-value caching service.

2 BACKGROUND AND MOTIVATION

Data processing units (DPUs) represent a family of recent SmartNICs that incorporate System-on-Chip (SoC) cores and accelerators for flexible and efficient network offloading. Generally, the hardware features of DPUs are summarized as follows (Figure 2):

- A CPU in energy-efficient architectures, e.g., Arm or MIPS.
- A modest amount of onboard DRAM memory.
- A set of hardware accelerators for compute-heavy tasks.
- A high-speed network interface connecting to the network.
- A PCIe switch connecting to the host and other devices.

These resources are customized to optimize the performance of data movement over the network: the NIC can provide hundreds of gigabits/s network bandwidth, the PCIe switch enables direct data transfers between the DPU, the host, and other PCIe devices such as SSDs, the hardware accelerators can execute compute-intensive tasks at the line rate (as shown in Figure 1), and the CPU and the memory allow for implementing applications that can benefit from customized network-path optimizations. Major hardware vendors have been offering DPU devices at mass production, e.g., AMD [6], Broadcom [13], Intel [29], Kalray [32], and NVIDIA [52], to name a few. Given the hardware availability, there has been increasing popularity in rethinking the design of various distributed system components in data centers with DPUs [27, 34, 41, 44, 79, 84].

2.1 Compute Resources on DPUs

A DPU has two types of resources that can be utilized to improve the data-path efficiency of cloud data processing systems. First, the set of hardware accelerators target specific tasks, such as data compression and decompression, erasure coding recovery, and Regular Expression (RegEx) matching, which are commonly executed in the data paths of data center applications. These accelerators are exceedingly high-performance. In addition to decompression in Figure 1, we further show the performance of other hardware accelerators on BF-3 compared to the server CPU in Figure 3a. As observed, dedicated hardware accelerators can outperform the software implementation on the host server by orders of magnitude. For example, erasure coding (EC), which can be leveraged for fault tolerance [59], is computationally expensive on the server: 2.6 Gbps, 35× slower than the EC engine on BF-3. The gap is not as large for RegEx matching, but the DPU's accelerator is still significantly faster (3.5× higher throughput). However, in exchange for the high performance, hardware accelerators are not general, i.e., they cannot execute compute tasks that are not their targets.

In contrast, DPUs' SoC cores are generally programmable but are less performant. To show the performance (and the trend) of the CPU cores on DPUs, we run the DEFLATE decompression on the SoC cores on BF-2 and BF-3, two generations of NVIDIA DPUs, and Pensando Elba, a DPU from AMD. The specifications of these cores are shown in Table 1. Our results (Figure 3b) show that newer generation of DPUs are closing the single-core performance gap to the server CPU; the core count is also growing. As the performance of energy-efficient CPU architectures advances, offloading compute tasks from server CPUs to DPU SoC cores becomes more feasible.

2.2 Utilization Challenges

Programming challenge. To utilize DPU compute resources, especially hardware accelerators, one uses vendor-provided software development kits (SDKs). Table 1 shows SDKs for three different DPUs from two vendors, which provide low-level mechanisms for applications to access the acceleration engines on DPUs. For example, in DOCA [54] 1.5.0, the following process is required to utilize the compression accelerator to execute a compression task.

(1) Initialization: discover the compression accelerator, open it as a DOCA device, and initialize its work queue.
(2) Buffer preparation: allocate DPU memory for input (original data) and output (compressed data) buffers, convert them into DOCA buffers with DOCA buffer inventory, and set appropriate lengths for the buffers.
(3) Task construction: create a DOCA *Compress Job* object, and populate it with proper parameters and the buffers.
(4) Task submission: invoke the work queue submission API to submit the compress job.
(5) Completion polling: continuously invoke the work queue progress retrieval API to poll the completion of the job.
(6) Cleanup: release all the DOCA buffers and objects and DPU memory allocated in the process.

The onerous implementation also obsoletes easily: in DOCA 2.2.1—the first version that supported BF-3, the API for work queue initialization changed to support buffer extensions, and in DOCA 3.1.0–the latest version as of this writing, the API for task construction, submission, and polling completely evolved. The libraries in the Software-in-Silicon Development Kit (SSDK) [5] for accessing accelerators on AMD Pensando DPUs share similar characteristics. Hence, the first challenge of utilizing DPU compute resources for data processing systems is low-level, vendor-specific SDKs for accessing hardware accelerators.

Portability challenge. Different DPUs are equipped with different sets of hardware accelerators, even for DPUs of the same vendor. Table 1 shows the hardware accelerator availability on BF-2 (C D E R), BF-3 (D E R), and Elba (C D E), where C stands for compression, D for decompression, E for encryption, and R for RegEx matching. These availability variations pose another portability issue: the compute task accelerated on one DPU may fail to execute on another because the required accelerator is not available. For example, the compression task implementation for BF-2 is not runnable on BF-3. Considering the many DPU offerings [6, 13, 29, 32, 52], the

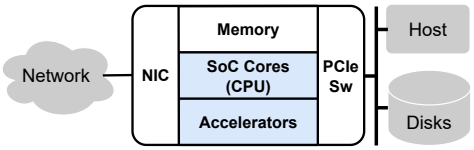
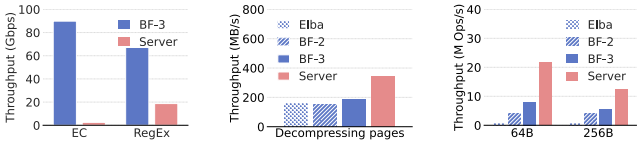


Figure 2: DPU architecture.



(a) Erasure Coding and RegEx. **(b) SoC core for decompression.** **(c) Messaging via a queue.**

Figure 3: Performance of performing compute tasks.

second challenge of utilizing DPU compute resources is the diverse configurations of hardware accelerators on different DPUs.

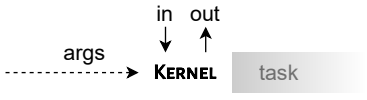
Performance challenge. Utilizing DPU compute resources for data-path processing means data needs to be moved between system components on a DPU, for example, from the NIC to the memory and then to the target hardware accelerator. However, the performance of data movement highly depends on that of memory operations, which are generally less efficient on DPUs. To show the challenge, we run an experiment where two processes pass request and response messages via a fast concurrent queue [24] on each of the DPUs in Table 1 and compare the performance with that on the server. Figure 3c reports the results. We observe that all DPUs are significantly slower than the server—the gap is up to an order of magnitude. This experiment shows that data movement on a DPU, if not carefully managed, can become the roadblock to harvesting the efficiency of DPU compute resources for data-path processing, highlighting the importance of optimizing data movement.

3 SYSTEM OVERVIEW

We propose a DPU computing framework that allows cloud data processing systems to easily utilize onboard compute resources to improve their data-path efficiency. To address the three challenges of utilizing compute resources on DPUs (§2), we adopt a holistic design centering around the interface and execution strategies.

3.1 dpKernels—First Level of Abstraction

To tackle the programming challenge, we abstract compute tasks on data paths with dpKernels. A dpKernel is a basic compute primitive that processes input data and generates output data with *fixed* logic as illustrated below.



This general abstraction applies to any compute task using arbitrary compute resources on DPUs. For example, a data compression task takes as input a block of uncompressed data and utilizes the compression accelerator to generate a block of compressed data as output. Although the processing logic is predetermined, some kernels are parameterized (e.g., the compression level in a compression kernel), an invocation request can instantiate these parameters as arguments in addition to the input and output data.

Table 1: Compute resources and access methods vary across DPUs.

DPU	SoC Cores	Accelerators	SDK
NVIDIA BlueField-2	Arm A72 ×8 @2.5 GHz	C, D, E, R	DOCA 1.5.0–3.1.0
NVIDIA BlueField-3	Arm A78 ×16 @3.0 GHz	D, E, R	DOCA 2.2.1–3.1.0
AMD Pensando Elba	Arm A72 ×16 @3.0 GHz	C, D, E	SSDK

For applications, this abstraction is similar to database stored procedures [46]. In fact, it is inspired by the increasing popularity of adopting stored procedures as the interface between data-intensive applications and heterogeneous computing platforms in recent years [31, 64, 78]. However, three notable differences distinguish dpKernels from regular stored procedures. First, rather than target a specific database, dpKernels are designed for (any) data paths. Hence, they are schemaless—the input and the output have no assumed structures, and both are explicitly provided when invoking a kernel. Second, dpKernels are designed to utilize DPU compute resources, especially the hardware accelerators, which harden the processing logic for specific tasks and are not reconfigurable (unlike programmable hardware such as CPUs or FPGAs). Hence, they are implemented by the service providers, not by the users. Applications simply invoke the provided kernels. Finally, they execute on DPUs where general resources (i.e., CPU and memory) are less performant than servers, to achieve performance higher than servers. Hence, their executions demand careful orchestration and optimization.

The key benefit enabled by dpKernels is a high-level, vendor-agnostic programming interface to harvest DPU compute resources. To invoke dpKernels, an application calls the following function.

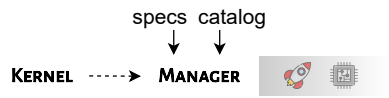
```
ctx ← invoke(dpk_name,in,out,args)
```

To facilitate pipelined task execution, invocations are asynchronous. Using the returned context object, the application can query if a previously-invoked task has finished as follows.

```
status ← query(ctx)
```

3.2 dpManager—Second Level of Abstraction

Upon applications' invocations, kernels are executed by the framework using proper DPU compute resources. As discussed (§2), a sizable number of DPU brands are offered by different vendors and are evolving at a steady pace. **To tackle the portability challenge**, we introduce dpManager, a standalone DPU service that manages dpKernels and abstracts DPU compute hardware as below.



Specifically, dpManager employs two core components: *Platform Specs*, which is a set of provider-defined functions that prepare the DPU environment (e.g., hardware accelerators, cores, and memory) and map dpKernels to implementations with DPU-specific SDKs, and *Catalog*, which stores profiles incorporating hardware characteristics, especially those of accelerators. The former instructs how to compile and execute kernels on the target DPU, while the latter adapts execution strategies and optimizations based on the available hardware. In particular, dpManager requires a backup

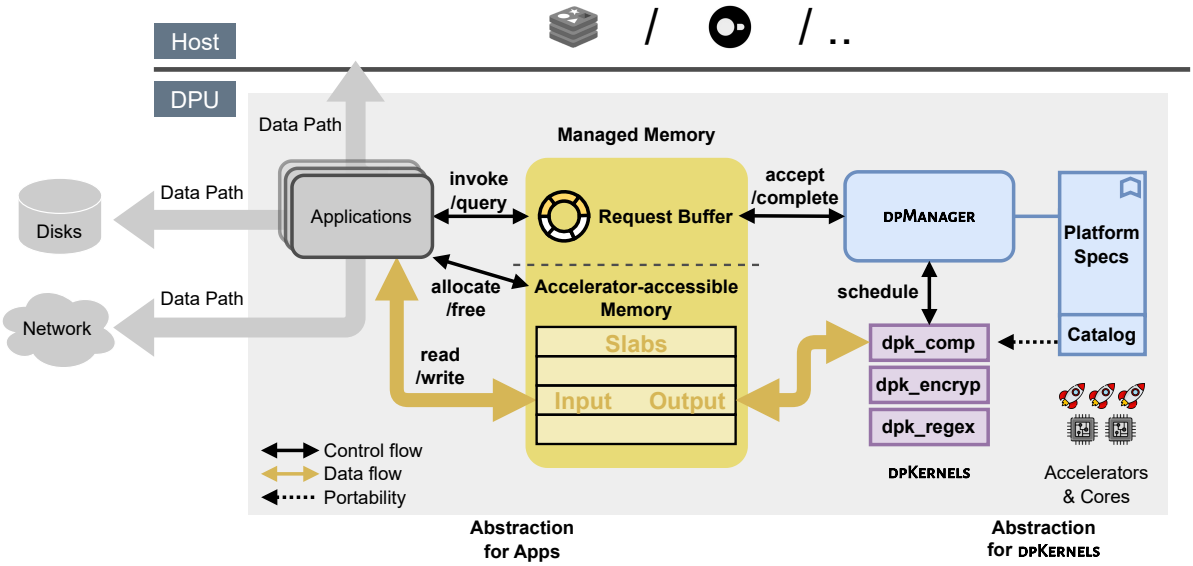


Figure 4: System architecture with DPKERNELS and DPMANAGER.

software implementation for each DPKERNEL to accommodate varying availability of hardware accelerators. It enables or disables the hardware-accelerated kernel execution depending on whether the corresponding hardware accelerators are available and can always run the backup implementation on DPU SoC cores.

With DPMANAGER, a DPKERNEL can execute on any DPU platform, harvesting the native performance of the target DPU platform and achieving the possibility of "offload once, run anywhere." More detailed portability discussion is deferred to Section 5.

3.3 Architecture

The overall DPU architecture with DPKERNELS and DPMANAGER is shown in Figure 4. The server where cloud data processing systems (e.g., database systems and key-value stores) run is equipped with a DPU, which sits at a center point of data paths where it can access the network via high-speed NIC, disk storage via PCIe to SSDs or the NIC to disaggregated storage, as well as host memory via DMA. DPMANAGER, as a standalone DPU service, precompiles DPKERNELS and loads the executables for invocations and sets up the compute resources utilized by the kernels, both with Platform Specs.

Applications running on the DPU offload compute tasks on their data paths with the following workflow. The application first calls the `invoke` function, which generates kernel invocation requests. The requests are accepted by DPMANAGER, which then schedules the execution of these requests on the prepared compute resources. DPMANAGER continuously monitors the status of the execution and completes the requests once their kernel execution is finished by the compute resources, at which point the output of the kernels is generated. An application is notified of the completions of previously-invoked kernels when it calls the `query` function. Then, it forwards the output to the next hop in the data path at will.

Four activities in the workflow are crucial to its speed: (1) forwarding the input data from the application's data path to the compute resources, (2) passing invocation requests from application to DPMANAGER, (3) executing DPKERNELS on individual compute resources, and (4) scheduling kernels between compute resources.

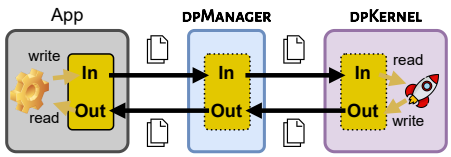
Making these activities efficient is challenging: the first two are bottlenecked by the data movement on DPUs, and the other two depend highly on varying hardware characteristics. **To tackle the performance challenge,** our framework introduces a zero-copy memory management module for applications and compute resources to access input and output, a lockless ring buffer tuned for DPU environments for submitting invocation requests, effective optimizations for kernel execution, and a dynamic scheduler for balancing the load between different hardware. Hardware characteristics are profiled in Catalog.

Key contributions. To summarize, the main contributions our DPU framework makes are (1) the application-facing and platform-facing API design based on the two levels of abstraction and (2) the execution engine based on the above data structures and optimization techniques. The former is the key to achieving stability of the API for data processing systems and portability for kernel implementations and execution, and the latter enables the kernels to harvest the hardware efficiency of DPU compute resources. The two complementary contributions are detailed in the next sections.

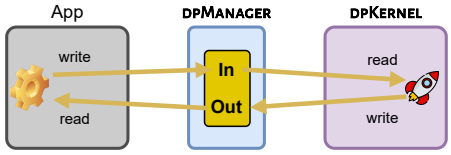
4 HARVESTING EFFICIENCY

4.1 Zero-copy Memory Management

Data copy overhead on DPUs. To forward data in data systems' original data paths for kernel execution, a basic approach would be moving the data across process boundaries, using e.g., inter-process communication (IPC) techniques, such as Unix domain sockets, message queues, which can potentially copy input and output data twice: from the application to DPMANAGER, which further forwards it to a buffer that is consumable by the DPU's hardware accelerator, and vice versa, as shown in Figure 5a. To demonstrate the overhead of such copies, we run a no-op kernel (`dpk_nop`) that takes as input a block of data with a configurable size on BF-3. For comparison, we repeat the same experiment on the server, where memory operations are more efficient. Figure 6a shows the overhead of copying the input data of different sizes:

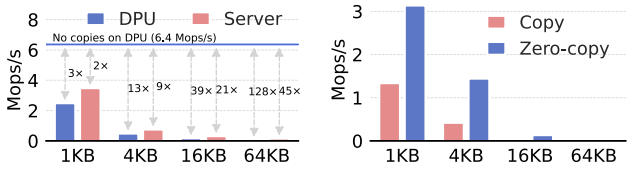


(a) Data copies



(b) Zero-copy data sharing

Figure 5: Data movement for kernel execution.



(a) dpk_nop (b) dpk_decompress

Figure 6: Overhead of memory copies in kernel execution.

with 1 KB input size, the throughput of dpk_nop invocations is slightly above 2 million operations per second (ops/s). With 4 KB, it drops to 500 thousand ops/s. This speed translates to 15 Gbps, which is significantly lower than the hardware speed (Figures 1b and 3a). The overhead of data copies increases with the input size—the throughput is 50 thousand ops/s with 64 KB input. Memory operations are more efficient on the server: the throughput is 2.8× higher than on the DPU with 64 KB input, achieving 72 Gbps. This experiment shows the need to avoid data copies on DPUs.

We further run dpk_decompress, a DPKERNEL that utilizes BF-3's decompression engine. As shown in Figure 6b, the overhead of memory copies is also highly pronounced. With 1 KB input dpk_decompress invocations can only achieve 1.3 million ops/s. The throughput further drops to 400, 33, and 2 thousand ops/s, with 4 KB, 16 KB, and 64 KB data, respectively.

Enabling zero-copy data sharing. To address the challenge of inter-process data copies, DPMANAGER adopts a memory management module (DPMM) that achieves zero-copy data sharing. Specifically, DPMM owns and manages a region of DPU memory. To applications, DPMM provides memory allocation (`allocate`) and deallocation (`free`) mechanisms for applications to create and release buffers in the region for holding kernels' input and output. As shown in Figure 5b, an application can directly write its data to the allocated input buffer. Whether a kernel runs on a hardware accelerator or an SoC core, it can directly read data from the input buffer and write processed data to the output buffer.

Novelty. Essentially, DPMM is a shared cache/memory manager. Differentiating it from generic shared memory management are:

- Accessibility to hardware accelerators. Regular DPU memory cannot be directly accessed by hardware accelerators. To achieve zero-copy data sharing between applications and DPKERNEL execution, DPMANAGER *pre-registers* the memory region it manages to the DPU driver such that any location in the region can be accessed by hardware accelerators.
- Isolation without compromising efficiency. To share memory between DPMANAGER and applications, which have separate address spaces, we provide *unified pointers*. Specifically, leveraging the fact that offsets within the shared memory region are invariant across processes, we use the offsets as the shared memory addresses and transparently convert them into process-local raw pointers. Kernels as well as applications can therefore use the converted raw pointers to directly access the same physical memory. To ensure isolation across processes, DPMM maps the allocated memory pages into application processes' own virtual memory address space. It also gracefully handles illegal (i.e., out-of-bound) pointers, by performing a bounds check before conversion from unified to process-local pointer (and returning an error code when such cases arise).

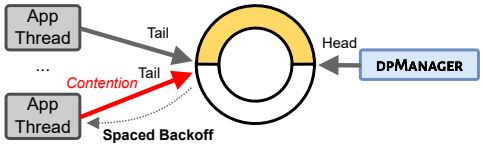


Figure 7: Request ring with minimal contention.

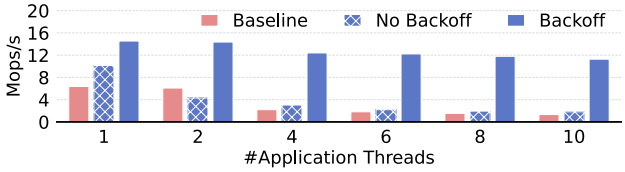


Figure 8: Performance of message exchange approaches.

Effect. Figure 6 shows the benefits of zero-copy data sharing. For dpk_nop, removing data copies increases the invocation throughput on BF-3 to 6.4 million ops/s, up to 128× higher than that with data copies. For dpk_decompress, zero-copy data sharing brings a range of speedups from 2.6× up to 13.5×.

4.2 Fast Message Exchange

When an application calls `invoke`, a request is generated encompassing the following information for a kernel execution: the name of the kernel, pointers to input and output buffers, kernel-specific arguments, and a context that can be checked for completion.

Message exchange overhead on DPUs. Exchanging these requests from applications to DPMANAGER, which accepts them and schedules execution, can become the performance bottleneck. We first adopted a popular concurrent queue [24] designed for servers. In addition to the performance degradation reported in Figure 3c when exchanging requests from a single application thread to DPMANAGER, the data structure also suffers from contention overhead. Figure 8 shows the performance of the baseline concurrent queue under different numbers of application threads calling `invoke` concurrently on BF-3. We observe that when more than two concurrent application threads submit requests, the performance of the queue drops below 2 million ops/s. With this data structure, *the speed of submitting requests to* DPMANAGER *does not keep up with that of executing kernels on the compute resources.*

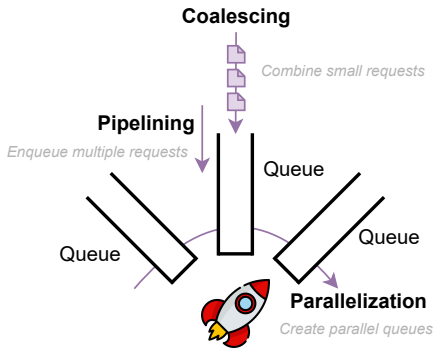


Figure 9: Kernel execution model and optimizations.

Lockless ring buffers. DPMANAGER designates a small part (dozens of MB) of its managed memory for request buffers (one for each application). Each buffer is organized as a multi-producer (application threads) and single-consumer (DPMANAGER) ring with two pointers: a tail that synchronizes application threads to insert invocation requests and a head DPMANAGER uses to consume the requests, as shown in Figure 7. Contention occurs between application threads: when an `invoke` call tries to insert a request to the buffer, it needs to acquire exclusive access to the tail and updates and releases it when the insertion completes.

To minimize contention overhead, we use `compare-and-swap` to implement the acquisition. Specifically, only one thread can succeed in an atomic operation to access the tail. Other competing threads will fail and *return*. This design avoids using slow and CPU-expensive lock primitives (i.e., lockless [51, 58, 63, 70, 79]) and can thus significantly reduce the overhead of contention.

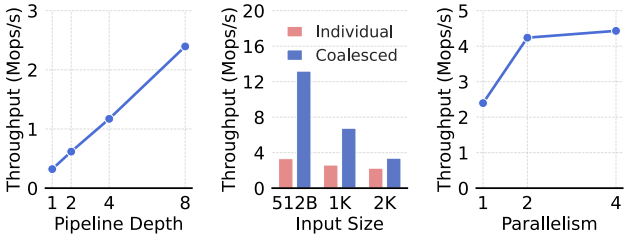
Novelty. To proactively reduce contention, we introduce *spaced backoff*: instead of busy retrying, each failed access to the tail leads to an exponentially growing duration of waiting. However, if threads yield at the same time for the same duration, it defeats the purpose. We space the yielding duration between different threads. Specifically, each failed insertion yields for `backoff_increment` time, up to `backoff_max` before wrapping around. The two backoff parameters are configured with power-of-two values.

Effect. As Figure 8 shows, our request ring buffer consistently maintains message exchange throughput above 10 million ops/s between applications and DPMANAGER, which is 8.5× higher than the baseline when 10 application threads `invoke` kernels in parallel. The spaced backoff strategy makes the most significant contribution, without which the ring is only 1.4× faster.

4.3 Optimized Kernel Execution

Kernel execution model. Hardware accelerators on DPUs are often presented as asynchronous devices with associated work queues, e.g., DOCA on BlueField, SSDK on Pensando, and DPDK on IPU. Therefore, the execution of DPKERNELS is modeled as such: DPMANAGER creates queues for each compute resource and when kernel invocation requests are received and processed, submits work items to corresponding queues. DPMANAGER exploits three optimizations in the execution model as follows (Figure 9).

#1 Pipelining. Due to asynchrony, keeping a single kernel invocation request in flight in a queue generally underutilizes the compute hardware. As shown in Figure 10a, executing `dpk_decompress` with a pipeline depth of 1 only achieves 0.3 million ops/s for requests with 1 KB input using the decompression accelerator on BF-3. Increasing the pipeline depth can increase the throughput significantly: with a depth of 8, the execution throughput reaches 2.4 million ops/s. This optimization can be enabled without special requirements on the kernel or the underlying hardware.



(a) `dpk_decompress` **(b)** `dpk_regex` **(c)** `dpk_decompress`

Figure 10: Effect of kernel execution optimizations.

Novelty. The optimal pipeline depth varies across resources and kernels. The mapping from pipeline depth to kernel performance is profiled offline for each compute resource and stored in Catalog.

#2 Coalescing. For requests with small input sizes, the execution can be bottlenecked by request rate rather than the full hardware bandwidth. Figure 10b shows this effect for `dpk_regex` on BF-3's RegEx engine: executing 512 B (input size), 1 KB, and 2 KB requests individually without coalescing achieves 3.3, 2.6, and 2.2 million ops/s, translating to 1.7, 2.7, and 4.6 GB/s bandwidth, respectively. These numbers are significantly lower than the line rate of the RegEx engine (~7 GB/s). To overcome this limitation, we can coalesce multiple small invocations into a single larger invocation with combined input data, thus bypassing the request rate limit.

Novelty. Coalescing requires that kernels be *coalesceable* but not all kernels qualify. Formally, let K be a kernel, a and b be the inputs, and let $\oplus$ denote concatenation of binary strings (of input or output). A kernel is coalesceable if the following holds:

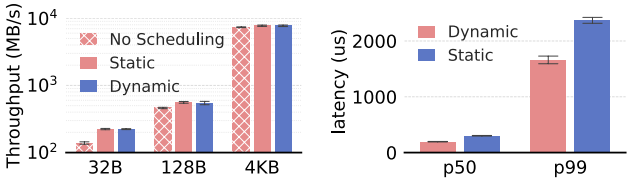
$$K(a \oplus b) = K(a) \oplus K(b)$$

A de/compression kernel is not coalesceable because separately compressed inputs cannot be reliably concatenated and decompressed into outputs exactly the same as original inputs. `dpk_regex` for RegEx matching is coalesceable when the input strings can be concatenated and processed together in a single invocation, where all the matches can be found. The coalesceability of each kernel and the platform-dependent maximum input size (e.g., 16 KB for `dpk_regex` on BF-3) are stored in Catalog.

Coalescing happens opportunistically: a single coalesced invocation will be scheduled for execution by DPMANAGER if the combined input size reaches the maximum limit, or a predefined timeout is reached. Internally, DPMANAGER bookkeeps all the original application-submitted invocation requests in the form of a linked list chain in the original order, and schedules the head with the combined input and output buffers as the single coalesced request.

Table 2: Summary of kernel execution optimizations.

Optimization	Requirement	Speedup
Pipelining	None	Up to 10.2×
Coalescing	Coalesceable kernels	Up to 4×
Parallelization	Multiple DPU cores	Up to 1.8×



(a) Throughput (higher is better) **(b) Latency (lower is better)**

Figure 11: Performance of different scheduling policies.

When the execution completes, dpManager traverses the chain and completes all original requests accordingly. Hence, this optimization is enabled transparently and requires no extra modification or cooperation from applications.

#3 Parallelization. For non-coalesceable kernels, we parallelize their execution with multiple queues, each processed by an additional dpManager thread. Parallelization is beneficial for two reasons. First, most hardware accelerators inherently support parallelism, and thus work items from multiple queues can be processed in parallel internally. Second, perhaps more importantly, small requests may not reach the full throughput achievable by hardware due to the task submission and completion overhead.

Novelty. To enable parallelization, dpManager spawns a configurable number of worker threads to submit requests to their corresponding compute resources concurrently. Each thread manages a dedicated queue where it submits requests and continuously monitors the status of in-flight requests. However, adding more threads beyond a certain number brings diminishing returns. We utilize more threads only when the improvement is significant, and the optimal thread count for each kernel is also profiled and added to Catalog when the framework is deployed on a DPU.

Effects. The effectiveness of the above optimizations is summarized in Table 2. Specifically, as shown in Figures 10a–10c, pipelining enables linear throughput increase until the optimal depth, coalescing allows small requests to harvest the full hardware speed and improves throughput by 4×, and parallelization with two threads achieves 1.8× higher throughput than single-thread execution.

4.4 Scheduling

A dpKernel can execute on SoC cores: for performance, the latency of running a task with small input synchronously on an SoC core may be lower than that of submitting it to the asynchronous hardware accelerator; for portability, the kernel can thus run on any DPU even when the corresponding hardware accelerator is unavailable. Two SoC cores are utilized for kernel execution by default, and the number is configurable and stored in Catalog. dpManager sets up a queue for each core and, depending on the parallelization optimization, one or multiple queues for each hardware accelerator. dpManager schedules requests between the queues.

Static scheduling. Given that a critical goal of scheduling is to improve overall kernel execution throughput, a greedy and static policy is to submit requests always to the hardware accelerator queues (HQs) and only to the SoC core queues (SQs) when HQs are full. This scheduling can fully utilize hardware accelerators, which achieve significantly higher peak processing throughput than the SoC cores. Figure 11a shows the effective decompression throughput of `dpk_decompress` with different input sizes. For large (4 KB) requests, the hardware accelerator dominates the overall performance, and as static scheduling prioritizes HQs, it achieves the best-case throughput. Compared to no scheduling where only the hardware accelerator executes requests, static scheduling effectively utilizes the additional SoC cores, the effect of which is especially pronounced when processing small (32 B) requests and achieves 60% higher throughput.

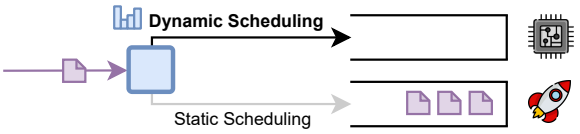


Figure 12: Scheduling requests between queues.

This scheduling policy, however, introduces head-of-line (HOL) blocking. Consider a scenario where the HQs are filled with some large requests but not yet full. An incoming small request will be submitted to the HQs even though the SQs are empty. HOL blocking leads to higher request execution latency, as shown in Figure 11b.

Dynamic scheduling. To address the latency issue in static scheduling, we introduce a dynamic policy where dpManager tracks the queuing delay and kernel execution time for every request in each queue and uses the information to estimate the latency a new request experiences in the queue. Let N be the total number of requests in a queue, E_i be the execution time of the i-th request, and T_i be the total time for the request to complete from admission to completion. As the compute resource processes the requests in a first-come, first-served (FCFS) manner, all requests follow

$$T_i = T_{i-1} + E_i$$

with the base case $T_0 = E_0$. If the execution time for each kernel invocation is known, dpManager can calculate the total latency for any incoming request by accumulating execution time of prior requests in each queue, and schedules it to the queue with the lowest latency. The difference from static scheduling is illustrated in Figure 12—it avoids HOL blocking for small incoming requests.

Dynamic scheduling requires estimating the execution time of each request. To do so, we adopt a statistical model M_K for kernel K that predicts E_i of request i given its input size I_i

$$E_i = M_K(I_i)$$

This is feasible because dpKernels have fixed processing logic and their executions are rather deterministic and predictable. Each kernel has its own model as a histogram stored in Catalog: the bins for different input sizes are profiled offline, and the average kernel execution latency is collected. When dpManager uses a histogram for scheduling, if the input size of a request falls between bins, a linear interpolation is performed to compute the estimated latency; otherwise, it is a fast lookup of the corresponding bin.

Effect. Figure 11b shows the impact of dynamic scheduling on the per-kernel total latency using a mixed workload of small (32 B), medium (1 KB) and large (4 KB) decompression requests. With the statistical modeling, dynamic scheduling reduces both the median and tail latencies (1.5× reduction on average).

5 ACHIEVING PORTABILITY

dpManager employs two components to provide portable kernel execution: Platform Specs and Catalog. They are detailed below.

Table 3: Functions in Platform Specs.

Name	Return Value	Arguments
`setup_dev`	`dev_ctx`	N/A
`setup_mem`	`dpmm`	`dev_ctx`
`register_kernels`	`{kernel_impl}`	N/A
`cleanup_dev`	status	`dev_ctx`
`init`	status	N/A
`execute`	status	`invoke_request`
`poll`	status	`invoke_request`
`cleanup`	status	N/A
`accelerated`	bool	N/A

Platform Specs are function templates instantiated by the service provider who deploys our framework on their target DPU platform. They are used by DPMANAGER to prepare the DPU for kernel execution. Broadly, these functions fall into two categories: device management and kernel implementation, as listed in Table 3.

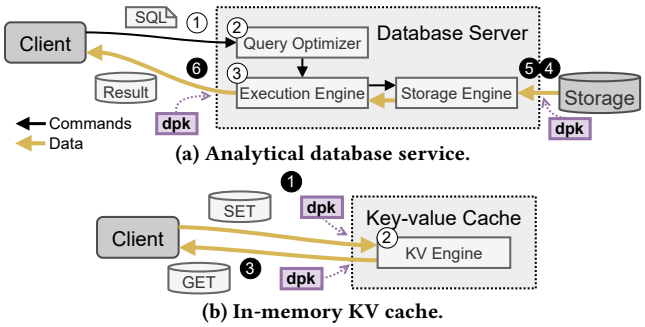
Device management involves the following aspects:

- *Device Initialization.* The `setup_dev` function is to perform DPU-wise setup, e.g., detecting hardware accelerators and SoC cores and setting up these resources. The setup can create a global device context and return its pointer.
- *Memory Initialization.* With an initialized device, its memory is expected to be available for DPMANAGER to allocate and manage (§4.1). The `setup_mem` function serves this purpose. It allocates DPU memory regions and registers them with the vendor driver to make them accessible to hardware accelerators. DPMM is expected to be created and returned.
- *Kernel Registration.* The `register_kernels` function allows to register a list of DPKERNELS that can be invoked on the target DPU. It returns the implementations of registered kernels, which we describe shortly.
- *Device Cleanup.* When DPMANAGER is terminated or fatal exceptions are captured (e.g., during kernel registration), the `cleanup_dev` function releases all allocated resources.

The remaining functions are for implementing kernels on a specific DPU, required when registering kernels to DPMANAGER. Specifically, `init` prepares a kernel for later invocations, e.g., a hardware-accelerated kernel may require information from device and memory. `cleanup` performs the opposite, releasing all resources used by a kernel and deregistering it. `execute` implements the kernel logic using the underlying compute resource: success means that the request will eventually be processed; failure means it cannot be executed. The return value will be relayed to the application when it calls `query`. `poll` is periodically called by DPMANAGER to check the status of a previously scheduled request and complete it up on a status change. Finally, `accelerated` returns if the kernel runs on a hardware accelerator or an SoC core. DPMANAGER enforces an SoC counterpart for each kernel during registration (§4.4).

Catalog stores metadata for kernels and compute resources, which helps port the kernel execution scheduling and optimizations. The metadata in Catalog is discussed in §4 and is summarized as follows:

- For each kernel, the metadata consists of coalesceability, minimum and maximum input size, and a histogram that maps input size to execution time.



(a) Analytical database service.

(b) In-memory KV cache.

Figure 13: Applying DPKERNELS when data flows across the boundary of (i.e., in to or out from) the server host.

- For each compute resource (a hardware accelerator or SoC core), the metadata consists of the maps from pipelining depth and parallelism to request execution throughput.

6 APPLICATION SCENARIOS

We now discuss the general applicability of DPKERNELS and show how they can benefit popular cloud data processing services.

6.1 Applicability

In general, DPKERNELS target scenarios where a data processing system running on a server host (1) moves a significant amount of data to/from local PCIe-attached devices (e.g., SSDs) or remote servers and (2) associates the data path with compute-intensive operations. With these two conditions, data will flow through the DPU, and the data-path computation can benefit from hardware acceleration. They cover practical deployments of cloud data services as we show in §6.2 and §6.3. However, when the two conditions cannot be satisfied, DPKERNELS have limited or no benefits. For example, data is cached and processed in the host (i.e., there is no data movement), or the volume of data being moved is low. We also point out such cases in the two applications next.

6.2 Analytical Database Service

Cloud data analytics services such as Azure Synapse SQL [49] and BigQuery [25] serve SQL queries from clients by reading data from local or cloud storage devices and processing it on the database servers. Figure 13a shows the overall architecture. Its end-to-end workflow for serving queries consists of the following steps. Steps where DPKERNELS apply (4–6) are highlighted.

- ① Query transfer: query is sent from the client to the server.
- ② Query optimization: query is parsed and optimized by the query optimizer to generate a physical execution plan.
- ③ Query execution: plan is executed by execution engine.
- ❹ **Data selection**: data is selected with query-specific predicates on columns and rows. String predicates can be converted into RegEx matching and accelerated by `dpk_regex`.
- ❺ **Data loading**: selected data is loaded from storage. As data is often compressed and encrypted in open formats (e.g., Parquet [8]) on disks, data decryption and decompression can be accelerated by `dpk_decrypt` and `dpk_decompress`.
- ❻ **Result transfer**: query result produced by the execution engine is finally returned back to the client. In cases where results need to be encrypted and compressed [16], this step can be accelerated by `dpk_encrypt` and `dpk_compress`.

Steps 1–3 do not move data across the boundary of the server host, and thus DPKERNELS do not apply. In Step 6, as most analytics queries generate small results, the benefits of DPKERNELS are limited. We evaluate how these steps contribute to end-to-end query execution times and the actual benefits of DPKERNELS in Section 8.

6.3 In-memory Key-value Cache

In-memory key-value (KV) cache, such as FASTER [47], Redis [48], and Memcached [11], provides intuitive SET and GET API for caching frequently-accessed objects in memory. As shown in Figure 13b, when data flows into the KV engine on the host with SET operations (❶) and is associated with computational tasks (e.g., compression to save memory footprint), DPKERNELS (e.g., `dpk_compress`) can be applied to accelerate the data-path throughput. Similarly, DPKERNELS also apply when data flows out with GET operations (❸). Beyond data-path computation, DPKERNELS do not impact how objects are processed in the KV engine (②). Section 8 also shows how DPKERNELS can improve overall KV cache SET and GET throughput.

7 IMPLEMENTATION AND EVALUATION

Prototype. The prototype of DPMANAGER and DPKERNELS has been implemented with 10,030 lines of C/C++ code. DPMANAGER adopts `mimalloc` [37] to implement its memory (de)allocation in DPMM, a lightweight allocator that provides fast allocation of arbitrary sizes, in order to be generally amenable to various data processing systems. Each work queue DPMANAGER creates is implemented as a bounded array with the pipelining depth as the size. The histogram for each kernel is materialized as an array, with each element being a bucket, facilitating fast lookup. In addition, Catalog is implemented as a succinct struct that bookkeeps all metadata as members.

Applications. We evaluate the efficacy of DPKERNELS with the applications in Section 6. Their detailed configurations are as follows.

Analytical database service. We deploy a data analytics service that stores database tables in Parquet [8] where data pages are protected by Parquet Modular Encryption [9] on server-attached NVMe SSDs and queries them with DuckDB [57]. Clients send SQL queries to the server and receive query answers in the Arrow [7] format. We use DPKERNELS to accelerate Steps ❹❺❻ in Section 6.2 and evaluate their benefits with the TPC-H benchmark [68] (scale factor 50).

In-memory KV cache. We deploy a KV cache service that serves client GET and SET requests backed by FASTER [14], a production KV library. The objects in the cache are compressed to save memory. We accelerate SETs (Step ❶ in Section 6.3) with `dpk_compress` and GETs (Step ❸) with `dpk_decompress`. For evaluation, we use the YCSB benchmark [17] with 250 million 8 B-key 1 KB-value records accessed with uniform and skewed (Zipfian $\theta = 1.5$) distributions. The values are loaded from the Wikipedia dataset [72].

Hardware setup. The client and server run on dedicated machines connected by 100 Gbps network links. Each machine is a Dell PowerEdge R7625 server with an AMD EPYC CPU (12 cores @2.90 GHz), 128 GB DDR5 memory, and 2 TB NVMe SSDs. The machine running the server is also equipped with different DPUs described next.

DPU platforms. We demonstrate the portability of DPMANAGER and DPKERNELS by running them on three DPUs from different generations and vendors: NVIDIA BlueField-2, NVIDIA BlueField-3, and AMD Pensando Elba. Table 1 shows the configurations.

8 EXPERIMENTAL RESULTS

8.1 Analytical Databases

Figure 14 shows the detailed breakdown of the end-to-end performance of the data analytics service for each of the TPC-H queries. The key findings are summarized as follows.

First, data loading is the most expensive step in the end-to-end query processing. Compared to all other steps, data loading (Step ❺, i.e., reading data pages from Parquet files and performing decryption and decompression) consumes most of the time (75.7% on average) for nearly all queries. The second most time-consuming step is DuckDB's query execution on the host (Step ③, 23.7%), followed by data selection (Step ❹), which is already insignificant (<1%). Other steps (①②❻) barely affect end-to-end performance.

Second, DPKERNELS *can significantly improve overall performance.* This is because they can accelerate the most expensive step in the workflow. Specifically, `dpk_decrypt` and `dpk_decompress` on BF-3 are 4.1× and 2.5× faster than decryption and decompression on the server with all cores. On average, they bring 3× speedup to data loading and 2× speedup to end-to-end query processing.

Finally, not all DPKERNELS *are effective.* Although `dpk_regex` is 6.8× faster than DuckDB's string predicate evaluation, the benefits are not reflected in the end-to-end performance because data selection overall is not a performance bottleneck.

8.2 KV Cache

We now assess DPKERNELS' benefits for accelerating the data-path efficiency of a KV cache. Compressing the values saves 60% memory footprint for the cache. However, the computational tasks ((de)compression) now become the performance bottleneck: without decompression, FASTER can achieve 3.6 million ops/s (Mops) GET throughput with a single core on the server, which drops to 0.5 Mops when decompression is required. Figures 15 and 16 show the GET throughput for serving YCSB uniform and Zipfian workloads. We observe that even with 8 server CPU cores, the throughput is still below 4 Mops/s. For comparison, when offloading to the SoC cores on BF-2, the throughput further degrades to 3 Mops/s with 8 cores. The improvement with `dpk_decompress` (backed by the BF-2 decompression accelerator) is significant. Without consuming any cores on the server, the application achieves 8.7 Mops/s, 17.8× and 2.2× faster than 1 core and 8 cores on the server, respectively.

Serving SET requests requires compressing the records before insertion. As compression is more costly than decompression, the task is even more challenging for CPUs. As shown in Figures 17 and 18, running the compression on the server achieves 0.1 Mops/s with a single core and 0.8 Mops/s with 8 cores. The degradation on DPU SoC cores is severer: just 0.3 Mops/s with 8 cores. Offloading it with `dpk_compress` on BF-2 compression accelerator speeds up the application by 76× compared to one server core and 9.8× compared to eight server cores, achieving 7.8 Mops/s.

We observe that in both analytical databases and KV cache, encryption/decryption and compression/decompression are frequently-executed and expensive operations in data paths. Accelerating them on DPUs can thus in general improve data processing performance.

8.3 Portable Efficiency

We next examine whether the design efficiency of DPMANAGER generalizes to different DPU platforms. Specifically, we evaluate the

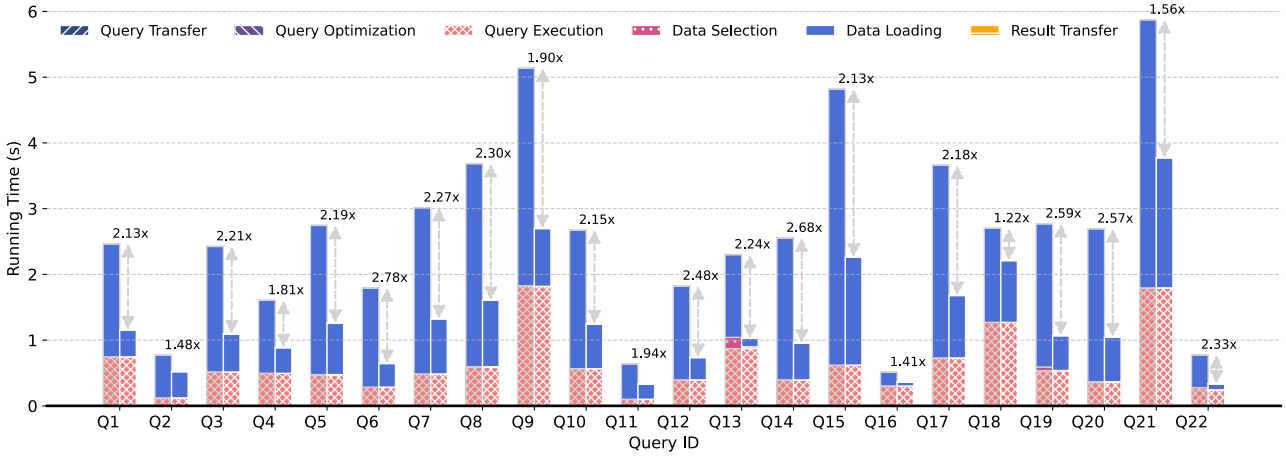


Figure 14: Breakdown of the end-to-end performance of the data analytics service on TPC-H. For each query, the left bar represents server-based deployment, and the right bar represents dpKernels-accelerated deployment.

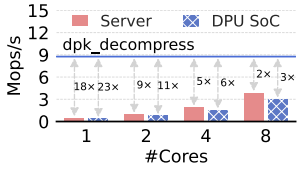


Figure 15: KV GET (uniform).

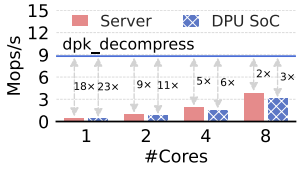


Figure 16: KV GET (Zipfian).

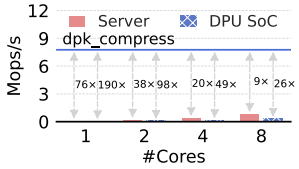


Figure 17: KV SET (uniform).

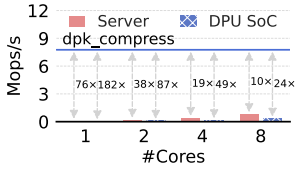


Figure 18: KV SET (Zipfian).

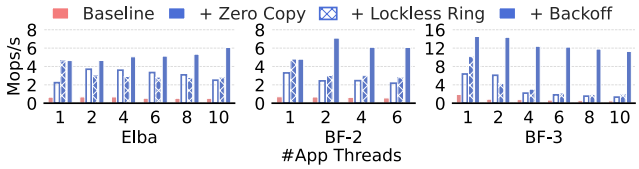


Figure 19: Workflow efficiency across different DPUs. Optimizations are applied incrementally.

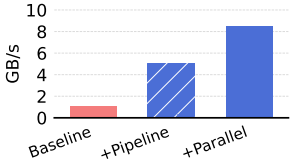


Figure 20: KV cache SET operations with dpk_compress.

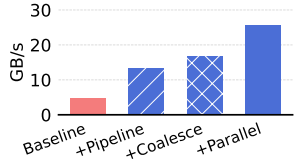
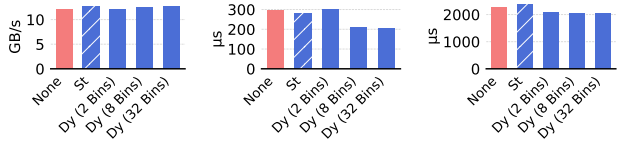


Figure 21: Kernel invocations for mixed applications.



(a) Throughput **(b) Median latency** **(c) Tail latency**

Figure 22: Impact of scheduling policies. St and Dy stand for static scheduling and dynamic scheduling, respectively.

benefits of dpManager's zero-copy memory management (§4.1) and lockless ring buffer and spaced backoff (§4.2). Execution optimizations for dpKernels are evaluated in the next sections.

Figure 19 shows the throughput improvement of dpk_nop on Elba, BF-2, and BF-3 ascribed to our system design. We vary the number of application threads that call `invoke` in parallel on each DPU. The baseline dpManager copies the input data (1 KB) of each request and sends the requests from the applications with the concurrent queue (§2.2). The throughput of this baseline is low across all three DPUs: up to 0.7 Mops on Elba and BF-2 and up to 1.8 Mops on BF-3, even without any thread contention. Zero-copy memory management eliminates the copy overhead and significantly improves the invocation throughput by 5.1× (Elba), 4× (BF-2), and 3.5× (BF-3), respectively. Compared to the concurrent queue, our lockless ring buffer reduces contention overhead when multiple application threads concurrently interact with dpManager, further improving throughput up to 1.1× (Elba), 1.3× (BF-2), and 2.3× (BF-3). Finally, by avoiding application contention with spaced backoff, the invocation request submission speed is significantly boosted, which is on average 8.7× (Elba), 9.4× (BF-2), and 17× (BF-3) higher than the baseline, achieving 6 Mops, 6.1 Mops, and 12 Mops, respectively.

This experiment verifies that the efficiency of dpManager's design generalizes to different DPUs across generations and vendors.

8.4 Optimizations and Scheduling

We also evaluate the effectiveness of the dpKernel execution optimizations and conduct sensitivity tests on dpManager's scheduler.

Kernel execution optimizations. Figure 20 shows improvements when individual optimizations are enabled for `dpk_compress`, and Figure 21 shows the scenario in which multiple applications run concurrently and invoke different dpKernels (`dpk_(de)compress` and `dpk_regex`) on the same DPU (each application runs on one or two SoC cores). Specifically, as shown in Figure 20: when `dpk_compress`

is executed without our optimizations on BF-2, performance is relatively low; pipelining and parallelization increases the throughput from 1 GB/s to 5 GB/s and 8.5 GB/s. To evaluate the ability of our framework to support multiple applications, Figure 21 shows the aggregated throughput of multiple applications on BF-2, which has hardware accelerators to execute the DPKERNELS adopted in all the applications. We make two key observations. First, the aggregated bandwidth across the data paths is higher than that in each individual application, showing the ability of DPMANAGER to accept and schedule requests from different applications *to utilize multiple hardware accelerators*. Second, *every optimization can effectively improve overall data-path throughput*: pipelining 288%, coalescing 26%, and parallelization 52%.

Dynamic scheduling. In the blob storage application (accelerated with `dpk_decompress`), we generate an equal number of requests of three sizes uniformly: 64 B, 4 KB, and 1 MB, and measure the impact of different scheduling policies on the request throughput and latency on BF-3. In particular, as DPMANAGER's dynamic scheduling requires a histogram that profiles the execution times of input sizes, we measure how sensitive its behavior is to the profiling precision: a higher precision leads to more fine-grained profiling and thus more buckets in the histogram. Figure 22 reports the result. With two SoC cores, static scheduling can achieve 5% higher throughput compared to no scheduling where only the hardware accelerator is utilized. With a 32-bucket histogram (i.e., 32 data points are profiled between the min and max input sizes), dynamic scheduling achieves the same throughput as static scheduling but 1.5× and 1.2× lower median and tail latencies, respectively. A 32-bucket histogram is materialized with a 32-integer array—the storage overhead to bookkeep this metadata in Catalog is negligible.

8.5 Reducing Hardware Cost

We now analyze how DPKERNELS can save the hardware cost of cloud data processing. In general, accelerating compute-intensive tasks in data processing with DPKERNELS on DPUs can reduce hardware cost because as we show in the two applications (§8.1 and §8.2) data processing workloads can complete in significantly shorter times. However, accurate cost measurements are difficult because cloud providers currently do not offer public instances with DPUs. We estimate the cost reduction enabled by DPKERNELS as:

$$\text{Cost Reduction} = \frac{C_{\text{Server}}}{C_{\text{DPKERNELS}}} = \frac{T_{\text{Server}} \times P_{\text{Server}}}{T_{\text{DPKERNELS}} \times P_{\text{Server+DPU}}}$$

where T and P denote the end-to-end execution time and the hardware price per time unit, respectively. We estimate P_{Server} by the price of the VM instance on Microsoft Azure that most closely matches our hardware setup (`E16as v5`, USD$0.9040/hour) and $P_{\text{Server + DPU}}$ by $P_{\text{Server}} + \alpha P_{\text{Server}}$ where α is the relative hardware cost of a DPU compared to a server and equals 0.44783 for the BF-3 DPU and the server in our on-premise cluster. The cost reduction of DPKERNELS is on average 1.4× for the data analytics service (§6.2) on TPC-H and 4× for the KV cache service (§6.3) on YCSB.

8.6 Saving Engineering Effort

We finally show the flexibility and portability enabled by our framework. Without DPKERNELS, the developer will need to use the low-level vendor-specific SDKs (e.g., the ones in Table 1) to program and access relevant compute resources on *every* DPU for *every* application. Table 4 summarizes the effort for accelerating the analytical database service, the KV cache, and other potential production data processing systems on the three DPUs (BF-2, BF-3, and Elba): each application requires thousands of LoC with vendor SDKs. In comparison, DPKERNELS can drastically save the engineering effort: each application only needs under 100 LoC modification that is portable across different DPUs. Benefiting from the two levels of abstraction, instantiating Platform Specs (Table 3) for each DPU is one-time effort from the service provider and can be directly used for any future application.

Table 4: Lines of code (LoC) for implementing acceleration for different applications across BF-2, BF-3, and Elba.

		DB	KV	New App
Vendor SDKs		3900	2300	1000s
DPKERNELS	**App modification**	63	38	10s
	Platform Specs	3300	0	0

9 RELATED WORK

Given the hardware availability of DPUs, a line of recent work has explored DPU offloading to improve various aspects of cloud data processing [19, 27, 28, 41, 61, 79, 84]. Specifically, DDS [79] offloads storage read operations to DPUs to save storage server host CPUs and reduce remote storage latency. HiDPU [84] proposes a hybrid index that spans between the host and the DPU to address the memory limitation on the DPU. SmartShuffle [41] offloads shuffle operations in Spark to DPUs. DFlush [19] utilizes the DMA engine and RISC-V cores on BlueField-3 to offload background compaction tasks in RocksDB. DPDPU [27] is a general proposal for using DPUs to improve data processing efficiency in different aspects.

In the general systems realm, iPipe [44] proposes an actor-based framework to write distributed applications on DPUs. Gimbal [50] uses DPUs to process multi-tenant storage requests. LineFS [34] offloads distributed file system activities to DPUs, employing compression to reduce network traffic. More recently, IO-TCP [35] offloads the data plane of TCP to a DPU for large data streaming, and PD3 [61] uses a DPU to prefetch for disaggregated memory.

Compared to the previous work, our proposal is unique in its focus on DPU compute resources, especially the hardware accelerators, and uses them for portable data-path efficiency.

10 CONCLUSION

We presented a computing framework to harvest DPU compute resources, i.e., hardware accelerators and SoC cores, to improve the data-path efficiency in cloud data processing. It consists of two key components: DPKERNELS and DPMANAGER, which provide two levels of abstractions that achieve "offload once, run anywhere". This portability is especially important given the heterogeneity of DPUs and their compute resources. DPKERNELS can significantly improve data-path efficiency in realistic applications, and their benefits generalize to different DPUs without application changes.

ACKNOWLEDGMENTS

We thank the reviewers for their constructive feedback. This work was funded by NSERC RGPIN-2023-04834 DGECR-2023-00308.

REFERENCES

[1] Azim Afroozeh and Peter A. Boncz. 2023. The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (2023), 2132–2144. https://doi.org/10.14778/3598581.3598587

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, August 17-22, 2008*. ACM, 63–74.

[3] Alibaba Cloud Community. 2022. A Detailed Explanation about Alibaba Cloud CIPU. https://www.alibabacloud.com/blog/599183.

[4] All Things Distributed. 2020. Reinventing virtualization with the AWS Nitro System. https://www.allthingsdistributed.com/2020/09/reinventing-virtualization-with-nitro.html.

[5] AMD. 2023. AMD Pensando SSDK - Product Brief. https://www.amd.com/content/dam/amd/en/documents/pensando-technical-docs/product-briefs/pensando-ssdk-product-brief.pdf.

[6] AMD. 2025. AMD Pensando Infrastructure Accelerators. https://www.amd.com/en/accelerators/pensando.

[7] Apache. 2026. Apache Arrow. https://arrow.apache.org/.

[8] Apache. 2026. Apache Parquet. https://parquet.apache.org/.

[9] Apache Parquet. 2026. Parquet Modular Encryption. https://parquet.apache.org/docs/file-format/data-pages/encryption/.

[10] Arvind Arasu, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. 2015. Transaction processing on confidential data using cipherbase. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman (Eds.). IEEE Computer Society, 435–446. https://doi.org/10.1109/ICDE.2015.7113304

[11] AWS. 2026. Memcached | Distributed Key-Value Store. https://aws.amazon.com/elasticache/what-is-memcached/.

[12] Nils Boeschen, Tobias Ziegler, and Carsten Binnig. 2024. GOLAP: A GPU-in-Data-Path Architecture for High-Speed OLAP. *Proc. ACM Manag. Data* 2, 6, Article 237 (Dec. 2024), 26 pages. https://doi.org/10.1145/3698812

[13] Broadcom. 2020. Broadcom Stingray SmartNIC Accelerates Baidu Cloud Services. https://www.broadcom.com/company/news/product-releases/53106.

[14] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 275–290. https://doi.org/10.1145/3183713.3196898

[15] Xinyi Chen, Liangcheng Yu, Vincent Liu, and Qizhen Zhang. 2023. Cowbird: Freeing CPUs to Compute by Offloading the Disaggregation of Memory. In *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*. ACM, 1060–1073.

[16] Monica Chiosa, Fabio Maschi, Ingo Müller, Gustavo Alonso, and Norman May. 2022. Hardware Acceleration of Compression and Encryption in SAP HANA. *Proc. VLDB Endow.* 15, 12 (2022), 3277–3291.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[18] Abhishek Dhamija, Balasubramanian Madhavan, Hechao Li, Jie Meng, Shrikrishna Khare, Madhavi Rao, Lawrence Brakmo, Neil Spring, Prashanth Kannan, Srikanth Sundaresan, and Soudeh Ghorbani. 2024. A large-scale deployment of DCTCP. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*. USENIX Association, 239–252.

[19] Chen Ding, Kai Lu, Quanyi Zhang, Zekun Ye, Ting Yao, Daohui Wang, Huatao Wu, and Jiguang Wan. 2025. DFlush: DPU-Offloaded Flush for Disaggregated LSM-based Key-Value Stores. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–28.

[20] Chen Ding, Jian Zhou, Kai Lu, Sicen Li, Yiqin Xiong, Jiguang Wan, and Ling Zhan. 2024. D^2Comp: Efficient Offload of LSM-tree Compaction with Data Processing Units on Disaggregated Storage. *ACM Trans. Archit. Code Optim.* 21, 3 (2024), 46:1–46:22. https://doi.org/10.1145/3656584

[21] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 31–46. https://doi.org/10.1145/2723372.2747642

[22] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 249–264.

[23] Girish Bablani. 2023. Microsoft announces acquisition of Fungible to accelerate datacenter innovation. https://blogs.microsoft.com/blog/2023/01/09/microsoft-announces-acquisition-of-fungible-to-accelerate-datacenter-innovation/.

[24] GitHub. 2025. A bounded multi-producer multi-consumer concurrent queue written in C++11. https://github.com/rigtorp/MPMCQueue.

[25] Google Cloud. 2026. BigQuery overview. https://docs.cloud.google.com/bigquery/docs/introduction.

[26] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*. ACM, 202–215.

[27] Jiasheng Hu, Philip Bernstein, Jialin Li, and Qizhen Zhang. 2025. DPDPU: Data Processing with DPUs. In *15th Annual Conference on Innovative Data Systems Research (CIDR'25)*.

[28] Jiasheng Hu, Chihan Cui, Anna Li, Raahil Vora, Yuanfan Chen, Philip A. Bernstein, Jialin Li, and Qizhen Zhang. 2025. dpBento: Benchmarking DPUs for Data Processing. *CoRR* abs/2504.05536 (2025).

[29] Intel. 2025. Intel Infrastructure Processing Unit (Intel IPU). https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html.

[30] Hao Jiang and Aaron J. Elmore. 2018. Boosting data filtering on columnar encoding with SIMD. In *Proceedings of the 14th International Workshop on Data Management on New Hardware, Houston, TX, USA, June 11, 2018*, Wolfgang Lehner and Kenneth Salem (Eds.). ACM, 6:1–6:10. https://doi.org/10.1145/3211922.3211932

[31] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.* 1, 2 (2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[32] Kalray. 2025. Kalray MPPA DPU Manycore. https://www.kalrayinc.com/products/mppa-technology/.

[33] John Kim, William J. Dally, Steve Scott, and Dennis Abts. 2008. Technology-Driven, Highly-Scalable Dragonfly Topology. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China*. IEEE Computer Society, 77–88.

[34] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 756–771.

[35] Taehyun Kim, Deondre Martin Ng, Junzhi Gong, Youngjin Kwon, Minlan Yu, and KyoungSoo Park. 2023. Rearchitecting the TCP Stack for I/O-Offloaded Content Delivery. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 275–292. https://www.usenix.org/conference/nsdi23/presentation/kim-taehyun

[36] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26. https://doi.org/10.1145/3589263

[37] Daan Leijen, Benjamin Zorn, and Leonardo De Moura. 2019. Mimalloc: Free list sharding in action. In *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*. Springer, 244–265.

[38] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proc. VLDB Endow.* 9, 14 (2016), 1647–1658. https://doi.org/10.14778/3007328.3007331

[39] Yinan Li, Jianan Lu, and Badrish Chandramouli. 2023. Selection Pushdown in Column Stores using Bit Manipulation Instructions. *Proc. ACM Manag. Data* 1, 2 (2023), 178:1–178:26. https://doi.org/10.1145/3589323

[40] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*. ACM, 44–58.

[41] Jiaxin Lin, Tao Ji, Xiangpeng Hao, Hokeun Cha, Yanfang Le, Xiangyao Yu, and Aditya Akella. 2023. Towards Accelerating Data Intensive Application's Shuffle Process Using SmartNICs. In *Abstract Proceedings of the 2023 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2023, Orlando, FL, USA, June 19-23, 2023*, Evgenia Smirni, Konstantin Avrachenkov, Phillipa Gill, and Bhuvan Urgaonkar (Eds.). ACM, 9–10.

[42] Shu Lin, Arunprasad P. Marathe, Per-Åke Larson, Chong Chen, Calvin Sun, Paul Lee, Weidong Yu, Jianwei Li, Juncai Meng, Roulin Lin, Xiaoyang Chen, and Qingping Zhu. 2022. Near Data Processing in Taurus Database. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 1662–1674. https://doi.org/10.1109/ICDE53745.2022.00170

[43] Jianshen Liu, Carlos Maltzahn, Craig D. Ulmer, and Matthew Leon Curry. 2021. Performance Characteristics of the BlueField-2 SmartNIC. *CoRR* abs/2105.06619

(2021).
[44] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*. 318–333.
[45] Marvell. 2025. Marvell Octeon Data Processing Units (DPUs). https://www.marvell.com/products/data-processing-units.html.
[46] Microsoft. 2024. Stored procedures (Database Engine). https://learn.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine.
[47] Microsoft. 2025. FASTER: A fast concurrent persistent key-value store and log, in C# and C++. https://microsoft.github.io/FASTER/.
[48] Microsoft. 2026. Azure Managed Redis. https://azure.microsoft.com/en-us/products/managed-redis.
[49] Microsoft Azure. 2026. Synapse SQL architecture. https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/overview-architecture.
[50] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, Fernando A. Kuipers and Matthew C. Caesar (Eds.). ACM, 106–122. https://doi.org/10.1145/3452296.3472940
[51] Ruslan Nikolaev. 2019. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue. In *33rd International Symposium on Distributed Computing, DISC 2019, Budapest, Hungary, October 14-18, 2019 (LIPIcs)*, Jukka Suomela (Ed.), Vol. 146. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:16. https://doi.org/10.4230/LIPICS.DISC.2019.28
[52] NVIDIA. 2025. Transform the Data Center With NVIDIA DPUs. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.
[53] NVIDIA Developer. 2025. NVIDIA BLUEFIELD-3 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf.
[54] NVIDIA Developer. 2025. NVIDIA DOCA Software Framework. https://developer.nvidia.com/networking/doca.
[55] Peter Judge. 2022. Intel and Google Cloud jointly launch data center accelerator chip. https://www.datacenterdynamics.com/en/news/intel-and-google-cloud-jointly-launch-data-center-accelerator-chip/.
[56] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Muhammad Mukarram Bin Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve D. Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. 2022. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *SIGCOMM '22: ACM SIGCOMM 2022 Conference, Amsterdam, The Netherlands, August 22 - 26, 2022*. ACM, 66–85.
[57] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212
[58] Red Hat Inc. 2009. Lockless Ring Buffer Design. https://docs.kernel.org/trace/ring-buffer-design.html
[59] Yanjing Ren, Yuanming Ren, Xiaolu Li, Yuchong Hu, Jingwei Li, and Patrick P. C. Lee. 2024. ELECT: Enabling Erasure Coding Tiering for LSM-tree-based Storage. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. USENIX Association, Santa Clara, CA, 293–310. https://www.usenix.org/conference/fast24/presentation/ren
[60] Eyal Rozenberg and Peter A. Boncz. 2017. Faster across the PCIe bus: a GPU library for lightweight decompression: including support for patched compression schemes. In *Proceedings of the 13th International Workshop on Data Management on New Hardware, DaMoN 2017, Chicago, IL, USA, May 15, 2017*. ACM, 8:1–8:5. https://doi.org/10.1145/3076113.3076122
[61] Sidharth Sankhe, Felix Zhang, Umayrah Chonee, Sherman Lim, Jiasheng Hu, Jialin Li, and Qizhen Zhang. 2026. PD3: Prefetching Data with DPUs for Disaggregated Memory. In *23rd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2026, Renton, WA, May 4-6, 2026*, Srikanth Kandula and Hakim Weatherspoon (Eds.). USENIX Association, 1207–1223. https://www.usenix.org/conference/nsdi26/presentation/sankhe
[62] Mo Sha, Yifan Cai, Sheng Wang, Linh Thi Xuan Phan, Feifei Li, and Kian-Lee Tan. 2024. Object-oriented Unified Encrypted Memory Management for Heterogeneous Memory Architectures. *Proc. ACM Manag. Data* 2, 3 (2024), 155. https://doi.org/10.1145/3654958
[63] Junyi Shu, Kun Qian, Ennan Zhai, Xuanzhe Liu, and Xin Jin. 2024. Burstable Cloud Block Storage with Data Processing Units. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 783–799. https://www.usenix.org/conference/osdi24/presentation/shu
[64] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael J. Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2021. DBOS: A DBMS-oriented Operating System. *Proc. VLDB Endow.* 15, 1 (2021), 21–30. https://doi.org/10.14778/3485450.3485454
[65] The DPDK Project. 2024. Data Plane Development Kit (DPDK). Data Plane Development Kit Official Website. http://www.dpdk.org Accessed on June 9, 2026.
[66] Neil C. Thompson and Svenja Spanuth. 2021. The decline of computers as a general purpose technology. *Commun. ACM* 64, 3 (Feb. 2021), 64–72. https://doi.org/10.1145/3430936
[67] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. 2021. DFI: the data flow interface for high-speed networks. In *Proceedings of the 2021 International Conference on Management of Data*. 1825–1837.
[68] TPC. 2026. TPC-H Homepage. https://www.tpc.org/tpch/.
[69] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 789–796.
[70] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. Plush: A Write-Optimized Persistent Log-Structured Hash-Table. *Proc. VLDB Endow.* 15, 11 (2022), 2895–2907. https://doi.org/10.14778/3551793.3551839
[71] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated Database Systems. In *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*. ACM, 37–44.
[72] Wikimedia. 2025. English Wikipedia dump. https://dumps.wikimedia.org/enwiki/20250401/.
[73] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. 2013. Vectorizing Database Column Scans with Complex Predicates. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2013, Riva del Garda, Trento, Italy, August 26, 2013*, Rajesh Bordawekar, Christian A. Lang, and Bugra Gedik (Eds.). 1–12. http://www.adms-conf.org/2013/muller_adms13.pdf
[74] Chenyuan Wu, Mohammad Javad Amiri, Jared Asch, Heena Nagda, Qizhen Zhang, and Boon Thau Loo. 2022. FlexChain: An Elastic Disaggregated Blockchain. *Proc. VLDB Endow.* 16, 1 (2022), 23–36.
[75] Yifei Yang, Matt Youill, Matthew E. Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. *Proc. VLDB Endow.* 14, 11 (2021), 2101–2113. https://doi.org/10.14778/3476249.3476265
[76] Xiangyao Yu, Matt Youill, Matthew E. Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS Using S3 Computation. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1802–1805. https://doi.org/10.1109/ICDE48307.2020.00174
[77] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. 2021. Redy: Remote Dynamic Memory Cache. *Proc. VLDB Endow.* 15, 4 (2021), 766–779.
[78] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. 2022. CompuCache: Remote Computable Caching using Spot VMs. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org.
[79] Qizhen Zhang, Philip A. Bernstein, Badrish Chandramouli, Jason Hu, and Yiming Zheng. 2024. DDS: DPU-optimized Disaggregated Storage. *Proc. VLDB Endow.* 17, 11 (2024), 3304–3317.
[80] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. 2020. Rethinking Data Management Systems for Disaggregated Data Centers. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*.
[81] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *Proc. VLDB Endow.* 13, 9 (2020), 1568–1581. https://doi.org/10.14778/3397230.3397249
[82] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. [n.d.]. Optimizing Data-intensive Systems in Disaggregated Data Centers with TELEPORT. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. 1345–1359.
[83] Xinjing Zhou, Viktor Leis, Xiangyao Yu, and Michael Stonebraker. 2025. OLTP Through the Looking Glass 16 Years Later: Communication is the New Bottleneck. In *15th Annual Conference on Innovative Data Systems Research (CIDR'25)*.
[84] Wenbin Zhu, Zhaoyan Shen, Qian Wei, Renhai Chen, Xin Yao, Dongxiao Yu, and Zili Shao. 2025. {HiDPU}: A {DPU-Oriented} Hybrid Indexing Scheme for Disaggregated Storage Systems. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 271–285.