DCRuntime: Toward Efficiently Sharing CPU-GPU Architectures

Ovidiu-Cristian Marcu, Grégoire Danoy, and Pascal Bouvry

University of Luxembourg, Luxembourg
{ovidiu-cristian.marcu,gregoire.danoy,pascal.bouvry}@uni.lu

Abstract. Managing distributed data movement and computation for large-scale, data-intensive applications (Big Data, ML/AI) on modern heterogeneous architectures (CPU, GPU) presents significant challenges. Current systems often rely on passive, application-driven (pull-based) data access, leading to inefficient resource utilization, particularly high GPU idle times (up to 70%), complex manual memory management, and limited opportunities for global optimization and fault tolerance. This paper introduces DCRuntime, a vision for a unified, runtime-orchestrated system designed to actively manage both data and compute streaming. DCRuntime employs a proactive, push-based "compute-follows-data" execution model. It exposes two core abstractions: 1) distributed data streams, representing potentially unbounded sequences of (im)mutable buffers spanning cluster resources, and 2) global compute streams, enabling asynchronous task execution across multiple devices, tightly coupled with data availability. By delegating data sourcing, sinking, shuffling, and compute scheduling to the runtime, DCRuntime gains a global perspective to optimize data placement, minimize I/O stalls, mitigate interference and power jitter, and enable faster, application-aware fault recovery. This approach aims to abstract away low-level complexities, significantly improve resource utilization (especially for GPUs), enhance scalability, and provide a resilient foundation for demanding workloads on shared, heterogeneous high-performance computing infrastructure.

Keywords: data stream · compute stream · data movement · distributed memory and compute orchestration · compute-follows-data · shared GPU clusters.

1 Introduction: Background and Vision

Distributed memory management is a difficult system problem exacerbated by three factors: 1) large-scale data-intensive distributed applications that require efficient data movement in heterogeneous exascale architectures (e.g., CPU, GPU, high-end networking), 2) specialized frameworks (big data, machine learning) needing interoperability (i.e., to avoid inefficient data transfers and sharing) and portability across device-specific accelerators, and 3) the need to co-design memory management with system schedulers for energy-efficient fault tolerance (e.g., lineage, replication, checkpointing).

Solving this memory system problem would alleviate significant burdens on application developers who currently struggle with manual memory management, explicit data transfers, complex logic for data consistency and failure recovery, and the intricacies of dynamic data partitioning for scalability. Effective memory management is increasingly important given the rising cost and rapid advancements in specialized hardware (like NVIDIA GPUs), making it exceedingly difficult for developers to efficiently tune both existing and new workloads to maximize the utilization of these costly resources, particularly in shared, multi-tenant [38, **?**] GPU environments that optimize resource usage and reduce training costs for deep neural network applications [34].

Entity-scoped multi-tenancy and collaboration in data-intensive systems present significant challenges, particularly for frameworks like JAX/XLA [11]. While JAX/XLA optimizes code for specific hardware accelerators (CPUs, GPUs, TPUs), existing limitations related to cluster device sharing (multi-tenancy) and dynamic scaling hinder efficient resource utilization in collaborative environments. Addressing these limitations will necessitate a reevaluation of how data and compute are abstracted, highlighting the need to orchestrate both data and compute management, which we discuss next.

The need to abstract away data orchestration. Most (in-memory and/or disk-based) distributed storage architectures remain *passive* relative to computations while exposing *pull-based* (application-initiated) data access APIs to compute-optimized systems for handling big data, machine learning, and artificial intelligence (BD/ML/AI) workloads. Processing engines typically deploy pipelines of operators, including source, sink, and shuffle operators. Source operators fetch input data from a storage system using a pull-based approach, while sink operators write data to the storage system using a *push-based* approach. Additionally, shuffle operators are responsible for redistributing data based on partitioning methods, such as key-based partitioning. Traditionally, the shuffle mechanism is implemented at the application level. Unfortunately, system developers handling large-scale computations often overlook critical optimizations due to *lacking a global I/O perspective*. For example, GPUs can spend up to 70% of their time waiting for data [7], a stall primarily caused by data movement latency.

A *push-based* strategy for data source delivery, however, can mitigate some of these delays by proactively staging data closer to the compute resource. This approach may look similar to how operating systems may prefetch data into the page cache before an application requests it, or how CUDA [18, 33] programmers may invoke `cudaMemPrefetchAsync` to preload data onto the GPU before launching a kernel. However, application-driven prefetching requires complex management logic within the application itself and often lacks a holistic view of the system's state, including network congestion, storage tier performance, or the resource needs of other concurrent applications/tenants. This can lead to suboptimal prefetching decisions (either too aggressive, wasting resources, or too conservative, still resulting in stalls). Furthermore, even with prefetching, the final step often involves the application checking or waiting for data arrival before initiating computation, introducing potential micro-stalls.

A push-based strategy for data delivery, orchestrated by the runtime system itself, fundamentally shifts this dynamic. Instead of computation pulling data when needed (or prefetching reactively), the runtime proactively pushes data to designated compute resources based on declared application requirements (e.g., data stream dependencies for compute kernels). This approach moves beyond mere prefetching; it enables the runtime, with its potential global view, to manage the end-to-end data flow and tightly couple data availability with compute scheduling. By shifting from a purely application-initiated model (pull or app-level prefetch) to a runtime-initiated push model, storage systems and processing frameworks can minimize data stalls more effectively, enable more sophisticated cross-application optimizations, and improve overall throughput, as envisioned by DCRuntime.

The need to abstract away compute execution. Additionally, extreme power jitter [3] [section 3.3], arising from the synchronization of tasks like checkpointing [28], collective communication [10], and training computations during large language model (LLM) training, presents a significant challenge (synchronized power fluctuations across tens of thousands of GPUs can strain data center power grids, potentially reaching tens of megawatts). To address this, system developers should be aware of pending computations from source, sink, and shuffle operations and utilize idle GPU resources for other computations to reduce power jitter by dynamically scheduling other tasks on otherwise idle GPUs. For example, this is possible with NVIDIA Hopper, which provides the ability for waiting threads to sleep until all other threads arrive [4]. On previous chips, waiting threads would spin on the barrier object in shared memory.

Therefore, a runtime system should not only actively manage data movement on behalf of applications but also be responsible for scheduling computation when data is available and efficiently utilize available memory and compute resources by sharing them with other applications when possible.

Traditionally, system researchers explore runtime API abstractions to address this question: *How to abstract distributed memory management into a single-system runtime* [36, 5, 39, 11, 29, 17, 12]? We argue that the main runtime *strategy and challenge* should be to delegate (abstract away) *both* data movement and computation to a distributed runtime. In this context, data movement encompasses data allocation, garbage collection, transfer, ingestion, and shuffling on top of distributed memory management. We believe that this dual objective is best achieved in a *streaming* runtime designed with a *compute-follows-data* strategy, as further introduced.

2 Motivation: Lack of Global I/O and Compute Perspective in Current Middleware Systems

Today's high-performance computing relies on complex, heterogeneous systems comprising CPUs, GPUs, and specialized hardware. Applications processing massive datasets require efficient data movement across these components, but often face severe bottlenecks. Expensive accelerators like GPUs frequently sit

idle, waiting for data, which points to fundamental inefficiencies in how data is supplied to compute resources, limiting overall system throughput and increasing operational costs.

The software ecosystem adds another layer of complexity. Diverse frameworks for big data, machine learning, and AI often lack seamless interoperability, forcing cumbersome data transfers and hindering portability across different accelerators. Developers currently struggle with low-level details: manually managing memory, orchestrating data transfers, ensuring consistency, implementing fault tolerance, and scaling applications through intricate partitioning logic, detracting significantly from core application development.

Current system architectures typically exacerbate these issues. Storage systems often remain passive, requiring applications to actively pull data when needed. This application-centric control over data flow, including input, output, and data redistribution (shuffling), lacks a global system perspective. This prevents holistic optimization, hinders proactive data placement, and can lead to system-wide inefficiencies like synchronized power spikes during large-scale computations. A more integrated approach is needed, where the runtime takes a proactive role in orchestrating both data and compute resources. The goal is to abstract away low-level concerns like data partitioning and movement efficiency, allowing developers to concentrate solely on defining the computation and its inherent data relationships, while the runtime should optimize resource utilization, performance etc.

We design the following experimental setup to evaluate GPU performance optimization techniques using a convolutional neural network (CNN) for CIFAR-100 image classification and large 4000×4000 matrix multiplications as workloads, see Figure 1. The baseline consists of sequential execution on a single GPU, where data loading, transfer, computation, and result retrieval occur sequentially without overlapping.

The optimized implementation introduces several techniques to enhance GPU utilization. It employs compilation caching to avoid recompiling JAX functions between runs, carefully separates and times different execution phases (data loading, compilation, computation, evaluation), and implements precise memory transfer optimization with explicit CPU-to-GPU transfers. We measure both GPU compute throughput and memory utilization.

Most significantly, the experimental approach implements concurrent processing and parallelized data transfers through an 'AdvancedMatrixWorker' class that creates a pipeline with separate threads for transfers and computation. This enables overlapping of operations that would otherwise happen sequentially, allowing matrix computations to run in the background during CNN training. Although we observe improved GPU utilization and overall end to end application runtime decreases, there is more room for improvement.

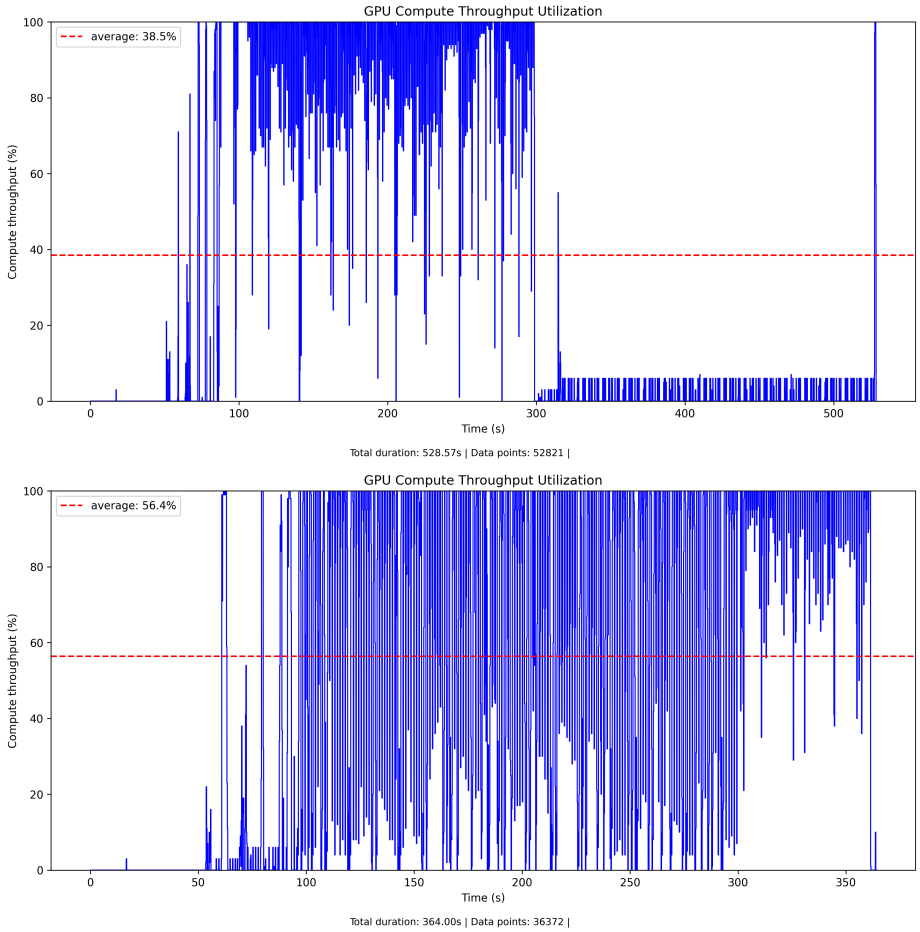

Fig. 1: Sequential (top) versus Concurrent (bottom) workloads running on GPU.

3 The DCRuntime Approach

This paper introduces the *stream-active runtime approach*, DCRuntime, to manage distributed data movement and computation across heterogeneous architectures, as shown in Figure 2. DCRuntime shifts data movement management, including source/sink handling and data shuffling, from BD/ML/AI applications directly to DCRuntime. It operates on a virtual log-structured in-memory storage framework that exposes references to pooled data stream mutable buffers and computation streams (see next section). By leveraging immutable data access patterns and facilitating efficient real-time data movement, the DCRuntime architecture will be deployed on tens of thousands of large many-core CPU and CPU-GPU nodes. DCRuntime will harness their memory and computational re-

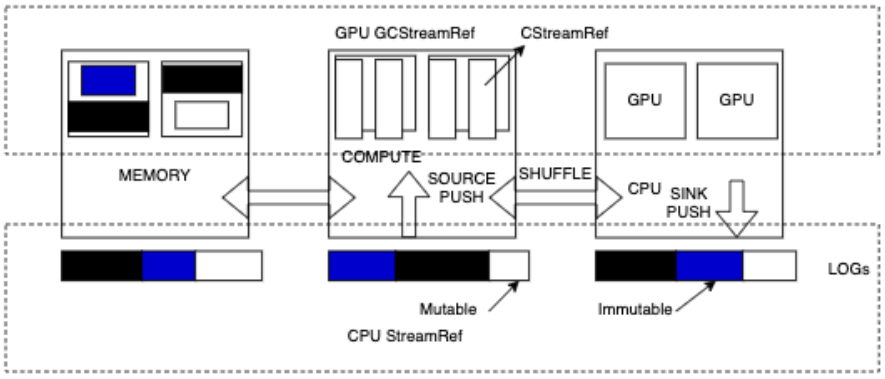

Fig. 2: Distributed data and compute runtime (DCRuntime) for shared heterogeneous computing. Data movement and computation scheduling are actively handled by the DCRuntime.

sources while ensuring efficient and transparent communication with traditional disk-based file storage systems.

We propose a push-based streaming execution model that enables DCRuntime to efficiently leverage application-specific data (such as consumer/producer offsets and data access patterns, including read, write, and shuffle). This model supports several optimizations: scalable data movement partitioning algorithms; faster stream storage recovery; mitigation of application stragglers; power fluctuation mitigation during large-scale ML/AI training by utilizing idle GPU resources for other computing tasks; and reduced I/O interference in multi-CPU-GPU deployments for multiple applications sharing an exascale computing infrastructure.

The rationale behind a push-based streaming model, in addition to its low-latency processing advantages, is driven by the continuous and bursty [3] data processing demands of use cases such as LLM training and real-time stream processing that often involve petabytes of input data and checkpointing data with tens of TB/s peak throughput. By proactively managing data movement (input, output, and shuffling), a push-based stream storage runtime minimizes GPU idle time, reduces costs, and enables optimizations not easily achievable with a pull-based model, such as reduced I/O interference, faster recovery, straggler mitigation, and higher ingestion/checkpointing throughput. Moreover, insights into application behavior provided by the push-based protocol's offsets eliminate the need for costly monitoring infrastructure to predict access patterns, thereby simplifying optimization and fault recovery.

Through its global view of I/O, enabled by a push-based in-memory computing approach, DCRuntime promises significant performance improvements for data-intensive applications by actively handling data movement and eliminates the need for manual memory tuning and inefficient application-based data man-

agement. This will also help efficiently handle fault tolerance, a critical challenge at exascale that can result in significant wasted compute capacity (20% or more) due to failures and recovery, as highlighted by the European Strategic Research Agenda for HPC [16] [pages 79-82]. DCRuntime achieves this through a virtual immutable log-structured design and its novel push-based stream-active programming model that couples global computational streams with data streams. Our envisioned approach enables valuable insights into the patterns of data access in applications, facilitating faster recovery, as detailed in the following sections.

Our global vision for DCRuntime is a unified stream-active storage and compute architecture for BD/ML/AI processing on heterogeneous HPC infrastructure, enabled by a **push-based streaming execution model** with the following key benefits:

- **Unified CPU-GPU Deployment and Optimized Performance:** DCRuntime will be deployed across the CPU-GPU HPC infrastructure, leveraging combined memory and computing resources to support a push-based, stream-based programming model (see Figure 2 and API). This unified approach will facilitate efficient data movement and processing at exascale.
- **Transparent Scalability and Resiliency:** DCRuntime will provide users with dynamic partitioning and automatic resiliency while efficiently scaling ML pipelines on HPC infrastructure. This will enable users to focus on their core research and development tasks without the burden of managing manual scaling and fault-tolerant storage.

DCRuntime introduces a memory-agnostic runtime that exposes two core abstractions: 1) distributed data streams, which provide a potentially unbounded set of (im)mutable buffers spanning the aggregated memory of a cluster of CPU-GPU nodes, supported by pools of mutable buffers that become immutable once added to the data streams; and 2) global compute streams, similar to CUDA streams, designed for asynchronous pipelining of computations and I/O across multiple devices. Applications use compute stream references to access these global compute streams, enabling resource sharing through futures (e.g., as in XLA/PJRT and Ray). Compute streams are tightly coupled with data streams via a push-based protocol that orchestrates computations based on data availability. Data streams abstract memory allocation (e.g., malloc) and data transfer. DCRuntime actively manages both data movement and computation scheduling, providing a global I/O perspective and facilitating seamless resource sharing for interoperability between applications and data-intensive engines. Its push-based model, unlike traditional pull-based approaches, is central to optimizing data movement and minimizing GPU idle time by proactively anticipating computational needs.

Recognizing that data-intensive applications (e.g., real-time streaming, LLM training) demand continuous data movement, including input, output, and shuffling, we propose delegating these operations to DCRuntime itself. Specifically, source operators register their input stream requirements (including any filtering

functions) with DCRuntime, which then proactively fills input buffers using a push-based approach. Sink operators operate on preregistered stream buffers and notify DCRuntime when data are ready to be written, triggering asynchronous persistence to disk and buffer reuse. Shuffle operations function similarly, allowing DCRuntime to reorganize input stream buffers asynchronously.

Our vision of decoupling data movement operations from processing operators is realized through the DCRuntime middleware layer positioned between the disk-based file storage system and application engines. DCRuntime manages both CPU and GPU host memory and integrates GPU device memory via native code (e.g., CUDA streams) under a push-based approach. Garbage collection is managed at the DCRuntime level. All stream metadata are registered with DCRuntime before and during deployment when source, sink, and shuffle operations are delegated. When an application crashes or shuts down, DCRuntime automatically cleans up associated active streams.

Managing data movement for scalability, performance, faster recovery, and reduced power jitter at exascale can be significantly more efficient when managed by the DCRuntime system. While handling metadata for a large number of streams presents challenges, DCRuntime differs from traditional approaches, where this responsibility typically falls on application developers and their engines, leading to suboptimal performance and increased complexity.

4 DCRuntime Data and Compute API

DCRuntime's goal is to provide a global perspective on distributed memory and compute resources. The DCRuntime API exposes two core abstractions: 1) global compute streams, which have semantics similar to CUDA streams but refer to computational resources across multiple nodes or devices rather than a single device, and 2) data streams, which consist of immutable, shareable buffers distributed across multiple nodes and tightly associated with compute streams. This design ensures parallelism for data-intensive applications while enabling the overlap of computations and data movement, both delegated to DCRuntime for scheduling. Data streams are coupled with compute streams to allow the DCRuntime runtime to implement a push-based *compute-follows-data* execution.

Table 1 outlines the essential data and compute stream APIs. DCRuntime manages global memory and computational resource pools exposed via these APIs. Consumer and producer operators (e.g., GPU kernel tasks) within compute engines create data and compute streams to interact with shared in-memory buffers managed by DCRuntime. The source, sink, and shuffle operators delegate their read and write I/O operations to DCRuntime, which manages these shared stream buffers. Stream creation generates a reference (`StreamRef`) similar to RPC references in [36]; however, data stream references offer better support for partitioning by pointing to stream partitions and their buffers across multiple nodes. A source is represented by a data stream of immutable stream buffers. A

Table 1: Data and Compute Stream Operations.

GCStreamRef	A reference to a global compute stream to associate with all data streams.
CStreamRef	A reference to a local compute stream.
StreamRef	A reference to a distributed data stream.
StreamBufferRef	A reference to a buffer (chunk/partition) of a data stream.
`CreateComputeStream( KernelAttributes, ResourceProperties) -> GCStreamRef`	Creates a new global compute stream and returns `GCStreamRef`. `ResourceProperties` can specify the number of nodes/devices to use or if sharing computational resources with other applications is possible.
`GetCStreams(GCStreamRef) -> Set<CStreamRef>`	Provides a set of existing `CStreamRef` references to local compute streams.
`CreateDataStream( GCStreamRef, Properties, [StreamRef]) -> StreamRef`	Creates a new data stream and returns `StreamRef`. It can optionally build on another data stream. `Properties` can tag data streams to be shared with other applications.
`GetNextBuffer(StreamRef) -> StreamBufferRef`	Dynamically provisions a local mutable `StreamBufferRef` reference to a memory chunk that acts as the application's local state.
`GetBuffers(StreamRef) -> Set<StreamBufferRef>`	Provides a set of existing local mutable `StreamBufferRef` references to memory chunks that act as the application's global state.
`WriteTo(StreamRef, StreamBufferRef)`	Writes the content of `StreamBufferRef` to a specified stream, marking the stream buffer immutable.
`ReadFrom(StreamRef) -> Set<StreamBufferRef>`	Provides a set of immutable `StreamBufferRef` references for push-based data access.
`ShuffleStream(StreamRef, ShuffleFunction, Set<Nodes>)`	Repartitions a data stream across a specified set of nodes.
`KernelFunction.Execute( GCStreamRef, StreamRef, GlobalPartitioner) -> StreamRef`	Schedules computation using global compute stream resources over a data stream and returns a reference to a (potentially new) data stream to share with other applications.
`KernelFunction.Execute( Set<CStreamRef>, Set<StreamBufferRef>, LocalPartitioner) -> StreamRef`	Schedules computation using local compute stream resources over a set of local stream buffers and returns a reference to a (potentially new) data stream to share with other applications.
`DestroyStream(StreamRef)`	Schedules the deletion of the stream.

sink converts mutable buffers into immutable ones. Shuffle operates on mutable buffers, which are materialized as immutable when transferred to data streams.

DCRuntime exposes global compute streams similar to single-device CUDA streams but with key differences. A CUDA stream is a sequence of operations (I/O and compute) executed in order on a specific CUDA device, allowing operations in different streams to run concurrently on the same device. In contrast, DCRuntime introduces compute streams that span multiple devices globally, though they may be internally implemented similarly to CUDA streams. `CStreamRef` functions similarly to a CUDA stream, while `StreamRef` is conceptually similar to a streaming topic. Whereas a kernel function typically executes by default in the CUDA default stream on a single device, DCRuntime compute streams enable kernel functions to execute by default across multiple devices. Additionally, compute streams are tightly coupled with data streams through a push-based execution protocol. The kernel function launch syntax resembles CUDA's but includes additional arguments for data streams associated with the compute streams. Push-based data consumption is handled transparently by DCRuntime.

Central to DCRuntime is its push-based streaming execution model. This model is deliberately chosen over traditional pull-based or even purely runtime-managed prefetching approaches for several key reasons:

- Tighter Compute-Data Coupling: In a pull or prefetch model, the compute engine typically initiates the data request and often needs to check for its completion before starting work. In DCRuntime's push model, the runtime knows precisely when the required data partitions have been pushed into the target StreamBufferRefs associated with a compute task (CStreamRef/GCStreamRef). This allows the runtime to directly trigger or schedule the computation immediately upon data readiness, minimizing the scheduling latency between data arrival and kernel launch. Control shifts from *compute asking for data* to *data arrival enabling compute*.
- Truly Proactive Orchestration: Application prefetching is inherently limited by the application's local view and predictive capability. Even runtime-managed prefetching often relies on predicting future pulls. DCRuntime's push model, however, operates on declared intent. Applications register their computational graphs and associated data stream dependencies via the API (Table I). The runtime uses this explicit declaration to proactively orchestrate data movement end-to-end, pushing data not just in anticipation of a pull, but as a direct consequence of the defined workflow.
- Enhanced Global Optimization Potential: By managing the active push of data, DCRuntime gains a superior advantage point for global optimization compared to systems merely reacting to pull requests or managing predictive prefetching. It can make more informed decisions about resource allocation, data placement, interference mitigation, power jitter reduction and simplified application flow control.

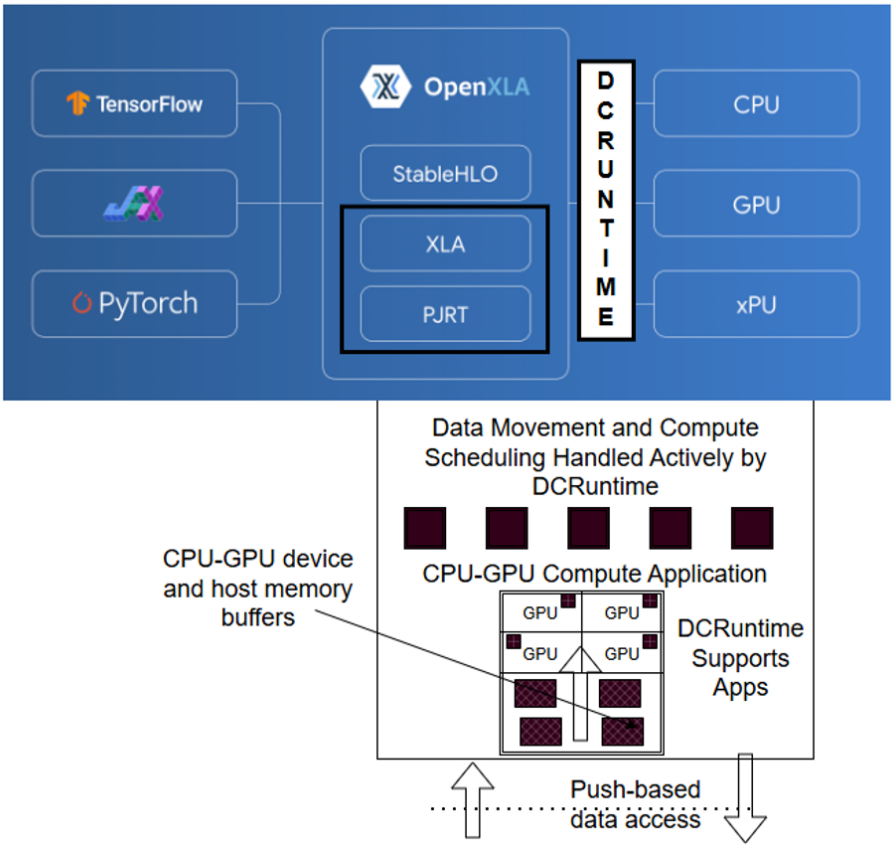

Fig. 3: DCRuntime middleware controls devices memory and data movement (eg CUDA streams) and shares responsibilities with PJRT.

5 Discussion: Implementation Feasibility and Challenges

JAX is a Python library designed for high-performance numerical computing and machine learning research. It provides a familiar NumPy-style API while enabling advanced capabilities like automatic differentiation, just-in-time compilation, and parallel computation across accelerators. JAX achieves its performance by building on XLA (Accelerated Linear Algebra), an open-source compiler that optimizes mathematical computations for various hardware platforms including GPUs, CPUs, and specialized ML accelerators. Together, JAX and XLA allow researchers and engineers to write high-level Python code that executes with near-native hardware performance.

XLA takes computational graphs from frameworks like JAX and applies optimizations such as fusion (combining operations to reduce memory transfers), layout optimization (arranging data for efficient memory access patterns), and spe-

cialized code generation for target hardware. These optimizations significantly reduce memory usage and execution time compared to naive implementations. XLA achieves this through a multi-stage compilation process that converts high-level operations into optimized machine code specifically designed for the target architecture.

PJRT (Portable JAX Runtime) provides the critical abstraction layer that enables JAX and XLA to run efficiently across diverse hardware. It defines a uniform device API with components such as PjRtClient (managing framework-device communication), PjRtDevice (representing computational resources), PjRtMemorySpace (abstracting memory locations), PjRtBuffer (handling data storage and transfer), and PjRtExecutable (managing compiled computation). This architecture separates framework concerns from hardware-specific implementations, allowing frameworks to interact with a consistent API while hardware vendors can focus on optimizing their specific implementations without modifying the frameworks themselves.

From an architectural implementation perspective, DCRuntime can enhance the JAX+XLA+PJRT stack through strategic integration at multiple levels of the PJRT abstraction hierarchy, see Figure 3. This integration would enable DCRuntime's push-based data orchestration while preserving PJRT's clean hardware abstraction model: At the client level, DCRuntime would extend PjRtClient to implement global resource coordination across multiple JAX applications. This extended client would function as a multi-tenant resource manager that maintains awareness of all active workloads in the system. When implemented, this component would intercept client operations like device enumeration, buffer creation, and execution requests, augmenting them with DCRuntime's global scheduling logic. The client extension would communicate with DCRuntime's fault-tolerant coordinators to maintain system-wide metadata and manage resource allocation policies across applications, enabling true multi-tenancy without requiring application-level coordination.

For memory management, DCRuntime would implement specialized versions of PjRtMemorySpace and PjRtBuffer that interface with its distributed data streams abstraction. These components would transform PJRT's default pull-based memory model by integrating with CPU-based stream brokers that proactively prefetch data from storage into appropriate memory locations. The implementation would leverage DCRuntime's push-based architecture to anticipate data needs based on analysis of computational patterns across applications, ensuring buffers are populated before computation begins. When JAX requests data transfers through PJRT APIs, DCRuntime-enhanced buffers would often have the data already positioned, eliminating transfer latency at computation time.

The execution layer integration would modify how PjRtLoadedExecutable instances schedule work on devices. DCRuntime would enhance PJRT's execution methods to coordinate with its GPU cache components, which schedule I/O and compute operations across the hardware. This integration would transform PJRT's async computation model to fully leverage DCRuntime's distributed

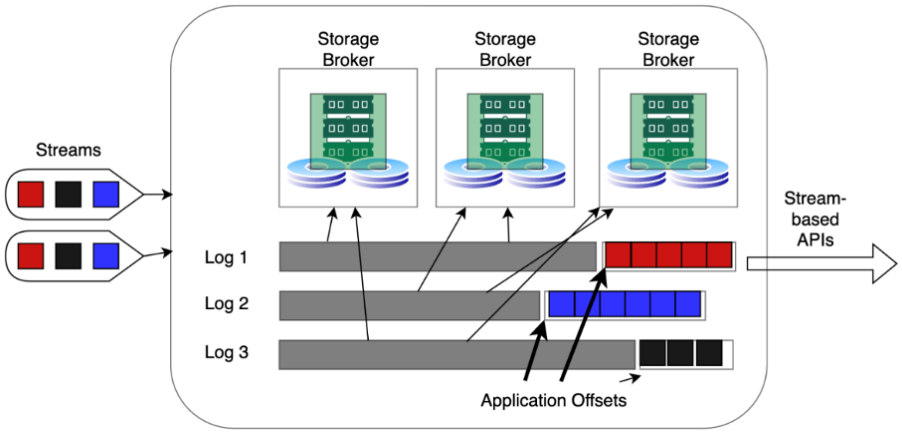

Fig. 4: Partial fault recovery with push-based offsets in DCRuntime.

compute streams, enabling asynchronous pipelining of computations and I/O across multiple GPU devices. The implementation would extend PjRtFutures to work with DCRuntime's orchestration layer, maintaining appropriate synchronization while maximizing parallel execution opportunities across the entire resource pool.

Through these architectural enhancements, DCRuntime would address key limitations in current JAX+XLA deployments while preserving the programming model that developers rely on. Applications would continue to use standard JAX APIs, but would benefit from potentially improved resource utilization, multi-tenant efficiency, and dynamic scaling capabilities without requiring code modifications.

As shown in Figure 4, a potential implementation of DCRuntime will support *partial recovery* by leveraging log-structured in-memory storage [32] and push-based data movement, potentially using RDMA technology [22]. Our novel approach to fast crash recovery in unified, in-memory, log-structured storage (e.g., [27]) hinges on DCRuntime's push-based model, enabling recovery from the most recent consumer/producer offsets. This design allows for faster recovery compared to traditional methods that require full log recovery. By focusing solely on the relevant application stream offsets, DCRuntime substantially reduces the overhead associated with restoring large-scale applications following a crash.

DCRuntime's architecture integrates three interacting components: fault-tolerant coordinators orchestrating metadata and system state, CPU-based stream brokers proactively prefetching and then pushing data from underlying filesystem storage into stream buffers, and GPU-based application stream components managing memory allocation and access for CPU-GPU. Applications will directly or transparently through runtime use DCRuntime APIs to create and manage data and compute streams, with coordinators orchestrating broker ac-

tions. DCRuntime's global view of data and compute resources enables more informed data partitioning strategies to efficiently utilize multiple GPUs. Its push-based model facilitates proactive caching, reducing application delays by anticipating computational needs. Furthermore, a significant aspect involves designing and developing a push-based approach for seamless CPU-to-CPU-GPU node integration, balancing trade-offs in areas such as availability, partitioning granularity, performance overhead, and fault tolerance mechanisms. Unlike traditional systems, DCRuntime leverages data stream access patterns, specifically tracked consumed data stream offsets, to prioritize recovery.In the event of a crash, it resumes from the application's last known progress point, enabling faster restarts by avoiding the overhead of full log recovery.

Key challenges addressed by DCRuntime's architecture include finding appropriate data partitioning and program parallelism mechanisms to efficiently feed multiple GPUs. DCRuntime's global view of data streams and compute resources allows for more informed partitioning strategies. Another challenge is determining how to cache datasets to prevent application delays. The push-based model enables proactive caching based on anticipated computational needs.

Current storage and processing systems often handle recovery in isolation, lacking the crucial application-level insights needed for effective recovery prioritization. DCRuntime directly addresses this limitation by leveraging data stream access patterns, particularly the application's consumed offsets, which are tracked and managed by the runtime. Through push-based consumption, DCRuntime gains knowledge of the application's progress. Upon a crash, DCRuntime immediately prioritizes recovering logs starting from the application's last known consumed offsets, enabling *partial recovery*. This allows for rapid access to the critical data needed for restarting the application and significantly reduces downtime compared to traditional systems that execute full log recovery, incurring substantial overhead and unnecessarily delaying the application restart.

6 Related Work

DCRuntime can serve as a storage and compute runtime foundation for multi-tenant and collaborative data-intensive systems. Beyond direct application use, the DCRuntime runtime API holds the potential to serve as a foundational layer for other data-intensive frameworks. For instance, frameworks like JAX and OpenXLA could leverage DCRuntime to manage the underlying data movement and distribution of large tensors across the distributed hardware, potentially simplifying the integration of accelerators and improving data locality. Similarly, Apache Spark could utilize DCRuntime's stream and compute abstractions to implement more efficient and lower-latency shuffle operations, or to manage the persistence and access of RDD partitions in a more fine-grained manner. Spark can share (although not efficiently) RDDs via Apache Ignite [1], another JVM-based in-memory data platform. By providing a unified and actively managed data layer, DCRuntime could simplify the development and deployment of complex data-intensive applications built on top of these higher-level frameworks.

Enabling the unified DCRuntime architecture requires the integration of several functional components. Data Ingestion acquires, buffers, and temporarily stores fast data streams and raw file data in memory. Data storage (persistent/caching) [14] ensures durability, availability, and fault tolerance. Big data processing [9] and ML/AI analytics [40, 2] further enable efficient data stream consumption by ML/AI applications.

DCRuntime distinguishes itself from existing work in several ways. Unlike Ray, whose object references are tied to specific tasks which limits interoperability [37], DCRuntime offers globally managed compute streams. Chapel [8] lacks native GPU support and fine-grained control over overlapping data transfers and computations, both of which are central to DCRuntime's design. While monolithic architectures [41] achieve tighter integration, DCRuntime strikes a balance between modularity and cross-layer optimization. Moreover, unlike systems that handle static streams (e.g., Apache Kafka [35, 31]), DCRuntime provides application engines with direct, dynamic access to stream buffers.

Apache Kafka [35, 31], a CPU-only cloud stream storage solution, will not scale at HPC exascale and it requires time-consuming and costly manual data re-partitioning, lacking support for partial recovery. DCRuntime could build over our in-memory storage system KerA [26, 24, 27, 25], leveraging its dynamic partitioning and push-based streaming integration, although it currently lacks support for GPUs as argued in this paper.

Although fault-tolerant storage systems typically fully recover crashed nodes [30], they often do so without considering application-specific requirements [23]. Notably, no existing system fully implements the DCRuntime active data movement approach. As a result, efforts to mitigate stragglers and multi-application I/O interference often rely on resource-intensive monitoring tools that introduce considerable overhead. Although DCRuntime uses CPU memory to manage metadata and orchestrate data movement, the potential performance gains and reduced operational costs associated with its active approach can justify the additional CPU memory overhead (to be explored).

Prior work advocates for a push-based streaming model [25] across the computing continuum, and recent research on data flow in modern hardware [21] also supports the concept of stream processing across the entire architecture. While their focus is primarily on reducing data movement (and thus orthogonal to ours), DCRuntime takes a different data movement approach to integrate seamlessly with and enhance existing processing engines. To ensure DCRuntime's correctness and robustness (e.g., [15]), we plan to use a holistic design approach using TLA+ [19, 6], a language specifically designed for specifying and verifying concurrent and distributed systems.

Exascale computing provides significant computational resources, yet I/O bottlenecks remain a critical challenge. Even with large-scale hardware deployments, such as 16000 H100 GPUs used for LLAMA 3.1 training (where Model FLOPs Utilization is only around 38–43% [MFU]), maximizing performance demands a fundamental shift in the way we manage data and computations. Solutions like ADIOS2 [13] offer improvements, but they serve more as intermediate

steps. Although ADIOS2 is a powerful embeddable I/O library, it explicitly states that ADIOS2 is not a Memory Manager library. It optimizes data staging and transfer but leaves memory management to the application, limiting its ability to address data movement challenges at exascale comprehensively. It is essential to move beyond solely optimizing I/O operations and focus on actively managing global memory.

The implementation of distributed futures in Ray [37, 36] extends RPC mechanisms to efficiently manage parallelism and data movement on behalf of applications, enabling support for fine-grained tasks that execute in milliseconds. These distributed futures closely resemble distributed procedure calls [20], as their references function similarly to handles. However, DCRuntime advances this concept by tightly integrating data movement and computation within a unified, push-based runtime. Unlike Ray's task-centric approach, DCRuntime's global compute and data streams provide seamless scalability across heterogeneous hardware while minimizing overhead. This integration allows DCRuntime to proactively manage data placement and computation scheduling, further reducing latency and improving performance for data-intensive workloads.

7 Conclusion

DCRuntime introduces a unified, push-based runtime vision to integrate data movement and computation across heterogeneous systems. By abstracting memory and compute management through global data and compute streams, DCRuntime aims to optimize resource utilization, to reduce GPU idle time, and to optimize fault tolerance. Beyond addressing scalability and performance challenges at exascale, DCRuntime opens exciting new research directions in dynamic resource management, cross-layer optimization, and multi-tenant system design. We invite the community to engage in exploring and advancing this vision for the future of data-intensive computing.

Acknowledgments. This work is partially funded by the SnT-LuxProvide partnership on bridging clouds and supercomputers and by the Fonds National de la Recherche Luxembourg (FNR) POLLUX program under the SERENITY Project (ref.C22/IS/17395419).

References

1. Apache ignite: A memory-centric distributed database, caching, and processing platform. https://ignite.apache.org/ (2025), accessed: 2025-01-12
2. Abadi, M.e.a.: Tensorflow: A system for large-scale machine learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. p. 265–283. OSDI'16, USENIX Association, USA (2016)
3. et al., A.D.: The llama 3 herd of models (2024), https://arxiv.org/abs/2407.21783
4. Andersch, M., Palmer, G., Krashinsky, R., Stam, N., Mehta, V., Brito, G., Ramaswamy, S.: Nvidia hopper architecture in-depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/ (Mar 2022), accessed: 2025-01-12

5. Anneser, C., Vogel, L., Gruber, F., Bandle, M., Giceva, J.: Programming fully disaggregated systems. In: Proceedings of the 19th Workshop on Hot Topics in Operating Systems. p. 188–195. HOTOS '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3593856.3595889, https://doi.org/10.1145/3593856.3595889
6. Batson, B., Lamport, L.: High-level specifications: Lessons from industry. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects. pp. 242–261. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
7. Ben David, S.: Why a data plane architecture is critical for optimizing next-generation workloads. In: Proceedings of the 2022 Workshop on Emerging Open Storage Systems and Solutions for Data Intensive Computing. p. 9. EMOSS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3526061.3532101
8. Callahan, D., Chamberlain, B., Zima, H.: The cascade high productivity language. In: Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. pp. 52–60 (2004). https://doi.org/10.1109/HIPS.2004.1299190
9. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (jan 2008). https://doi.org/10.1145/1327452.1327492
10. Dryden, N., Maruyama, N., Moon, T., Benson, T., Yoo, A., Snir, M., Van Essen, B.: Aluminum: An asynchronous, gpu-aware communication library optimized for large-scale training of deep neural networks on hpc systems. In: 2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC). pp. 1–13 (2018). https://doi.org/10.1109/MLHPC.2018.8638639
11. Frostig, R., Johnson, M., Leary, C.: Compiling machine learning programs via high-level tracing. In: SysML Conference 2018 (2018), https://mlsys.org/Conferences/doc/2018/146.pdf
12. Ghemawat, S., Grandl, R., Petrovic, S., Whittaker, M., Patel, P., Posva, I., Vahdat, A.: Towards modern development of cloud applications. In: Proceedings of the 19th Workshop on Hot Topics in Operating Systems. p. 110–117. HOTOS '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3593856.3595909, https://doi.org/10.1145/3593856.3595909
13. Godoy, W.F., Podhorszki, N., Wang, R., Atkins, C., Eisenhauer, G., Gu, J., Davis, P., Choi, J., Germaschewski, K., Huck, K., Huebl, A., Kim, M., Kress, J., Kurc, T., Liu, Q., Logan, J., Mehta, K., Ostrouchov, G., Parashar, M., Poeschel, F., Pugmire, D., Suchyta, E., Takahashi, K., Thompson, N., Tsutsumi, S., Wan, L., Wolf, M., Wu, K., Klasky, S.: Adios 2: The adaptable input output system. a framework for high-performance data management. SoftwareX **12**, 100561 (2020). https://doi.org/https://doi.org/10.1016/j.softx.2020.100561, https://www.sciencedirect.com/science/article/pii/S2352711019302560
14. Hennessy, J.L., Patterson, D.A.: Computer Architecture, Sixth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edn. (2017)
15. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (jul 1990). https://doi.org/10.1145/78969.78972, https://doi.org/10.1145/78969.78972

16. for High Performance Computing (ETP4HPC), T.E.T.P.: Strategic research agenda for hpc in europe (2022), https://www.etp4hpc.eu/pujades/files/ETP4HPC-SRA5_2022_web.pdf
17. Hu, C., Wang, C., Wang, S., Sun, N., Bao, Y., Zhao, J., Kashyap, S., Zuo, P., Chen, X., Xu, L., Zhang, Q., Feng, H., Shan, Y.: Skadi: Building a distributed runtime for data systems in disaggregated data centers. In: Proceedings of the 19th Workshop on Hot Topics in Operating Systems. p. 94–102. HOTOS '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3593856.3595897, https://doi.org/10.1145/3593856.3595897
18. Kirk, D.: Nvidia cuda software and gpu parallel computing architecture. In: Proceedings of the 6th International Symposium on Memory Management. p. 103–104. ISMM '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1296907.1296909, https://doi.org/10.1145/1296907.1296909
19. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (may 1994). https://doi.org/10.1145/177492.177726
20. Lee, C.S.T.: DISTRIBUTED PROCEDURE CALL. Ph.D. thesis, Stanford University (June 2021), https://web.stanford.edu/ ouster/cgi-bin/papers/LeePhD.pdf
21. Lerner, A., Alonso, G.: Data flow architectures for data processing on modern hardware. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE) (May 2024), presented at the Data Engineering Future Technologies Track
22. Ma, S., Ma, T., Chen, K., Wu, Y.: A survey of storage systems in the rdma era. IEEE Trans. Parallel Distrib. Syst. **33**(12), 4395–4409 (dec 2022). https://doi.org/10.1109/TPDS.2022.3188656
23. Magalhaes, A., Monteiro, J.M., Brayner, A.: Main memory database recovery: A survey **54**(2) (mar 2021). https://doi.org/10.1145/3442197
24. Marcu, O.C.: KerA: A Unified Ingestion and Storage System for Scalable Big Data Processing. Theses, INSA Rennes (Dec 2018), https://theses.hal.science/tel-01972280
25. Marcu, O.C., Bouvry, P.: In support of push-based streaming for the computing continuum. In: 15th Asian Conference on Intelligent Information and Database Systems. Phuket, Thailand (Jul 2023)
26. Marcu, O.C., Costan, A., Antoniu, G., Pérez-Hernández, M., Nicolae, B., Tudoran, R., Bortoli, S.: Kera: Scalable data ingestion for stream processing. In: 2018 IEEE 38th ICDCS. pp. 1480–1485. IEEE (2018)
27. Marcu, O.C., Costan, A., Nicolae, B., Antoniu, G.: Virtual Log-Structured Storage for High-Performance Streaming. In: Cluster 2021 - IEEE International Conference on Cluster Computing. pp. 1–11. Portland / Virtual, United States (Sep 2021)
28. Maurya, A., Underwood, R., Rafique, M.M., Cappello, F., Nicolae, B.: DataStates-LLM: Lazy Asynchronous Checkpointing for Large Language Models. In: HPDC'24: 33nd International Symposium on High-Performance Parallel and Distributed Computing. Pisa (IT), Italy (Jun 2024). https://doi.org/10.1145/3625549.3658685, https://hal.science/hal-04614247
29. Nicolae, B.: DataStates: Towards Lightweight Data Models for Deep Learning. In: SMC'20: The 2020 Smoky Mountains Computational Sciences and Engineering Conference. Nashville (virtual conference), United States (Aug 2020), https://hal.science/hal-02941295
30. Ongaro, D., Rumble, S.M., Stutsman, R., Ousterhout, J., Rosenblum, M.: Fast crash recovery in ramcloud. In: Proceedings of the Twenty-Third ACM Symposium

on Operating Systems Principles. p. 29–41. SOSP '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2043556.2043560, https://doi.org/10.1145/2043556.2043560
31. Povzner, A.e.a.: Kora: A cloud-native event streaming platform for kafka. Proc. VLDB Endow. **16**(12), 3822–3834 (aug 2023). https://doi.org/10.14778/3611540.3611567, https://doi.org/10.14778/3611540.3611567
32. Rumble, S.M., Kejriwal, A., Ousterhout, J.: Log-structured memory for dram-based storage. In: Proceedings of the 12th USENIX FAST. p. 1–16. FAST'14, USENIX Association, USA (2014)
33. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.m.W.: Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. p. 73–82. PPoPP '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1345206.1345220, https://doi.org/10.1145/1345206.1345220
34. Strati, F., Ma, X., Klimovic, A.: Orion: Interference-aware, fine-grained gpu sharing for ml applications. In: Proceedings of the Nineteenth European Conference on Computer Systems. p. 1075–1092. EuroSys '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3627703.3629578, https://doi.org/10.1145/3627703.3629578
35. Wang, G.e.a.: Consistency and completeness: Rethinking distributed stream processing in apache kafka. In: Proceedings of the 2021 International Conference on Management of Data. p. 2602–2613. SIGMOD '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3448016.3457556
36. Wang, S., Hindman, B., Stoica, I.: In reference to rpc: it's time to add distributed memory. In: Proceedings of the Workshop on Hot Topics in Operating Systems. p. 191–198. HotOS '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3458336.3465302, https://doi.org/10.1145/3458336.3465302
37. Wang, S., Liang, E., Oakes, E., Hindman, B., Luan, F.S., Cheng, A., Stoica, I.: Ownership: A distributed futures system for Fine-Grained tasks. In: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). pp. 671–686. USENIX Association (Apr 2021), https://www.usenix.org/conference/nsdi21/presentation/cheng
38. Xue, Y., Liu, Y., Nai, L., Huang, J.: V10: Hardware-assisted npu multi-tenancy for improved resource utilization and fairness. In: Proceedings of the 50th Annual International Symposium on Computer Architecture. ISCA '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3579371.3589059, https://doi.org/10.1145/3579371.3589059
39. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing. p. 10. HotCloud'10, USENIX Association, USA (2010)
40. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: Fault-tolerant streaming computation at scale. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. p. 423–438. SOSP '13, Association for Computing Machin-

ery, New York, NY, USA (2013). https://doi.org/10.1145/2517349.2522737, https://doi.org/10.1145/2517349.2522737

41. Zou, J., Iyengar, A., Jermaine, C.: Pangea: Monolithic distributed storage for data analytics. Proc. VLDB Endow. **12**(6), 681–694 (feb 2019). https://doi.org/10.14778/3311880.3311885, https://doi.org/10.14778/3311880.3311885