Analyzing Near-Network Hardware Acceleration with Co-Processing on DPUs

Dimitrios Giouroukis
BIFOLD, TU Berlin
dimitrios.giouroukis@tu-berlin.de

Dwi P. A. Nugroho
BIFOLD, TU Berlin
d.nugroho@tu-berlin.de

Varun Pandey
BIFOLD, TU Berlin
varun.pandey@tu-berlin.de

Steffen Zeuch
BIFOLD, TU Berlin
steffen.zeuch@tu-berlin.de

Volker Markl
BIFOLD, TU Berlin, DFKI GmbH
volker.markl@tu-berlin.de

ABSTRACT

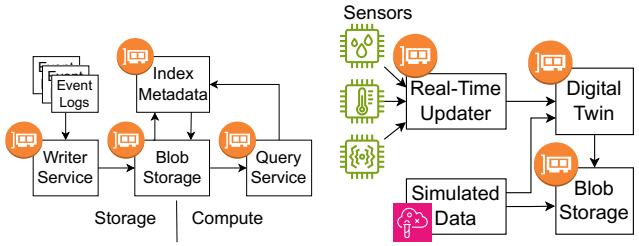
Data Processing Units (DPUs) are PCIe network cards (SmartNICs) equipped with specialized hardware accelerators for data processing. DPUs offer the opportunity to process data near the hardware network stack (near-network). By enabling near-network computation, DPUs reduce CPU load and improve end-to-end performance, an increasingly attractive approach to trends like compute-storage disaggregation and real-time data ingestion. However, existing research on DPU-based processing often overlooks hardware acceleration or relies on static offloading to the ARM subsystem, leaving open questions about how best to split work (or co-process) with the host CPU. In this paper, we analyze near-network hardware acceleration with co-processing on DPUs, revealing that DPU performance varies significantly depending on input data types, task and query-imposed configurations. Through our micro-benchmark experiments, we explore partial offloads and co-processing strategies that demonstrate the trade-offs between higher throughput against reconfiguration overhead on DPUs. Our findings offer practical insights for data systems practitioners seeking to leverage near-network accelerators in data processing pipelines.

PVLDB Reference Format:
Dimitrios Giouroukis, Dwi P. A. Nugroho, Varun Pandey, Steffen Zeuch, and Volker Markl.Analyzing Near-Network Hardware Acceleration with Co-Processing on DPUs. PVLDB, 18(13): 5689 - 5702, 2025.
doi:10.14778/3773731.3773743

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/digiou/dpu-coprocessing-artifacts.

1 INTRODUCTION

In recent years, data management systems designed for the "infinitely" scalable cloud (so called cloud-native) make use of resource-dissaggregated designs [4, 16, 63, 66]. The key advantage of this design is effective and efficient use of resources since applications use shared pools for each resource. Thus, they consume only what they need for specific tasks [1, 12, 58]. Resource disaggregation enables the decoupling of software logic from managing the underlying resource, thus removing the burden of resource management from the application logic. To achieve this level of decoupling, cloud providers use specialized hardware, such as memory controllers, solid-state drives, or network equipment, which enable computation near or on the resource, thus bypassing the CPU [15, 35, 50]. However, while bypassing the CPU is beneficial, resource disaggregation introduces a new challenge: increased network usage and latency from using physically separated resources. Since resources are not colocated, applications are fully reliant on the network for data retrieval, even when data resides within the cloud, which leads to potential network congestion or higher processing latency.

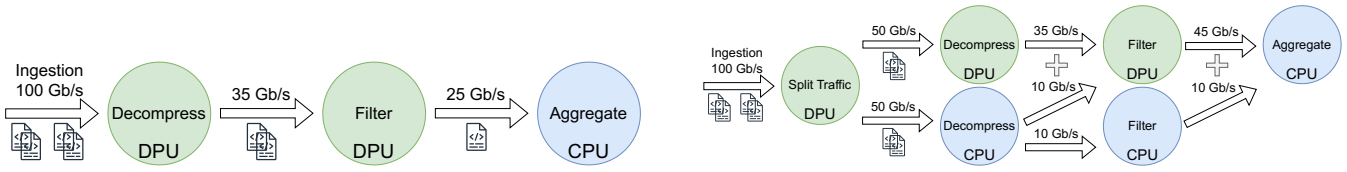


(a) Observability Platform (b) Digital Twins Platform

Figure 1: Disaggregation in Cloud Applications

In order to mitigate the increased network usage and latency in the disaggregated cloud, current networking SmartNIC hardware (or Data Processing Units, DPUs) can perform complex compute tasks during the ingestion of data [33, 34]. This style of processing closer to the network stack (near-network) allows systems to react to data or infrastructure changes faster and mitigate additional costs. For example, consider the two applications in Figure 1. The first application (Figure 1a) is an observability platform, which uses disaggregation to decouple storage costs from compute costs. The second application (Figure 1b) is a digital twins platform that maintains digital models of devices in order to answer what-if queries, e.g., driving range estimation for electrical vehicles speeding over 150 km/hr. The observability platform stores metrics as text logs in the cloud's object storage and makes heavy use of its networked file system to cache or load data when answering user queries. To keep storage costs low, the platform compresses data before storage and consequently decompresses data before processing, a common best practice with cloud object stores [13, 45, 60]. The digital twins platform not only has to store data from its device sensor fleet, but it also has to combine simulated data with real-time data that match live user queries; thus improving query results. To do so, the digital twins platform filters relevant data during ingestion. With

Proceedings of the VLDB Endowment, Vol. 18, No. 13 ISSN 2150-8097.
doi:10.14778/3773731.3773743



(a) Full DPU offload **(b) Adaptive DPU offload**

Figure 2: Task placement examples between Host and DPU.

DPUs, the observability platform can offload (de)compression to the card while the digital twins platform can perform string filtering (similar to a LIKE predicate) directly on incoming data; thus the applications can reduce data volume and operational costs since they use less CPU time on non-critical tasks ((de)compression or filtering).

While offloading tasks to DPUs improves throughput and reduces network and CPU load, it introduces the challenge of effective task processing between Hosts and DPUs. However, existing research does not offer an adequate foundation to effectively utilize DPUs as either dedicated task offloaders or co-processors. This is essential since failure to account for the real-time characteristics of the combined Host-DPU environment, such as task throughput and resource contention, may negate any gains from near-network processing. For example, consider once more the use cases in Figure 1, where the observability platform decompresses incoming data and recompresses it for storage. If the entirety of the (de)compression tasks are offloaded to the DPU, it may saturate the DPU's processing capacity, which results in suboptimal processing throughput, either on the DPU, on the subsequent host CPU operators, or both. Similarly, the digital twins platform decompresses and filters data exclusively on the DPU, but this operator order does not yield optimal throughput for all data types or user queries. Our micro-experiments (Section 3.4) show that DPUs and host CPUs exhibit different performance characteristics depending on factors like large vs. small input buffer size or simple vs. complex filtering queries, making a static offloading scheme unsuitable in many real-world scenarios. Figure 2 illustrates a bottleneck where DPU performance constraints introduce a straggler operator early in the processing pipeline, slowing down all subsequent operations. While in the full DPU offload (2a) the host's CPU is barely involved outside of performing aggregations, the combined Host-DPU offload (2b) achieves higher performance by co-processing data and routing to more capable operators in the DPU, when necessary. Therefore, effective task co-processing between DPUs and hosts is essential when leveraging near-network processing, especially in disaggregated environments where network efficiency is crucial when reducing operational and runtime costs.

In this analysis paper, we conduct comprehensive experiments on hardware acceleration and co-processing on DPUs for data-related tasks. Specifically, we measure how factors such as data volume, query complexity, and DPU configuration affect performance when hardware accelerated tasks are offloaded partially or fully. We compare DPU-accelerated executions against both CPU-only and SIMD-accelerated baselines to provide a broader context regarding DPU performance. Our findings reveal that partial offloads can deliver similar throughput to DPU accelerator setups while outperforming CPU or SIMD implementations by orders of magnitude, only if the overhead of reconfiguring the DPU is carefully managed. To our knowledge, our work is the *first to analyze the co-processing potential of BlueField-class DPUs* on data-related tasks, namely compression, decompression, and string filtering, by sharing their hardware accelerators as well as their ARM cores with the Host. By identifying bottlenecks and trade-offs, our work offers practical guidance for data system practitioners looking to leverage or integrate DPUs and their accelerators in their systems.

In this paper, after motivating the need for co-processing in near-network systems in Section 2, we contribute the following:

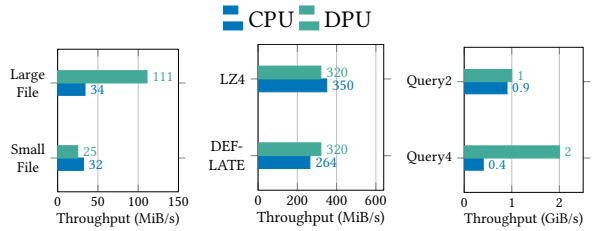
- We introduce essential background concepts, formalization, and constraints for the combined Host-DPU systems with multiple hardware accelerators in Section 3.
- We present the design of a benchmarking suite that supports both full and partial offloading, together with a set of co-processing strategies, in Section 4.
- We conduct an extensive benchmarking analysis on hardware accelerated tasks by using CPU-only, DPU-only, Host-DPU, and SIMD-optimized implementations in Section 5.
- Building on our benchmarking takeaways, we identify a set of key findings on using DPUs and discuss future performance expectations at the end of Section 5.
- We gather and discuss related work for Host-DPU data management in Section 6.

Finally, we provide our concluding remarks in Section 7.

2 THE CASE FOR EVALUATING NEAR-NETWORK CO-PROCESSING

In this section, we discuss the motivation behind evaluating near-network co-processing of tasks with DPUs. We highlight outcomes from related work and our own benchmarks, both indicating that a static cost model is inflexible and that DPUs can offer additional performance benefits. These findings motivate a deeper experimental analysis to determine how and when co-processing best exploits DPUs and their hardware accelerators.

In the past, specialized network processing units (NPUs) offered function-specific instruction sets to accelerate common networking tasks at line rate [19]. However, NPU fixed-size pipelines and limited RAM force an inflexible, NPU-only processing model without any possible fallback to the host [2]. To allow flexible and fast transfers with host interplay, DPUs, namely NVIDIA BlueField-class SmartNICs, use a combination of ARM CPUs, PCIe switches, and



(a) Compression **(b) Decompression** **(c) Regex**

Figure 3: Task throughput for different input types

Table 1: Symbols used throughout the paper.

Symbol	Meaning
q	user query in domain-specific language
QO	representation of a query operator
$\{k\}$	set of operators QO that are in q
I	QO's input queue of (one or more) $\{\tau\}$
O	QO's output queue of (one or more) $\{\tau\}$
f	operator function, reads τ from I, writes τ' to O
TT	the QO/task's throughput (data volume over time)
OT	accumulated throughput of all $QO \in q$

DMA engines. This flexibility creates co-processing opportunities, as data processing may happen partly or fully on either the Host or the DPU. As a result, DPUs are increasingly adopted in both industrial use-cases such as virtualization [65], programmable data infrastructure [59], and disaggregated storage [46], as well as in recent academic research [22, 25, 61, 70]. DPUs process data near the network stack, reducing data movement before standard data processing tasks. Current state-of-the-art on near-network data reduction falls into three categories: using SmartNICs for RDMA or as a host-bypass node, using non-DPU hardware for in-network processing, and using non-networking hardware before data movement occurs. First, RDMA approaches enable high-throughput remote memory access but put the burden of managing workload skew adaptation on the application [24, 62]. Second, host-bypass approaches offload tasks directly to specialized hardware yet require careful consideration of resource limits and workload characteristics [61, 70]. Third, in-network processing uses programmable switches or ASICs for flow-level computation but introduces a separate programming model, has limited on-switch resources, and increases per-task latency since it happens on a separate physical host at least one network hop away [25, 34]. Lastly, on-path specialized hardware, such as FPGA accelerators, reduces data volume during query processing but requires careful integration through scheduling or "middleware" solutions like AWS AQUA [6]. Taken together, these lines of work focus only on static or manual decisions about when and how to offload work to DPUs. This leaves a research gap in exploiting the co-processing capabilities of joint Host-DPU systems, where the system makes real-time decisions on how and where to handle a given task while maintaining performance guarantees. We discuss related work in detail in Section 6.

Our initial experiments build on related work insights and shortcomings by benchmarking multiple Host-DPU tasks, demonstrating that performance varies based on data type and query-specific factors. Figure 3 shows how identical tasks can yield different throughput results on the Host CPU versus the DPU, depending on the input data (Figure 3a), on task configuration (Figure 3b), or query-imposed configuration (Figure 3c). We focus on DPU hardware-accelerated tasks that are important for data systems, namely (de)compression and regex. Disaggregated data systems (e.g., Delta Lake or Iceberg) store data in compressed columnar formats (e.g., Parquet or ORC); thus compression/decompression is necessary to execute queries [40] while data systems like ClickHouse or DuckDB use regex engines to perform early-stage filtering [18]. Offloading these operations to a DPU relieves the CPU during concurrent query execution. We use the Silesia compression corpus [17] for compression/decompression and a text-based corpus [48] for string matching. The Silesia corpus contains diverse file types (text, binaries, images) to test compression and decompression, whereas the text-based corpus features multiple query patterns (prefix, suffix, wildcard) for string matching. We observe that there is no consistent winner across all tasks or queries, reflecting how the underlying hardware implementations and memory layouts shape performance, particularly in shared-memory interactions between Host and DPU. This underlines the shortcomings of static cost models and motivates real-time reactive co-processing strategies.

In summary, both related work and our initial micro-benchmarks highlight the need for evaluating co-processing on Host-DPUs on static or manual offloading decisions. By sharing tasks between the host and the DPU, systems can capitalize on near-network capabilities without incurring suboptimal placements or excessive data movement overhead. We delve deeper into these issues through our experimental analysis of Host-DPU co-processing in Section 5.

3 BACKGROUND AND PRELIMINARIES

In this section, we first discuss the necessary background about SmartNICs and their hardware characteristics in Section 3.1. Then, we introduce the set of operators that SmartNIC hardware can accelerate, which are our targets for this paper in Section 3.2. We then detail the necessary formalization for modelling operators and data tasks between Host and DPUs, similar to general task placement for data processing in Section 3.3. Finally, we introduce a set of constraints that a system would require in order to guarantee performance from a Host-DPU setup in Section 3.4. Table 1 contains the symbols used throughout this paper.

3.1 DPU Hardware

At their core, SmartNICs are traditional network cards (NICs) with additional subsystems that enable data processing near the physical hardware of the network stack. Figure 4 shows the high-level details of the components of the BlueField platform, which is generalizable to any SmartNIC. Namely, the platform consists of four subsystems: **(i)** CPU, **(ii)** NIC, **(iii)** PCIe switch, and **(iv)** Data Acceleration. The CPU subsystem **(i)** adopts an energy-efficient design, e.g., Armv8 or MIPS architecture, with increasing core counts and multiple levels of L2 and L3 cache. The CPU subsystem boots a full OS, from auxiliary eMMC storage and has its own DDR4/5 RAM. The CPU enables easier access to the other subsystems since its purpose is to sit in the control path and not to perform critical processing [11]. The

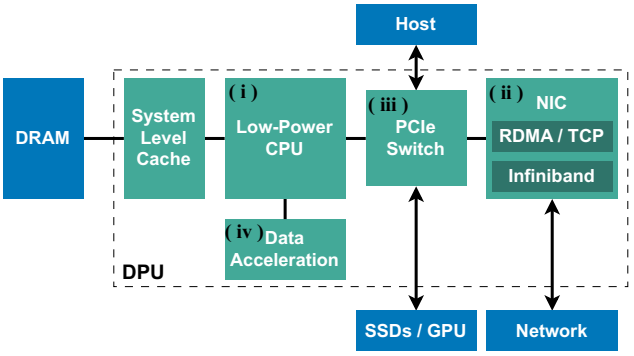


Figure 4: BlueField v3 architecture

NIC subsystem **(ii)** is a high-performance network stack (100 – 400 Gb/s) with line-rate data processing capabilities (e.g., deep packet filtering, encryption, RDMA, and Infiniband). Having a CPU near the NIC enables a logical "bump-in-the-wire" [1] design, where the NIC CPU can transparently intercept and process live data before forwarding to the Host, significantly reducing latency. The PCIe switch **(iii)** enables interconnect of the various subsystems, either with the Host system that physically hosts the DPU card as well as other subsystems, such as Host-attached storage (via NVMe-over-Ethernet), or GPUs (`GPUDirect` in NVIDIA platforms). The PCIe switch complements the "bump-in-the-wire" design by enabling flexible application logic placement with minimal overhead: applications can run entirely on the host CPU, on the DPU, or partially between the two. Finally, the Data Acceleration subsystem **(iv)** contains hardware accelerators that perform post-processing of networked or on-Host data, reducing the burden of performing common data tasks from the Host CPU (e.g., (de)compression, Regex Matching, or SSL encryption).

3.2 Hardware Acceleration

The hardware acceleration (or Data Acceleration) subsystem of a DPU is a set of high performance FPGAs or ASICs dedicated to specific tasks, with different capabilities across DPU generations or vendors. For example, BlueField DPUs integrate a specialized chip to perform line-rate Regex analysis (called RXP) [51]. To expose these capabilities to end users, vendors provide APIs that hide the communication complexity with the Data Acceleration subsystem through task-based queues, available from both the DPU's CPU subsystem and the Host [49, 50]. Through these APIs, systems are able to offload tasks on the Data Acceleration hardware and take advantage of higher throughput, better energy efficiency, and early reduction of data. In this paper, we refer to hardware acceleration when a system or user makes use of the Data Acceleration subsystem explicitly and avoids the use of the DPU's CPU subsystem. The ARM CPU has a coordination focus; thus performance-oriented tasks should be offloaded to the Data Acceleration subsystem. Specifically, we use the BlueField platform (v2 and v3) to hardware accelerate decompression, regex, and compression tasks.

[1] We use "bump-in-the-wire" to refer to augmenting NIC functionality by executing application logic on the DPU, rather than just hardware-only acceleration.

3.3 Data and Operator Models

In this paper, we assume the presence of a dataflow model [3], where queries combine relational operators to form pipelines across potentially distributed hosts. This setup aligns with our focus on hardware acceleration via DPUs for data processing.

Data Model. We assume a dataflow perspective in which records (or tuples) flow through the system. Each record comprises a set of typed attributes, such as integers, floats, or strings, and may originate from a bounded or unbounded source. This design enables a uniform approach to both batch and streaming scenarios, without strict reliance on timestamps or unbounded queues.

Operator Model. A query q consists of k query operators QO. Each QO reads records from one or more upstream operators and writes its output to downstream operators. Internally, each operator applies a function f to incoming records, producing zero or more output records. These functions range from simple filters to more complex transformations (e.g., joins or aggregations). Each operator may leverage hardware acceleration from a capable DPU.

3.4 DPU Tasks and Constraints

In this paper, we evaluate a set of data management tasks on the hardware accelerators of a BlueField DPU. Specifically, we focus on compression, decompression, and string comparison tasks and analyze how hardware acceleration affects them. We also examine how these tasks can utilize both the host CPU and the DPU for specific QOs that compose a single query. Our analysis focuses on two metrics: individual task throughput (TT) and overall co-processed operator throughput (OT). This perspective clarifies whether a task is better placed on one target (CPU or DPU) or split across both. The DPU's bump-in-the-wire design intercepts network traffic on the SmartNIC's Linux OS, creating a trade-off between TT and OT. Our primary goal is to determine when *using a DPU is better than using a Host CPU-only design*, either by fully offloading tasks to the DPU or by co-processing them. Accordingly, we define two performance-related constraints:

- **C1:** Tasks will use the DPU iff its TT is better than the Host CPU's TT.
- **C2:** Tasks can use both Host CPU and DPU to improve OT only if OT is better than the individual TTs.

These constraints capture typical system requirements, i.e, no offloading if it fails to improve throughput, and guide our evaluation on co-processing. The constraints are necessary in avoiding degraded performance, especially when it is not clear a-priori if the DPU outperforms the Host or if it reduces OT.

4 BENCHMARK DESIGN AND TASK PLACEMENT CONSIDERATIONS

DPUs contain hardware accelerators that can significantly reduce CPU load by processing ingested data with low latency. However, misconfigurations can introduce performance bottlenecks, negating their benefits. Accurately characterizing DPU throughput under various workloads and scenarios is thus essential for effective offloading. In this section, we present our benchmark design in Section 4.1 and then outline several task placement policies that guide potentially efficient DPU utilization in Section 4.2.

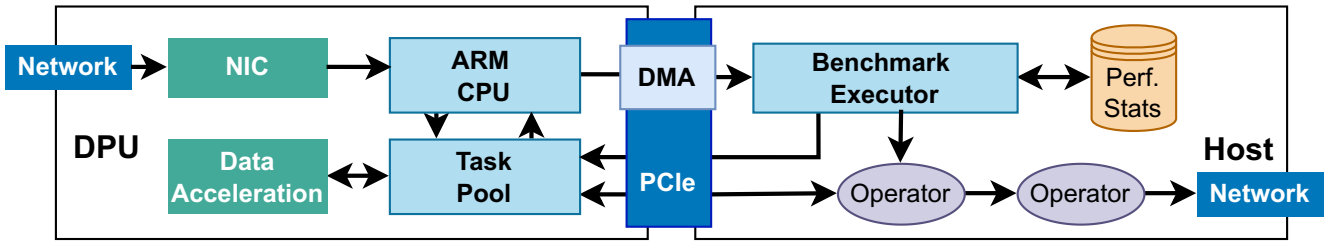


Figure 5: The overall design employed by our benchmarks across DPU and Host.

4.1 Benchmark Design

Benchmarking near-network hardware requires a flexible design that integrates DPUs in data processing pipelines in two ways: (1) as a dedicated offload processor, where tasks are fully delegated to the DPU, and (2) as a co-processor, where the Host CPU and the DPU share the workload. These approaches adhere to the two key constraints from Section 3.4: **(C1)** offload a task entirely if the DPU's throughput exceeds the CPU, and **(C2)** co-process if combined Host-DPU throughput outperforms the CPU. Our benchmarking framework, illustrated in Figure 5, employs an executor to coordinate and measure performance under varying workload configurations, from 0% (CPU-only) to 100% (DPU-only). A micro-benchmark in this framework consists of a single, hardware-accelerated task operating on a given dataset or set of data chunks. When a benchmark involves DPU-only execution, the executor submits each micro-benchmark to the DPU subsystem via the ARM CPU; the DPU processes these tasks in isolation and returns metrics directly to the Host. For CPU-only benchmarks, the executor schedules tasks on the CPU and records metrics in-situ. Finally, our design also accommodates coarse-grained co-processing: the executor divides the input into proportionally sized chunks for both the DPU and CPU, then merges results on the host via the DPU's DMA capability.

4.2 Task Placement Policies

In this section, we present the possible task placement policies between Hosts and DPUs. Because DPUs handle network traffic, they can perform task processing earlier than the Host, potentially filtering data early or introduce slowdowns if misconfigured. We first establish a set of static placement policies that serve as baselines for Host-DPU systems. We then introduce co-processing placement strategies, which we later use to evaluate the combined Host-DPU system's throughput on different tasks in Section 5.

Static Placement. We first consider the full placement of tasks on the Host CPU (*fCPU*), where the DPU is not involved in operator processing. From here, we explore the static placement of specific tasks on the DPU, specifically decompression on DPU (*dDPU*), compression on DPU (*cDPU*), and regex on DPU (*rDPU*). Through these, we can evaluate the throughput of individual operators compared to the Host without any communication overhead.

Co-Processing Placement. We consider a second set of placement strategies that involve the Host CPU offloading tasks to the DPU. In this scenario, the Host CPU has access to the data and can offload a percentage to the DPU. In a full system, this decision would be made preemptively to check if **C1** and **C2** are satisfied. If either **C1** or **C2** hold, then the Host CPU can switch to the following co-processing policies: co-processed compression (*cCOMP*), co-processed decompression (*cDECOMP*), and co-processed regex filtering (*cREG*). Through these, a Host-DPU system can evaluate whether partial offload is beneficial or if the entire operator should be run on DPUs without co-processing.

Overall, the placement policies cover all possible DPU task placements. They allow us to evaluate the individual accelerator performance (*TT*) together with a co-processed variant from Host and DPU (*OT*). These performance counters together with the Section 3.4 constraints (**C1**, **C2**) supply all the information that a Host-side feedback controller needs to realize automated co-processing through eddy-style operators [7]. At run time, the Host submits a probe task instance to a DPU, the DPU executes it and the Host receives the task's *TT* or *OT* metrics. Through **C1** or **C2** the Host decides to co-process or revert to Host execution. Hence, our benchmark design and placement policies constitute a minimal foundation for future fully automated Host-DPU systems.

5 EVALUATION

In this section, we evaluate the performance characteristics of Host-DPU systems with micro- and macro-benchmarks involving real-world data sets. We first discuss our hardware setup with multiple Host-DPU configurations in Section 5.1. We then follow up with the details of our implementation and various baselines in Section 5.2. We then list the datasets workloads used across our evaluation section in Section 5.3 followed by a breakdown of the DPU's task runtime in Section 5.4. Initially, we analyze directional DMA performance in Section 5.5. After that, we perform a set of benchmarks that evaluate the performance of hardware accelerators on multiple DPU models for compression in Section 5.6. We move on to benchmarking decompression in Section 5.7. We also benchmark the performance of the regex hardware accelerators in Section 5.8. Next, we examine the benefits of co-processing in Section 5.9 as well as its effect on data routing in Section 5.10, followed by the key insights of our experimental analysis in Section 5.11.

5.1 Hardware Setup

For our evaluation, we utilize two types of Host-DPU systems, namely an EPYC - BlueField 2 (from now on *BF2*) and an EPYC - BlueField 3 (from now on *BF3*). The BF2 Host is an AMD EPYC 7282 16-Core CPU with 128GB RAM and 4TB SSDs. The BF2 DPU is a BlueField 2 MBF2H332A-AENO, with 8× A72 Armv8 CPU cores, 16GB RAM, and 32GB eMMC storage. The BF3 Host is an AMD

Table 2: DPU hardware details and comparison.

Metric \ DPU	BlueField 2	BlueField 3
CPU	8× Armv8 A72 (2.5 GHz)	16× Armv8.2+ A78 (3.0 GHz)
Memory	16 GB (3.2 GHz DDR4)	32 GB (5.2 GHz DDR5)
Network	2× 100 Gb/s	400 Gb/s
Interconnect	PCIe Gen 4.0	PCIe Gen 5.0
Power Envelope	75 W (PCIe)	150 W (PCIe + PSU)

EPYC 9124 16-Core CPU with 1.8TB RAM and 4TB SSDs. The BF3 DPU is a BlueField 3, with 16× A78 Hercules Armv8.2+ cores, 32GB RAM, and 32GB eMMC storage. Table 2 compares the hardware details between the two DPUs while Table 3 summarizes the available hardware accelerators on each card between the two DPU models. NVIDIA provides access to BlueField devices through DOCA[2], a C-based software framework. We found that the Regex engine's development headers are not present in newer versions of DOCA (post v2.2.1) for both cards; thus this is the reason why we consider it as *partially* supported. We downgraded the DOCA installation on the BF2 DPUs in order to evaluate Regex performance separately but newer DPU firmware versions make downgrades impossible. All systems use Ubuntu 20.04.2 LTS (kernel 5.4.0-26) for their Host, gcc-11, Python 3.9, and DOCA 2.9.0 LTS on hosts and DPUs.

5.2 Implementation and Baselines

Our task implementation focuses on hardware accelerated tasks present across all DPUs from Table 3 and offer a CPU-only implementation equivalent. We use a C++ `zlib` (v1.3.1) implementation for the DEFLATE compression and decompression CPU-only tasks and a C++ `lz4` (v1.10.0) [14] implementation for the LZ4 decompression equivalent. We use TurboBench to include SIMD-based (de)compression from `libdeflate` (1.23) [10, 42]. For the Regex task we use two C++ regex libraries popular in data management systems, namely `re2` (v2024-07-02), and `hyperscan` (v5.4) [18, 23].

Our microbenchmarks utilize Host CPUs and DPUs in separation in addition to the DPU's ARM and accelerators. This allows for controlled performance testing with and without memory allocation interference from DOCA's memory model [55]. Our *static setting* emulates workloads with predictable buffer sizes (e.g., fixed-size Parquet row groups), suitable for batch processing. Inversely, our *reconfiguration setting* reflects workloads with variable buffer sizes (e.g., diverse sources, schema evolution, or multi-tenant systems), where output buffer sizes cannot be determined in advance. The trade-offs between static and reconfiguration settings vary across application domains. In networking workloads (packet parsing or filtering), packet sizes are well-defined and known in advance; thus static buffer sizes make sense. Inversely, data systems frequently operate on compressed or variably sized records [40, 68]; thus reconfiguration is unavoidable. We discuss these trade-offs in Section 5.4.

For our co-processing experiments, we utilize both Host CPUs and DPUs with the main DOCA program running on the Host or the ARM CPU. To emulate the windowing effect while measuring throughput, we spawn two separate C++ threads. The first thread performs a CPU-only task while the other submits an equivalent

[2]https://developer.nvidia.com/networking/doca

Table 3: Hardware data accelerator matrix across DPUs.

DPU \ Feature	Compress (DEFLATE)	Decompress (DEFLATE)	Decompress (LZ4)	REGEX
BlueField 2	✓	✓	×	Partial
BlueField 3	×	✓	✓	Partial

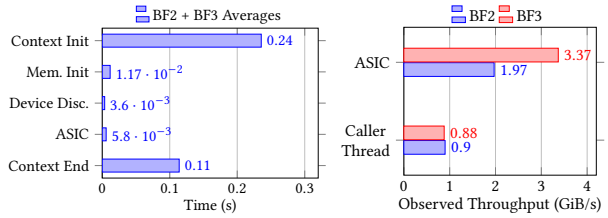


Figure 6: End-to-End lifetime breakdown of a DOCA task.

DOCA task to the DPU. We synchronize the threads using two thread barriers. The first barrier waits for memory initialization and data loading for both threads to finish while the second barrier waits for task processing. Logging is disabled and results are persisted to memory. This allows raw task performance evaluation without serialization overhead from system-specific integration while still accounting for co-processing latency, where the system waits for the slower accelerator before moving to the next pipeline stage.

5.3 Datasets and Workloads

We utilize a set of different datasets tailored for the task at hand. Namely, for (de)compression tasks we evaluate the accelerators using the *Silesia* corpus [17] from the lossless compression field. The corpus includes 12 files with a variety of properties that help assess the effectiveness of (de)compression implementations. The file types range from free-form English, to source code, trace logs, executables, `tar`ed directories, database files, PDFs, and image blobs. The average file size is 17.5 MiB, with raw file sizes ranging from 5.3 MiB to 51.2 MiB. The (de)compression task workload is to benchmark each accelerator and compare their performance against CPU-only and a SIMD implementations. Silesia's diverse file set mirrors workloads in modern data platforms, such as Lakehouse [5] or DNN training [47], which routinely handle structured and unstructured data types. Using real world data provides better compression performance than random input, which lacks data redundancy [56].

For Regex, we use the open source *US-Accidents* dataset [48]. The dataset contains car accident logs from 49 USA states, collected from February 2016 to March 2023 [69]. The data contains 7.7 million rows of accident descriptions with standardized terminology in the *description* column. This makes the content easy to filter using a set of regexes, namely *Q1-4* in Table 4. The Regex workload is to benchmark the hardware accelerator on *Q1-4*, which contains the use of multiple different prefix, suffix and wildcard markers.

For our co-processing benchmarks, we re-use the setup from our microbenchmarks, but we split the data to be processed according to a current offload percentage. This helps us compare various co-processing placements against their static allocation counterparts.

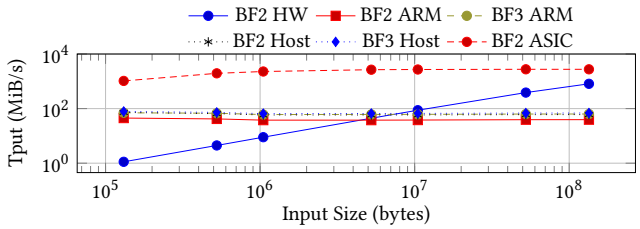
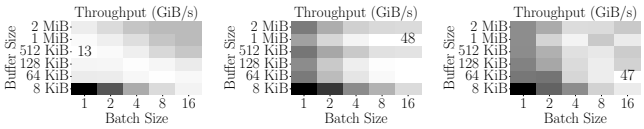


Figure 7: DPU Compression on DEFLATE.



(a) Host to BF2 **(b) Host to BF3** **(c) BF3 to Host**

Figure 8: DMA performance.

5.4 DOCA Task Breakdown

When users submit DOCA tasks from a caller thread, the task runtime goes through multiple stages. Figure 6 shows the average per-stage timing and throughput for all DOCA tasks, including only DOCA-related times. The sum is the total that the caller thread observes. *Context Init* initializes `DOCA Context` and `DOCA Progress Engine` (both represent one logical application thread) and accounts for 62% of the total runtime. While this represents the majority of the runtime, it is avoidable by re-using the caller thread; thus subsequent measurements in our evaluation exclude it. *Mem. Init* prepares and initializes DOCA memory areas aligned with the CPU while *Device Disc.* locates a DOCA device with `Compress` capabilities (local or over PCIe). These are also excluded from later measurements. *ASIC* is the hardware accelerator's data processing time, measured via state callbacks. BF2's ASIC tasks average 1.97 GiB/s throughput while BF3's average 3.37 GiB/s. Finally, *Context End* is the time for the caller thread to stop the current `DOCA Context` and re-configure. This stage includes the delay from the last callback until the Context is back to an `idle` state and is 19.4× larger on average than the ASIC time. Despite improved ASIC performance, caller thread throughput is similar between BF2 and BF3 if Context reconfiguration is required. In our experiments, we report both raw ASIC throughput and caller-observed throughput. Raw ASIC throughput reflects the ideal case, in which the Context does not require reconfiguration (static setting). Caller-observed throughput reflects frequent Context resets (reconfiguration setting).

5.5 DMA Performance

This experiment answers *what is the DMA performance between Host and DPU, across multiple configurations?* DMA is implicit in DOCA tasks when deciding where the input or output buffers should reside (Host-to-DPU or DPU-to-Host). Figure 8 shows DMA throughput heatmaps for Host-to-DPU transfers on BF2 (Figure 8a) and BF3 (Figure 8b), as well as DPU-to-Host DMA from BF3 (Figure 8c). We vary three parameters: *buffer size*, *per-thread task batch size*, and *initiator CPU core count*. Lighter colors indicate higher throughput;

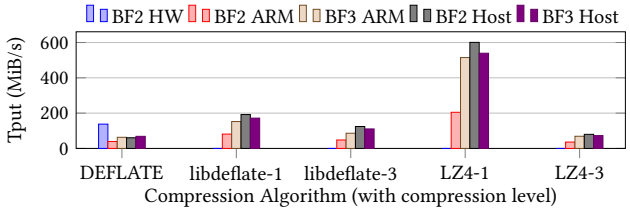


Figure 9: DPU Compression against other algorithms.

we show the buffer vs batch size configuration since it achieves the best throughput and discuss optimal core count in text due to lack of space. BF2 Host achieves peak throughput of 13 GiB/s with 512 KiB buffers, 2 host cores, and 1 per-thread batch size. In contrast, BF3 Host reaches 48 GiB/s using 1 MiB buffer size, 4 cores and 16 task batches, a 3.7× improvement. These results reflect architectural advances in BF3 DMA engine, including PCIe Gen5 support and deeper DMA queue pipelines. The BF2 Host, despite larger buffer size support (up to 128 MiB, omitted for brevity), fails to saturate its slower interconnect while requiring OS-level large-page support on BF2. To assess symmetry, we include the reverse configuration (DPU-to-Host from BF3), which peaks at 47 GiB/s using 64 KiB buffers, 16 ARM cores, and 16 task batches. BF3 ARM's throughput when initiating DMA to its Host performs similarly to its Host due a different configuration that exploits its topology maximally.

Findings: Host-initiated DMA on BF3 almost saturates its interconnect using larger buffers but similar task batches to the DPU-initiated DMA. ARM-initiated DMA achieves slightly lower throughput due to weaker ARM cores. Overall, optimal DMA configuration depends on the initiator's resource characteristics.

5.6 Compression

In this set of experiments, we answer *what is the performance of compression across the multiple accelerators in the Host-DPU system?* We cover the *fCPU* and *cDPU* placements, specific to DEFLATE compression. To do so, we first compare the performance of BF2's Compression ASIC (*BF2 ASIC*) as well as the performance as observed from the caller thread (*BF2 HW*), against the ARM CPUs on both DPU models (*BF2 ARM*, *BF3 ARM*) and their hosts (*BF2 Host*, *BF3 Host*). We then compare the performance of other algorithms and compression levels on the same hardware.

DEFLATE performance: Figure 7 shows the throughput of *BF2 HW* and *BF2 ASIC* on the DEFLATE compression scheme. We compare them against the Host CPUs and BF3. CPU results show the average of DEFLATE levels 1-3. We observe that *BF2 ASIC* (1.03 – 2.75 GiB/s) consistently outperforms other hardware since it uses fixed-size buffers throughout the experiment. This avoids the reconfiguration penalty of the DOCA runtime (Section 5.4), and leverages the ASIC's fully pipelined architecture for streaming input. By contrast, *BF2 HW* shows significantly lower throughput (1.1 – 44 MiB/s), especially on small inputs (< 10 MiB), where it is outperformed by CPUs (37 – 78 MiB/s) by 1.3 – 78×. This is primarily due to the high cost of buffer reconfiguration: *BF2 HW* uses one DOCA context per task, and must reallocate buffers on each task submission, leading to poor pipeline utilization for small and variable-size input. These limitations are in line with

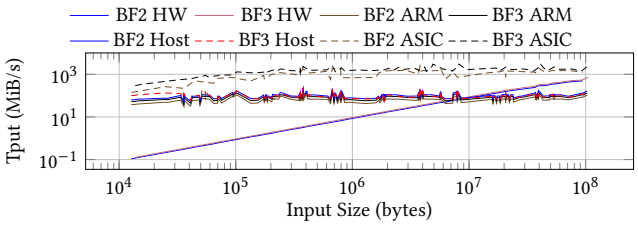


Figure 10: DPU Decompression on DEFLATE.

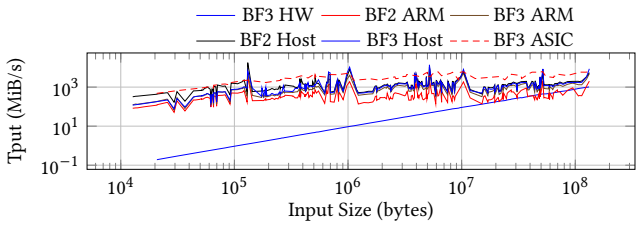


Figure 11: DPU Decompression on LZ4.

the caller thread throughput from Section 5.4. Despite this, *BF2 HW* outperforms CPUs on 11 out of 12 files from the corpus, with per-file throughput ranging between 45 – 378 MiB/s (vs. 23 – 111 MiB/s for CPUs), thanks to amortization at larger sizes.

DEFLATE vs Others: Figure 9 shows results of *BF2 HW* against other compression algorithms. We benchmark `libdeflate`, which uses SIMD instructions on CPUs and is level-compatible with raw DEFLATE and LZ4 that favors throughput over size reduction [14]. We omit size reduction results due to lack of space. LZ4 nears 42 – 48% size reduction while the DEFLATE family of algorithms nears 55 – 60% reduction. `libdeflate` brings CPU performance closer or better than *BF2 HW* DEFLATE (132 MiB/s) while offering similar reduction. Specifically, *BF3 Arm*, *BF2 Host*, and *BF3 Host* outperform *BF2 HW* and their performance increases from 60 – 68 MiB/s to 151 – 192 MiB/s due to vectorization. In LZ4-1 the performance jump is larger and BF2's ARM (205 MiB/s) outperforms *BF2 HW*. The rest of the CPUs perform similarly with each other and all of them outperform *BF2 HW* between 3.2 – 4.5× in LZ4-1, with *BF2 Host* reaching 599 MiB/s. Higher compression levels (`libdeflate`-3, LZ4-3) perform worse than *BF2 HW* but further reduce file size while performing similar or better to DEFLATE (between 1.05 – 2×).

Findings: This set of experiment explores compression trade-offs on Host-DPU systems. *BF2 HW* outclasses CPUs for DEFLATE, despite DOCA's per-task reconfiguration overhead. However, there are cases where it performs worse (5 MiB `xml` file) or is outperformed by its own ARM core under vectorized implementations (`libdeflate` or LZ4). These cases demonstrate that the *BF2 HW* pipeline is sensitive to buffer reconfiguration and requires careful planning in order to compete with high-performance CPU libraries. Additionally, we observe that the newer BF3 ARM is on-par with server-grade Host CPUs. Moreover, BF3 ARM's LZ4 throughput compensates for the lack of a dedicated BF3 Compress ASIC. Overall, while *BF2 HW* outperforms CPUs in DEFLATE, careful orchestration is needed to avoid degradation on small or irregular inputs.

5.7 Decompression

In this set of experiments, we answer *what is the performance of decompression tasks in a Host-DPU system?* With these experiments we cover the *fCPU* and *dDPU* placements, for two specific decompression tasks. We examine BF2 and BF3's Decompression ASICs, which support a different number of algorithms, recall Table 3. BF2 (*BF2 ASIC*) supports only DEFLATE-based decompression while BF3 (*BF3 ASIC*) additionally supports LZ4 decompression. Initially, we examine the performance of DEFLATE decompression, where we compare *BF2* and *BF3 ASIC* against their caller thread-observed performance (*BF2 HW* and *BF3 HW*) as well as their ARM CPUs (*BF2 ARM*, *BF3 ARM*) and the CPUs present on their Hosts (*BF2 Host*, *BF3 Host*). After DEFLATE, we examine the same set of hardware accelerators minus *BF2 ASIC* and *BF2 HW* on LZ4 decompression, since BF2 lacks a dedicated LZ4 hardware decompressor. Similarly to compression, we additionally examine other decompression algorithms on the same hardware. Due to BF3 lacking ASIC compression in BF3, we prepare input as raw DEFLATE frames using `zlib` and raw LZ4 blocks. All results are computed locally on the ARM CPU of each DPU and only contain DOCA related task times without context and memory initialization.

DEFLATE Performance: Figure 10 depicts the performance of the various hardware accelerators on DEFLATE decompression. Since compressed input can not be constrained to specific sizes we include multiple different points for each line in the plot, which do not necessarily overlap between accelerators due to the difference in compression ratios. For this reason, we omit individual markers and only show the lines. We observe that overall performance behaves similarly to the DEFLATE compression in BF2 with the main difference being a lower performance cut-off point for *BF2 HW* and *BF3 HW*. Specifically, both *BF2 HW* and *BF3 HW* perform close to the CPUs near the 7 MiB mark, with the CPUs averaging 121 MiB/s and outperforming the DPUs by up to two orders of magnitude at small input sizes (near 100 KiB). This is due to the fact that, as in compression, the DOCA runtime for *BF2 HW* and *BF3 HW* requires task reconfiguration, which incurs a high overhead relative to the time spent processing small buffers. CPU performance starts around 75 – 182 MiB/s, decreases inversely with input size, and plateaus between 47 – 144 MiB/s. For the observed throughput in DPUs, the combined throughput for both *BF2 HW* and *BF3 HW* ranges between 1 – 5 MiB/s for ≤ 100 KiB, 6 – 40 MiB/s for inputs < 7 MiB, and 60 MiB/s – 1.1 GiB/s for ≥ 7 MiB. The two DPUs perform similarly but *BF3 HW* outperforms *BF2 HW* by 34% on average due to the underlying BF3 ASIC performing faster than BF2, recall Section 5.4. Finally, raw ASIC performance still outperforms the CPUs by 1.85 – 16×, where *BF2 ASIC* manages 140 MiB/s – 1.9 GiB/s while *BF3 ASIC* manages 292 MiB/s – 3.1 GiB/s, also a near 34% improvement from BF2 to BF3. In this benchmark, *BF2 HW* and *BF3 HW* outperform the CPUs only in 2 out of 12 files, with observed per-file throughput for *BF2 HW* and *BF3 HW* ranging between 36 – 124 MiB/s while for the CPUs between 43 – 117 MiB/s. This is due to 8 files out of 12 having sizes close to $7MiB$, below which DOCA reconfiguration overhead dominates BF2/BF3 performance.

LZ4 Performance: Figure 11 shows LZ4 decompression performance. We again omit individual markers. BF2 lacks LZ4 hardware

support (recall Table 3), thus we omit BF2 HW from Figure 11. We observe that average throughput increases for all the involved hardware accelerators, with the CPUs reaching similar performance to the ASIC for the first time. This is due to LZ4's throughput focus, which maps efficiently to SIMD instructions and saturates memory bandwidth even on lower-end CPUs [14]. Moreover, we observe that while *BF3 HW* behaves similarly to its DEFLATE results, the CPUs outperform it by two orders of magnitude across most input sizes. This is due to two compounding factors: (1) LZ4's high decompression throughput amplifies any orchestration overhead, and (2) BF3's reduced per-task buffer limit (2 MiB vs. 128 MiB in BF2) forces *BF3 HW* to issue multiple DOCA tasks even for small files— e.g., `xml` of 5 MiB, requires three separate task submissions. As a result, *BF3 HW* is on-par with the slowest CPU (BF2 ARM) at $\geq$ 20 MiB input, and is outperformed by all others. CPU performance ranges from 60 MiB/s to 7.2 GiB/s while the raw *BF3 ASIC* performance ranges between 464 MiB/s to 6.3 GiB/s. On average, *BF3 ASIC* (5.2 GiB/s) outperforms the high performance CPUs (Hosts and BF3 ARM at 2.7 GiB/s) by 1.9×. This advantage stems from the ASIC's benefiting from static buffer sizes, thus no DOCA reconfiguration. In this experiment, the CPUs outperform *BF3 HW* on all files. This is expected, as LZ4 prioritizes speed over compression ratio and is designed to scale on general-purpose CPUs [14]. In contrast, *BF3 HW* pays the full DOCA reconfiguration cost for each task, which worsens performance in a high-throughput workload like LZ4.

DEFLATE and LZ4 Against Others: Figure 12 compares DEFLATE *BF2 HW* and *BF3 HW* against each other and `libdeflate` as well as `LZ4`. For DEFLATE, *BF2* and *BF3 ASIC* (1.2 – 1.8 GiB/s) outperform all CPUs (66 – 105 MiB/s) by almost two orders of magnitude. However, reconfiguration-prone *BF2 HW* and *BF3 HW* (50 and 72 MiB/s) underperform even against BF2's ARM CPU. By switching to SIMD-optimized `libdeflate` all CPUs gain a larger performance gap against the DPUs, with *BF2 ARM* reaching 137 MiB/s while the rest perform similarly (*BF3 ARM* 215 MiB/s, *BF2 Host* 260 MiB/s, and *BF3 Host* 236 MiB/s). This yields a 2 – 2.5× improvement over *BF2 HW* and *BF3 HW* on DEFLATE. For LZ4, we observe a wider performance gap. While *BF3 ASIC* still delivers the highest throughput (4.1 GiB/s), *BF3 HW* remains bottlenecked at just 92 MiB/s, more than an order of magnitude slower that *BF3 ARM* and Hosts (between 1.2 – 1.6 GiB/s). This highlights how orchestration overhead and small buffer sizes disproportionately impact high-throughput codecs like LZ4 if reconfiguration is needed.

Findings: This set of experiment explores decompression on Host-DPU systems, this time across multiple hardware-accelerated algorithms. Similar to compression, input size plays a crucial role in *BF2 HW* and *BF3 HW* performance. Unlike compression experiments, DOCA reconfiguration overhead exacerbates DPU performance and while the raw ASICs still outperform the CPUs, observed DPU HW throughput improves only at larger input sizes. Moreover, CPU implementations like `libdeflate` outperform *BF2/BF3 HW* on throughput-focused LZ4. Finally, we again observe the trend of the BF3 ARM having similar performance to server-grade CPUs, both for DEFLATE, LZ4, and vectorized libraries. Overall, decompression experiments further reinforce the need for effective orchestration on Host-DPU systems since there are now more parameters, such as input buffer size and algorithm combinations, to take into account when choosing an appropriate accelerator.

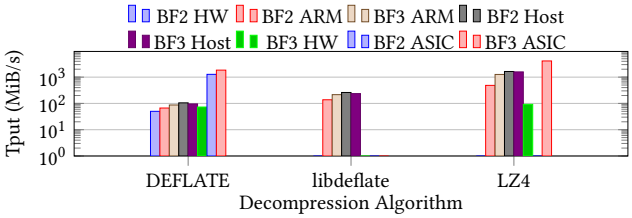


Figure 12: DPU Decompression against other algorithms.

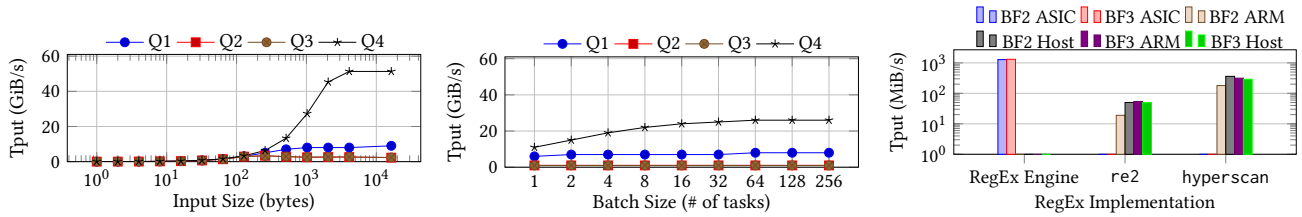
Table 4: Queries from the US Accidents dataset.

Query ID	Regex String
Q1	`At (.+)Exit (.+)`
Q2	`(.+) on (.+) at Exit (.+)`
Q3	`on (.+) at (.+)`
Q4	`Ramp to (.+)`

5.8 RegEx Matching

In this set of experiments, we answer *what is the performance of RegEx tasks in Host-DPU systems?* With these experiments we cover the *fCPU* and *rDPU* placements, specific to the regex tasks. We examine BF2 and BF3's RegEx engine that supports compiled RegEx matching code to run on its ASIC. Recall that we classify RegEx support on the BlueField platform as partial in Table 3, due to the development headers missing from recent DOCA versions. DOCA provides access to the regular expression processor (RXP), a hardware-accelerated engine on DPUs. We use the US Accidents dataset and evaluate performance using all queries. The dataset contains four queries with a variable complexity levels, named $Q1-4$, shown in Table 4. The queries contain different RegEx match groups and need to maintain different intermediate state sizes; thus we initially examine the effect of input buffer size against processing throughput. Moreover, the DOCA RegEx allows task batching; thus we examine the effect of batching in throughput. For both experiments, we compare the per-query performance without separating *BF2* and *BF3 HW* in the findings since performance is the same for both DPUs. We additionally compare RXP's performance against the CPUs, where we make use of `re2` and `hyperscan`. Results are DPU-local and exclude RXP or memory initialization time.

Buffer Size: Figure 13a depicts the performance of RXP under multiple input buffer sizes. From Table 4, Q2 and Q3 (`on (.+) at (.+)`) are similar since the entirety of Q3 is a subset of Q2. We observe this is reflected in their overall similar performance. While Q1 is also present in Q2 after its second match group, Q1 reduces the search space by having a single match group between strings. Finally, Q4 contains only a single match group at the end of the query string, thus it has a small match search space due to the prefix `Ramp to`. Results show that the three query groups behave differently to variable input size. Q4 benefits the most from larger input and reaches almost PCIe line rate (50 GiB/s) due to its large prefix. Moreover, we observe that while Q2/Q3 exhibit peak performance when input size is 256 bytes (3.2 – 3.3 GiB/s), they do not benefit from larger buffer sizes, where performance plateaus. Finally, while Q1 benefits from larger input and reaches its peak



(a) Variable buffer sizes **(b) Variable batch sizes** **(c) Other algorithms**

Figure 13: DPU Regex performance results on US Accidents queries.

performance on 16 KiB input (9 GiB/s), performance stops doubling after 64B input. This is due to DOCA `Mem. Init` step aligning buffers to ARM 64B cache lines pre-processing (recall Section 5.4).

Batch Size: Figure 13b shows RXP performance when DOCA makes use of various batch sizes. We observe that performance improvements are similar to the buffer size results, with Q4 ($11-26$ GiB/s) performing the best due to its simplicity (large prefix, single match group). Q1 ($6-8$ GiB/s) is less affected from the batch sizes and its performance plateaus immediately, which is similar to Q2-3 ($1-2$ GiB/s). This is due to the complexity of Q1-3's patterns, where the RegEx engine's internal matching is likely saturated from the multiple capturing points and state tracking; thus a single query's runtime invalidates batch amortization. Even if the ASIC is not overwhelmed, RegEx matching eventually becomes a bottleneck.

DPU RegEx Engine vs Others: Figure 13c shows the RXP performance againt two other regex engines, `re2` and `hyperscan`. We observe that the RegEx engine's ASICs ($1.2-1.3$ GiB/s) outperform both CPU-based libraries by $1-2$ orders of magnitude. `re2` is slowest ($19-54$ MiB/s), which is expected due to the library's focus on input safety. Finally, `hyperscan` is 3.9× slower ($290-361$ MiB/s) than the ASICs. It is noteworthy that both RXP and hyperscan allow compacting multiple regexes into a single set of match rules, further increasing per-regex throughput. Our results show that per-regex averages remain similar while the runtime reduces N-fold, where N is the number of unique regex queries.

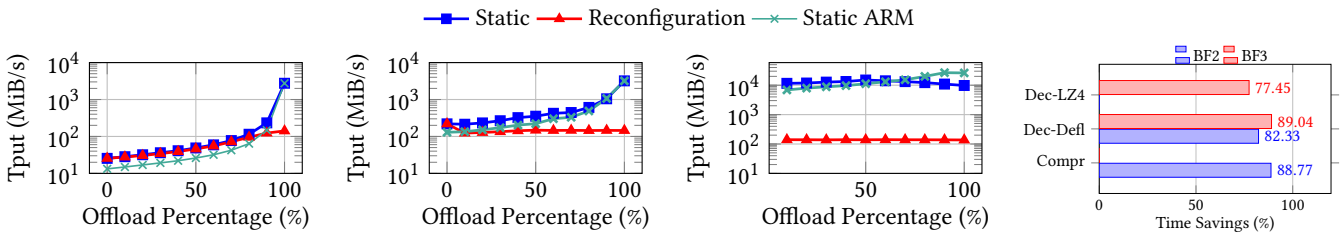
Findings: This set of experiment explores RegEx on Host-DPU systems. The main difference from (de)compression is that even on the same hardware, queries behave differently under different buffer sizes and require re-configuration to perform optimally. Moreover, batching helps improve performance but increases the search space for an optimal configuration. Finally, our CPU-only comparison shows that DPU hardware, if tuned correctly, outperforms SIMD implementations. Overall, RegEx requires orchestration on Host-DPU systems, since performance varies not only across heterogeneous hardware but also across query configurations.

5.9 Task Co-Processing

In this set of experiments we answer *what are the performance implication of co-processing on a Host-DPU system?* We examine gradual task offload from the Host CPU to the DPU accelerators as well as BF2/BF3 ARM CPUs to the DPU accelerators (*ARM*). This allows evaluation of the multiple co-processing task placement policies (*cCOMP* and *cDECOMP*), with 0% being CPU-only ($fCPU$) and 100% full DPU offload ($fDPU$). We omit Regex tasks (*cREG*) due to the lack of development headers, which prohibits the use of this specific accelerator for co-processing analysis. To evaluate throughput, we follow the design for co-processing from Section 5.1. We evaluate two code configurations for the Host, one that re-uses the same `DOCA Context` object (labeled *Static*) and one that needs reconfiguration (labeled *Reconfiguration*) if the input buffer size changes, where the `DOCA Context` would need to stop and re-start. This helps evaluate cases where systems have long-running in-memory workloads against ad-hoc analytical workloads with user queries that requiring multiple different buffer sizes. The Static setting maps the input buffer size to the least amount of buffers possible and never changes for the duration of the experiment, for both Host (*Static*) and ARM CPUs *Static ARM*. In the Reconfiguration setting we change the current buffer size to the total size of the next file to process, so we occupy a single buffer in every task submission. We omit reconfiguration results from ARM CPUs since behavior is similar to the Host. Figure 14 contains all results for co-processing on the hardware-accelerated tasks, which contain only runtime measurements and omit memory preparation or any intermediate result materialization.

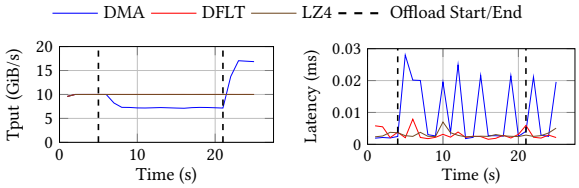
DEFLATE Compression: Figure 14a shows co-processing results from the Host-DPU (*Static/Reconfiguration*) and BF2 CPU-DPU (*Static ARM*) on the DEFLATE Compression task. We report average runtime for each offload percentage. We observe that the combined Host-DPU system outperforms Host-only placement (0% offload) in Static and Reconfiguration settings. Host-only placement starts at 25 MiB/s and the biggest increase is at 70%, with the Host-DPU throughput increasing from 77 MiB/s to 233 MiB/s. Performance almost doubles at 90% (233 MiB/s) and improves by an order of magnitude on the full DPU placement (2.95 GiB/s). For Reconfiguration, we observe that CPU-only placement still benefits from the DPU offload, with the biggest jump in performance at 90% offload (123 MiB/s) and reaches its peak (141 MiB/s) on the full DPU offload. Eventually, Static Host-DPU outperforms the Host-only placement by 116× and Reconfiguration by 20×, which is expected due to Reconfiguration's frequent Context End overhead (recall Section 5.4). On DPU-only *Static ARM* results we observe that using only the ARM CPU and DPU accelerators is slower than Host-DPU by 1.8×, which is inline with the individual accelerator performance from Section 5.6. Finally, Figure 14d contains CPU utilization reduction results and shows that compression co-processing (Compr) benefits the Host due to 88% reduced CPU utilization on average.

DEFLATE Decompression: Figure 14b shows co-processing results from the Host-DPU (*Static/Reconfiguration*) and BF2 and BF3



(a) Compress (BF2) (b) Decompress (BF2, BF3) (c) LZ4 Decomp. (BF3) (d) Host CPU Savings

Figure 14: Co-processing results per task on BF2 and BF3 for Static and Reconfiguration scenarios from Host and ARM.



(a) Throughput (b) Latency

Figure 15: Co-processing impact during data routing.

CPU-DPUs (*Static ARM*) setups on the DEFLATE Decompression task. We observe that Host-only performance starts at 235 MiB/s, the biggest increase is again at 70% offload (610 MiB/s), and full-DPU offload again outperforms other configurations (3.2 GiB/s). In contrast to Compression, Reconfiguration performance worsens when co-processing starts, with throughput dropping on 10% offload (123 MiB/s) and plateaus after 40% (155 MiB/s). This is due to the CPUs performing faster on decompression than compression, hence the DPU with its frequent Context restarts ends up being a straggler. In the Static setting, where no such bottleneck occurs, performance increases as we offload more data to the DPUs, with full-DPU offload outperforming the initial CPU-only by 13× on average. *Static ARM* results are similar to Compression, with ARM-DPU performance being worse than Host-DPU by 1.6× before DPU ASIC processes the majority of the data (≤ 50% offload). Co-processing DEFLATE decompression tasks (Dec-Defl) significantly reduces CPU utilization for the Host (85% on average).

LZ4 Decompression: Figure 14c shows co-processing results on LZ4 decompression tasks from the Host-DPU (*Static/Reconfiguration*) and BF3 CPU-DPUs (*Static ARM*) setups. We observe that for the first time, full-DPU ASIC offload performs worse than CPU setups (Host-DPU, ARM-DPU), with Host-DPU showing peak throughput on 50% (15 GiB/s) and ARM-DPU on 90% (26.9 GiB/s) offload. Moreover, this is the first task where ARM-DPU outperforms Host-DPU for ≤ 50% offload while Host-DPU results decline in performance, indicating that frequent PCIe communication hinders performance when CPU performance is similar to the DPU ASIC, which is the case for LZ4 tasks (recall Section 5.7). The Reconfiguration setting worsens in performance as soon as offloading starts (with 160 MiB/s on 10% mark) and is 168× worse than full-DPU (169 MiB/s) static setup. Finally, co-processing LZ4 decompression tasks (Dec-LZ4) reduces Host CPU utilization by 77% on average.

Findings: This set of experiments evaluates Host-DPU and ARM-DPU hardware-accelerated co-processing tasks. For operators like DEFLATE (de)compress throughput increases monotonically with higher offload ratios, and full-DPU placement yields optimal throughput when buffer sizes remain static. However, for LZ4, we observe that 90% ARM-DPU co-processing outperforms both Host- and DPU-only setups. This shows that ARM participation hides buffering or PCIe communication cost. Reconfiguration performance is worse by up to an order of magnitude but despite the overhead, DPUs improve reconfiguration throughput for DEFLATE (de)compress tasks. Our results show that optimal placement is operator-specific and depends on both offload ratio and runtime configuration. Finally, co-processing always benefits the Host due to the significant (≥ 77%) reduction in CPU utilization.

5.10 Co-Processing Impact on Data Routing

In this experiment we answer *does co-processing degrade data routing performance?* Figure 15 summarizes the co-processing effects on a DPU that is simultaneously routing network traffic from a peer DPU. `DPU0` streams data to `DPU1` using `iperf3` at its single-thread roofline of 10 GiB/s, (no packet loss observed at `DPU1`'s OpenVSwitch). We pin the `iperf3` server thread on `DPU1` to non-OS cores and log transfer throughput at the NIC-ARM ingress, the earliest ingress stage of an off-path DPU. We offload two types of tasks to `DPU1` from the Host: *i)* memory-bound (DMA) and *ii)* processing-bound (DFLT / LZ4). For *i)*, we start a DMA workload at $t = 4s$ that moves $2MiB$ buffers from Host memory to DPU without further processing. The copy traffic shares the ARM's AXI/PCIe (memory interconnect controller) fabric with the NIC [57]. We observe that throughput drops by 25% (Figure 15a) and latency increases by 231% during offloading (Figure 15b). When DMA offloading stops ($t = 21s$), the NIC processes its backpressured RX queues, which is the reason for the increased throughput (13 GiB/s) and high latency. For *ii)*, $2MiB$ buffers are passed to the (de)compression ASIC. Since the ASIC has its own private SRAM and the task is compute-bound [52, 53], neither throughput nor latency are affected.

Findings: Bandwidth-dominant tasks (DMA) that stress the DPU's memory fabric degrade NIC routing while compute-bound ASIC co-processing (DFLT / LZ4) leaves routing unaffected.

5.11 Key Takeaways and Discussion

This paper is driven by the key observation that near-network processing has variable performance characteristics, either before

or after receiving data on the Host [61, 62]. The following insights are the results of our experimental analysis:

DPU adaptivity is expensive. Frequent `DOCA Context` reconfiguration yields the lowest throughput whereas static allocation achieves the highest performance and surpasses SIMD implementations. On average, static allocation outperforms CPUs by 14× and reconfiguration by 22×. This aligns with the original DPU focus on static, long-lived network workloads. Notably, hardware vendors address reconfiguration overhead by offering more threads (from 8 in BF2 to 16 in BF3) or adding programmable datapaths (DPA) [54].

DPU's ARM is improving. BF3 ARM CPUs match server-grade x86 performance in single-thread workloads, reflecting a 40% improvement over BF2. Our experiments show that the single-thread performance gap with x86 reduced from 51% (BF2) to 19% (BF3), making BF3 ARM a capable processing target. Our DEFLATE decompression experiments show that BF3's ARM matches its ASIC during frequent reconfiguration and performs on par with the Host. Moreover, LZ4 co-processing shows that a 90% BF3 ARM and ASIC data split outperforms all other configurations.

DPUs become ingestion-focused. The current set of DPU hardware accelerators suggests a preference on ingestion. Our LZ4 and DEFLATE Decompression experiments show that static full-DPU offloads outperforms any other configuration. This finding in addition to the DPUs losing compression acceleration, which is useful post-data processing, and eventually losing RegEx, indicate that they are directed towards ingestion-only pipelines. System designers will make the most out of DPUs on early pipeline tasks.

Application-level adaptivity is required. Our experiments reveal that DPUs do not necessarily outperform CPUs. In LZ4 and RegEx, CPUs match or outperform DPUs while RegEx depends on query configuration. Orchestrating task placement and preallocating `Contexts` may improve throughput but DPU heterogeneity complicates these decisions. Optimal buffer sizes cannot be known a priori, output buffers may exceed current capacity (e.g. decompression), and device differences impose constraints (e.g., BF2 128 MiB vs. BF3 2 MiB max buffers). Thus, real-time workload reconfiguration and profiling are required for robustness and performance.

DPU software stack needs improvement. Our experiments show that DPU hardware and software need refinement. Our suggestions to hardware designers are: *Lower re-configuration costs*, applications must cope with skewed workloads and context switches; lightweight buffer re-configuration mechanisms would boost performance. *Re-balance or unify the accelerator mix*, on-card accelerators vary not only across vendors but even across models; designers should provide graceful co-processing primitives or fall-backs. *Enable task-level co-processing*, co-processing tasks can outperform DPU-only datapaths as shown in LZ4 analysis; a task-offload API may transparently steer tasks to the better engine. *Offer finer-grained observability*, DPUs currently lack task or queue performance counters; exposing such metrics allows future applications, like task schedulers, to implement placement with less overhead.

6 RELATED WORK

Hardware Acceleration Fallbacks: DPDPU [22] envisions a software framework where DPU tasks use hardware acceleration or fall back into CPU-based implementations. Our work is orthogonal to the DPDPU vision since it dictates when to offload to hardware accelerators or revert to CPUs; thus systems in the DPDPU vision can adopt our findings as heuristics or reconfiguration components.

On-Switch Processing: Sonata [20] allows network telemetry queries that exploit on-switch capabilities, NetAccel [36] pushes OLAP queries to a switch, Zero-sided RDMA [24] reduces Host CPU usage by offloading RDMA to the switch, and P4DB [25] uses switches for OLTP queries. This work highlights the growing heterogeneity in modern networks, where network-facing switches co-operate with Host-side DPUs for ingress-based pipelines.

DPU Offload and Characterization: Prior work on DPU offloading and benchmarking focuses on communication patterns, system integration, or compression pipelines. DPUBench [43, 44] and Xing et al. [67] characterize communication efficiency and focus on round-trip communication behavior on DPUs. LineFS [29], Styx [26], and Fuyao [41] showcase system-level DPU integration that offloads common OS-level tasks from file systems, kernel services, or serverless runtimes. PEDAL [38] focuses on compression acceleration in MPI-based workloads. Our paper complements these works by focusing on the operator level, benchmarking compute-intensive primitives on BF2 and BF3 while additionally exploring co-processing on the Host-DPU continuum; thus we provide insights for future systems that fully harness Host-DPU setups.

Near-Data Processing: This line of work explores cooperative execution between Host and PIM RAM (Processing-In-Memory), SSDs, or GPUs. PIM work reduces inter-operator data movement by exploiting the highly parallel PIM architectures and splitting query operators across Hosts and PIMs [8, 9, 27, 28, 39]. SSDs enable in-storage processing [21, 30, 32, 64], where analytical or key-value queries are executed locally; avoiding excessive PCIe traffic. GPU work reduces data movement or CPU utilization through full or partial offloads. HippogriffDB [37] is a full-offload system that decompresses column blocks on the GPU directly from SSDs. HybridSA [31] is a co-processing system that dispatches regexes to the GPU but executes irregular patterns on CPU. Near-network operates in the ingress path, before data reach RAM, SSDs, or GPUs; thus it is foundational for Host-less inter-device data paths [33].

7 CONCLUSION

In this paper, we lay the foundation for future data management research on SmartNICs by analyzing the performance of data-related hardware accelerators on BlueField DPUs. Our analysis shows that throughput hinges on input size, reconfiguration requirements, and user-imposed query settings. While DPUs outperform CPUs by orders of magnitude in certain tasks, they do not tolerate online reconfiguration. We further explore co-processing as a means to fully exploit DPUs, showing that it is viable against Host- or DPU-only approaches. Although co-processing unlocks substantial performance benefits, frequent reconfiguration imposes non-trivial overhead in dynamic, query-driven, and multi-tenant workloads. Finally, our findings suggest that future DPUs are already evolving beyond network workloads, thanks to more capable ARM CPUs.

ACKNOWLEDGMENTS

This work is funded by the German Federal Ministry of Education and Research under the grants BIFOLD24B and 02P22A060.

REFERENCES

[1] Daniel Abadi, Anastasia Ailamaki, David Andersen, Peter Bailis, Magdalena Balazinska, Philip A Bernstein, Peter Boncz, Surajit Chaudhuri, Alvin Cheung, Anhai Doan, et al. 2022. The Seattle report on database research. *Commun. ACM* 65, 8 (2022), 72–79.

[2] Mahmood Ahmadi and Stephan Wong. 2006. Network processors: Challenges and trends. In *Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing (ProRisc)*. 223–232.

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[4] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, et al. 2019. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*. 1743–1756.

[5] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Conference on Innovative Data Systems Research (CIDR)*, Vol. 8. 28.

[6] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data*. 2205–2217.

[7] Ron Avnur and Joseph M Hellerstein. 2000. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. 261–272.

[8] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Processing-in-memory for databases: Query processing and data transfer. In *Proceedings of the 19th International Workshop on Data Management on New Hardware (DaMoN)*. 107–111.

[9] Arthur Bernhardt, Andreas Koch, and Ilia Petrov. 2023. pimdb: From main-memory DBMS to processing-in-memory dbms-engines on intelligent memories. In *Proceedings of the 19th International Workshop on Data Management on New Hardware (DaMoN)*. 44–52.

[10] Eric Biggers. 2025. Libdeflate. https://github.com/ebiggers/libdeflate Accessed: October 6, 2025.

[11] Idan Burstein. 2021. Nvidia data center processing unit (DPU) architecture. In *2021 IEEE Hot Chips 33 Symposium*. IEEE, 1–20.

[12] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. 2021. Polardb serverless: A cloud native database for disaggregated data centers. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 2477–2489.

[13] Google Cloud. 2024. *Google Cloud Storage*. https://cloud.google.com/storage/docs/best-practices#uploading Accessed: October 6, 2025.

[14] Yann Collet. 2025. LZ4. https://github.com/lz4/lz4 Accessed: October 6, 2025.

[15] Heather Craddock, Lakshmi Prasanna Konudula, Kun Cheng, and Gökhan Kul. 2019. The Case for Physical Memory Pools: A Vision Paper. In *12th International Conference on Cloud Computing (CLOUD)*. Springer, 208–221.

[16] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. 215–226.

[17] Sebastian Deorowicz. 2024. Silesia compression corpus. https://sun.aei.polsl.pl//~sdeor/index.php?page=silesia Accessed: October 6, 2025.

[18] DuckDB. 2025. *Regex Internals*. https://duckdb.org/docs/sql/functions/regular_expressions.html Accessed: October 6, 2025.

[19] Brian Gold, Anastasia Ailamaki, Larry Huston, and Babak Falsafi. 2005. Accelerating database operators using a network processor. In *Proceedings of the 1st International Workshop on Data Management on New Hardware (DaMoN)*. 1–6.

[20] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 357–371.

[21] Niclas Hedam, Morten Tychsen Clausen, Philippe Bonnet, Sangjin Lee, and Ken Friis Larsen. 2023. Delilah: eBPF-offload on computational storage. In *Proceedings of the 19th International Workshop on Data Management on New Hardware (DaMoN)*. 70–76.

[22] Jiasheng Hu, Philip A Bernstein, Jialin Li, and Qizhen Zhang. 2025. DPDPU: Data Processing with DPUs. In *Conference on Innovative Data Systems Research (CIDR)*.

[23] Intel. 2025. *Introduction to Hyperscan*. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-hyperscan.html Accessed: October 6, 2025.

[24] Matthias Jasny, Lasse Thostrup, and Carsten Binnig. 2023. Zero-Sided RDMA: Network-Driven Data Shuffling. In *Proceedings of the 19th International Workshop on Data Management on New Hardware (DaMoN)*. 82–85.

[25] Matthias Jasny, Lasse Thostrup, Tobias Ziegler, and Carsten Binnig. 2022. P4db-the case for in-network oltp. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data*. 1375–1389.

[26] Houxiang Ji, Mark Mansi, Yan Sun, Yifan Yuan, Jinghan Huang, Reese Kuper, Michael M Swift, and Nam Sung Kim. 2023. STYX: Exploiting SmartNIC capability to reduce datacenter memory tax. In *USENIX Annual Technical Conference (USENIX ATC)*. 619–633.

[27] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. 2022. PIM-Tree: A Skew-Resistant Index for Processing-in-Memory. *Proceedings of the VLDB Endowment* 16, 4 (2022), 946–958.

[28] Tiago R Kepe, Eduardo C de Almeida, and Marco AZ Alves. 2019. Database processing-in-memory: An experimental study. *Proceedings of the VLDB Endowment* 13, 3 (2019), 334–347.

[29] 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 28th ACM Symposium on Operating Systems Principles (SIGOPS)*. 756–771.

[30] Christian Knödler, Naeem Ramzan, and Ilia Petrov. 2025. hybridNDP: Dynamic Operation Offloading and Cooperative Query Execution in Smart Storage Settings. In *Proceedings of the 28th International Conference on Extending Database Technology (EDBT)*. 769–782.

[31] Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2024. HybridSA: GPU Acceleration of Multi-pattern Regex Matching using Bit Parallelism. *Proceedings of the ACM on Programming Languages (OOPSLA2)* 8 (2024), 1699–1728.

[32] Kitaek Lee, Insoon Jo, Jaechan Ahn, Hyuk Lee, Hwang Lee, Woong Sul, and Hyungsoo Jung. 2023. Deploying computational storage for htap dbmss takes more than just computation offloading. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1480–1493.

[33] Alberto Lerner and Gustavo Alonso. 2024. Data Flow Architectures for Data Processing on Modern Hardware. In *40th International Conference on Data Engineering (ICDE)*. IEEE, 5511–5522.

[34] Alberto Lerner, Carsten Binnig, Philippe Cudré-Mauroux, Rana Hussein, Matthias Jasny, Theo Jepsen, Dan RK Ports, Lasse Thostrup, and Tobias Ziegler. 2023. Databases on Modern Networks: A Decade of Research that now comes into Practice. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3894–3897.

[35] Alberto Lerner and Philippe Bonnet. 2021. Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 2852–2858.

[36] Alberto Lerner, Rana Hussein, and Philippe Cudre-Mauroux. 2019. The Case for Network Accelerated Query Processing. In *Conference on Innovative Data Systems Research (CIDR)*.

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

[38] Yuke Li, Arjun Kashyap, Weicong Chen, Yanfei Guo, and Xiaoyi Lu. 2024. Accelerating lossy and lossless compression on emerging bluefield dpu architectures. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 373–385.

[39] Chaemin Lim, Suhyun Lee, Jinwoo Choi, Jounghoo Lee, Seongyeon Park, Hanjun Kim, Jinho Lee, and Youngsok Kim. 2023. Design and analysis of a processing-in-dimm join algorithm: A case study with upmem dimms. *Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data* 1, 2 (2023), 1–27.

[40] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A deep dive into common open formats for analytical DBMSs. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3044–3056.

[41] Guowei Liu, Laiping Zhao, Yiming Li, Zhaolin Duan, Sheng Chen, Yitao Hu, Zhiyuan Su, and Wenyu Qu. 2024. FUYAO: DPU-enabled Direct Data Transfer for Serverless Computing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vol. 3. 431–447.

[42] LzTurbo. 2025. TurboBench. https://github.com/powturbo/TurboBench Accessed: August 31, 2025.

[43] Benjamin Michalowicz, Kaushik Kandadi Suresh, Hari Subramoni, Dhabaleswar Panda, and Steve Poole. 2023. DPU-bench: a micro-benchmark suite to measure offload efficiency of SmartNICs. In *Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good*. 94–101.

[44] Benjamin Michalowicz, Kaushik Kandadi Suresh, Hari Subramoni, Dhabaleswar K DK Panda, and Steve Poole. 2023. Battle of the bluefields: An in-depth comparison of the bluefield-2 and bluefield-3 smartnics. In *2023 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 41–48.

[45] Microsoft. 2023. *Azure Blob Storage Performance Checklist*. https://learn.microsoft.com/en-us/azure/storage/blobs/storage-performance-checklist#capacity-and-transaction-targets Accessed: October 6, 2025.

[46] MinIO Blog. 2024. AIStor: NVIDIA BlueField-3 DPUs. https://blog.min.io/aistor-nvidia-bluefield-3-dpus/. Accessed: 2025-05-03.

[47] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and Mitigating Data Stalls in DNN Training. *Proceedings of the VLDB Endowment* 14, 5 (2021), 771–784.

[48] Sobhan Moosavi, Mohammad Hossein Samavatian, Srinivasan Parthasarathy, Radu Teodorescu, and Rajiv Ramnath. 2019. Accident risk prediction based on

heterogeneous sparse data: New dataset and insights. In *Proceedings of the 27th ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*. 33–42.

[49] Intel Networking. 2024. *Intel Infrastructure Processing Unit.* https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html Accessed: October 6, 2025.

[50] NVIDIA. 2024. *BlueField Networking Platform.* https://www.nvidia.com/en-us/networking/products/data-processing-unit/ Accessed: October 6, 2025.

[51] NVIDIA. 2024. *Mellanox acquires Network Intelligence Titan IC.* https://nvidianews.nvidia.com/news/mellanox-to-acquire-world-leading-network-intelligence-technology-developer-titan-ic-to-strengthen-leadership-in-security-and-data-analytics Accessed: October 6, 2025.

[52] NVIDIA. 2025. DOCA Compress. https://docs.nvidia.com/networking/display/bluefieldbsp480/compression+acceleration Accessed: May 31, 2025.

[53] NVIDIA. 2025. DOCA Compress. https://docs.nvidia.com/networking/display/nvidia-bluefield-dpu-bsp-v4-7-0.0.pdf#:~:text=NVIDIA%20BlueField%20DPU%20BSP%20v4,more%20information%2C%20please%20refer%20to Accessed: May 31, 2025.

[54] NVIDIA. 2025. DOCA DPA. https://docs.nvidia.com/doca/sdk/dpa+subsystem/index.html Accessed: October 6, 2025.

[55] NVIDIA. 2025. *DOCA Memory Model.* https://docs.nvidia.com/doca/sdk/doca+core/index.html#src-3169008654_id-.DOCACorev2.9.0LTS-DOCAMemorySubsystem Accessed: October 6, 2025.

[56] NVIDIA. 2025. DOCA Random Data suggestions. https://docs.nvidia.com/doca/sdk/doca+bench/index.html#src-3453016152_id-.DOCABenchv2.9.1-RandomData Accessed: October 6, 2025.

[57] NVIDIA. 2025. DPU Detailed Interconnect Architecture. https://docs.nvidia.com/networking/display/nvidia-bluefield-3-dpu-controller-user-manual.pdf#:~:text=BlueField,0 Accessed: May 31, 2025.

[58] Matthew Perron, Raul Castro Fernandez, David DeWitt, Michael Cafarella, and Samuel Madden. 2023. Cackle: Analytical Workload Cost and Performance Stability With Elastic Pools. *Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data* 1, 4 (2023), 1–25.

[59] Red Hat Developers Blog. 2025. DPU-enabled Networking in OpenShift and NVIDIA DPF. https://developers.redhat.com/articles/2025/03/20/dpu-enabled-networking-openshift-and-nvidia-dpf. Accessed: 2025-05-03.

[60] Amazon Web Services. 2024. *AWS Well Architected.* https://docs.aws.amazon.com/wellarchitected/latest/analytics-lens/best-practice-10.3---use-file-compression-to-reduce-number-of-files-and-to-improve-file-io-efficiency..html Accessed: October 6, 2025.

[61] Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. 2023. Distributed GPU Joins on Fast RDMA-capable Networks. *Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data* 1 (2023), 1–26.

[62] Lasse Thostrup, Daniel Failing, Tobias Ziegler, and Carsten Binnig. 2022. A DBMS-centric Evaluation of BlueField DPUs on Fast Networks. In *Thirteenth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. 1–10.

[63] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*. 1041–1052.

[64] Tobias Vincon, Arthur Bernhardt, Ilia Petrov, Lukas Weber, and Andreas Koch. 2020. nKV: near-data processing with KV-stores on native computational storage. In *Proceedings of the 16th International Workshop on Data Management on New Hardware (DaMoN)*. 1–11.

[65] VMware Cloud Foundation Blog. 2024. The Rise of DPUs in the Infrastructure. https://blogs.vmware.com/cloud-foundation/2024/07/16/the-rise-of-dpus-in-the-infrastructure. Accessed: 2025-05-03.

[66] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated database systems. In *Companion of the 2023 ACM SIGMOD International Conference on Management of Data*. 37–44.

[67] Tong Xing, Hesam Tajbakhsh, Israat Haque, Michio Honda, and Antonio Barbalace. 2022. Towards portable end-to-end network performance characterization of smartnics. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*. 46–52.

[68] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An empirical evaluation of columnar storage formats. *Proceedings of the VLDB Endowment* 17, 2 (2023), 148–161.

[69] Ling Zhang, Shaleen Deep, Avrilia Floratou, Anja Gruenheid, Jignesh M Patel, and Yiwen Zhu. 2023. Exploiting Structure in Regular Expression Queries. *Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data* 1, 2 (2023), 1–28.

[70] Qizhen Zhang, Philip A Bernstein, Badrish Chandramouli, Jiasheng Hu, and Yiming Zheng. 2024. DDS: DPU-Optimized Disaggregated Storage. *Proceedings of the VLDB Endowment* 17 (2024), 3304–3317.