Lineage-based Reuse and Memory Management for Multi-backend ML Systems

Arnab Phani
TU Berlin & BIFOLD
Germany
arnab.phani@tu-berlin.de

Matthias Boehm
TU Berlin & BIFOLD
Germany
matthias.boehm@tu-berlin.de

ABSTRACT

Modern machine learning (ML) systems leverage multiple backends, including CPUs, GPUs, and distributed execution platforms like Apache Spark or Ray. Depending on workload and cluster characteristics, these systems typically compile an ML pipeline into hybrid plans of in-memory CPU, GPU, and distributed operations. Prior work found that exploratory data science processes exhibit a high degree of redundancy, and accordingly applied tailor-made techniques for reusing intermediates in specific backend scenarios. However, achieving efficient holistic reuse in multi-backend data systems remains a challenge due to its tight coupling with other aspects such as memory management, data exchange, and operator scheduling. In this paper, we introduce MEMPHIS, a principled framework for holistic, application-agnostic, multi-backend reuse and memory management. MEMPHIS's core component is a hierarchical lineage-based reuse cache, which acts as a unified abstraction and manages the reuse, recycling, exchange, and cache eviction across different backends. To address challenges of different backends such as lazy evaluation, asynchronous execution, memory allocation overheads, small available memory, and different interconnect bandwidths, we devise a suite of cache management policies. Moreover, we extend an optimizing ML system compiler by special operators and rewrites for asynchronous data exchange, workload-aware speculative cache management, and related operator ordering for concurrent execution. Our experiments across diverse ML tasks and pipelines show improvements of up to 9.6x compared to state-of-the-art ML systems.

1. INTRODUCTION

Modern ML systems are widely used for model training, inference, as well as data preparation and feature transformations of multi-modal data [70]. Data scientists hierarchically compose complex ML pipelines from black-box primitives [15]. The exploratory nature of these pipelines causes high computational redundancy [71, 87, 23, 39].

Sources of Redundancy: This high computational redundancy arises from various sources including incremental modifications of ML pipelines and hyper-parameter tuning,

 DOI: https://doi.org/10.48786/edbt.2025.21

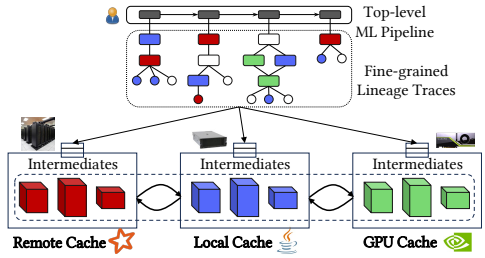


Figure 1: Multi-backend Reuse Cache for Intermediates.

as well as fine-grained redundancy in training and inference[71]. ML libraries construct high-level primitives from smaller built-ins (e.g., scikit-learn's `make_pipeline`), hiding the independent operations, causing unnecessary redundancy. AutoML systems [46, 22, 28, 49, 78] and ML agents [61, 27] combine tasks like data cleaning, feature engineering, hyper-parameter tuning, and model training [45, 44], and explore pipelines with slight changes. Deep neural network (DNN) workloads repeatedly execute data pipelines [59, 55, 33] and forward paths [60, 73] at batch granularity, while inference frameworks for object detection and machine translation [17] encounter duplicate inputs [19, 43, 86].

Multi-backend ML Systems: ML pipelines with diverse data and workload characteristics necessitate multiple, specialized backends for achieving efficiency and scalability. Example pipelines include (1) hybrid local/distributed runtime plans for large-scale ML [89, 14, 51, 74], (2) large-scale data validation [76, 72] and cleaning [78, 79], (3) sampling, feature selection [15], and data augmentation [20, 7], which iteratively change data sizes by orders of magnitude, (4) exploratory ML algorithm research [52], (5) model training on structured and unstructured features [60], (6) AutoML systems [14, 15, 46] supporting diverse ML algorithms, and (7) model debugging [48, 75] with configuration-dependent intermediate sizes. These tasks are often combined into complex pipelines, increasingly fostering the development of ML systems with optimizing compilers, specialized operator placement strategies [15, 46, 12], and multiple backends including local CPU/GPU/FPGAs, distributed MapReduce/Spark/Dask/Ray and federated backends [41, 10]. Example systems include PyTorch [64] and TensorFlow [1] (which leverage CPU, GPU), MLlib [89] and Dask-ML [74] (which utilize local and distributed), and SageMaker [46] (which utilizes CPU, GPU, and Spark). Additionally, unified data analytics frameworks [12, 24, 31], polyglot [3, 30, 29], federated [41, 10], and composable [56] data management systems support cross-platform runtime backends.

Challenges in Multi-backend Reuse: Introducing a static reuse cache for redundancy elimination into multi-backend systems—as shown in Figure 1—poses major challenges due to heterogeneous backend characteristics. These backends differ in execution models (eager, lazy, asynchronous), memory characteristics (large distributed, small on-chip), data exchange bandwidths, other backend-specific properties like GPU memory allocation overhead, and target workloads, ranging from pre-processing to mini-batch DNNs and LLMs. To address this diversity, modern data systems employ various techniques for memory management [67, 88, 4, 37], operator placement [53, 8, 9], data exchange [59, 33, 91, 83, 38], and parallelization [92, 26, 62, 77] tailored to specific workloads and backends. This heterogeneity necessitates a principled approach for efficient reuse and cache integration across diverse compilation, memory management, and operator scheduling techniques, which is currently lacking.

Existing Work on Reuse: There exists extensive research on reusing query intermediates in database systems [36, 93]. In ML, prior work on reuse rely on coarse-grained reuse of the top-level primitives (see Figure 1) [87, 23, 85, 90, 81, 39]. However, this black-box view of individual ML stages fails to handle the ubiquitous fine-grained redundancy (e.g., repeated matrix multiplications inside/across primitives) and non-determinism (randomized primitives). The LIMA framework [71] introduced fine-grained reuse, leveraging lineage traces on individual operations and functions to uniquely identify reusable intermediates, but was limited to local CPU operations. Prior work on application-specific, multi-backend reuse includes heuristics-based Spark RDD caching [14, 6, 33], input data pipeline reuse [59, 33, 55], prediction caching [19, 43], and GPU-CPU activation offloading for DNNs [66, 35, 50]. We refer the interested reader to a broader discussion of related work [69]. These approaches are tailored to workload-backend combinations, and fail to eliminate redundancy in modern data-centric ML pipelines.

MEMPHIS Overview and Contributions: We introduce MEMPHIS, a *holistic framework* for multi-backend reuse of intermediates and memory management. Key principles are (1) unified cache abstraction with multi-backend data objects, (2) backend-specific cache and memory management, and (3) robust integration with the compiler and runtime to support diverse workloads. MEMPHIS is fully integrated into Apache SystemDS[1] [15], and utilizes *three* representative backends: in-memory operations (SystemDS), distributed operations (Spark), and GPUs, while being applicable to diverse system architectures. MEMPHIS extends LIMA's [71] lineage-based reuse framework with novel compiler and runtime techniques for reusing Spark and GPU intermediates, handling Spark's lazy evaluation and small GPU memory. The introduced techniques also generalize to reusing intermediates in column-at-a-time or vector-at-a-time query processing with multiple backends. In the following, we discuss the relevant background, introduce our multi-backend lineage tracing framework with runtime cache management and compiler optimizations, and discuss the generalizability of these techniques across workloads and systems.

2. BACKGROUND

This section describes the necessary background of ML system internals, Spark and GPU backends, their execution models and memory management. The CPU, GPU, and Spark backends differ substantially in their execution model, memory management, and target workloads.

Program/DAG Compilation: We categorize the optimization scope of ML systems into three types [16]. *Eager execution* libraries like NumPy [84] and Scikit-learn [65] execute operations immediately and rely on Python for control flow. *DAG compilation* systems like TensorFlow [1, 11] and PyTorch [64] perform lazy evaluation over a compute DAG, enabling a larger optimization scope. Finally, *program compilation* in Julia [13] and SystemDS [15] (previously SystemML [14]) compile scripts into program block hierarchies, where last-level blocks comprise operator DAGs. Compiler rewrites like common subexpression elimination (CSE) and code motion fail to eliminate all redundancy, as conditional control flow is often unpredictable at compile time.

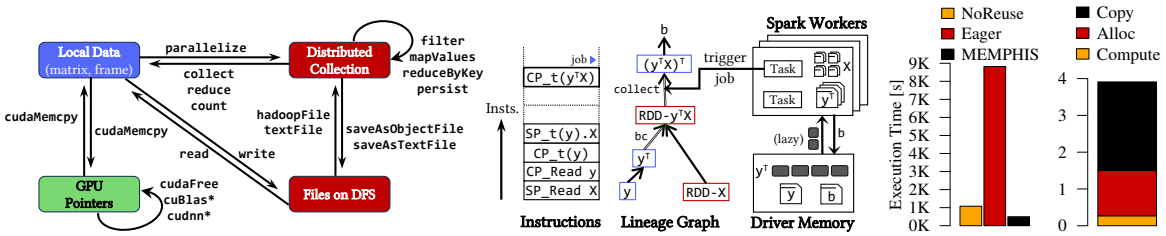
Operator Scheduling: An operator scheduler converts operator DAGs into backend-specific instruction streams and has two key responsibilities: (1) *operator placement*, which minimizes execution time in multi-backend runtimes using heuristics or configurations, and (2) *operator linearization*, which orders operator DAGs into sequential or parallel instruction streams, affecting parallelism and memory requirements. SystemDS linearizes the DAGs depth-first and places operations with higher memory estimates to Spark and compute-intensive, dense operations to GPU. Figure 2(a) shows the lifecycle of a data object across backends.

Spark's Execution and Memory Model: Spark follows a driver–executor architecture: the driver manages scheduling and local execution, while executors run distributed tasks. Computation is expressed using RDDs (Resilient Distributed Datasets) [89], representing partitioned collections of matrix/frame tiles manipulated through lazily evaluated *transformations*. An *action* invokes the `DAGScheduler` to construct a job DAG of stages separated by shuffle boundaries. Spark's unified memory model divides heap space into execution and storage (default 60%), allowing dynamic sharing between computation, temporary data, caching, and broadcasting. Shuffle outputs and broadcast data are implicitly cached, while explicit `persist()` enables reuse across jobs at different storage levels. Under memory pressure, cached partitions are evicted and recomputed from lineage.

Lazy Evaluation and Broadcast Challenges: Lazy evaluation presents several challenges for efficient reuse. First, eager caching (LIMA, tf.Data [59], Cachew) materializes RDDs after each instruction, triggering repeated jobs. In Figure 2(c), materializing 12K RDDs (4K reusable) is 10x slower than no caching. Second, caching all RDDs at runtime inflates cluster memory (20x in Figure 2(c)), requiring speculative caching. Finally, although broadcast operators are cheaper than shuffle joins, Spark's lazy execution delays data transfer: `TorrentBroadcast` splits serialized data into 4 MB chunks stored in the driver and lazily distributed to executors. Consequently, broadcast data and dependent RDDs must be retained until job completion, creating *dangling references* that consume memory. Figure 2(b) shows linearized instructions (`SP` denotes Spark, `CP` denotes CPU) and the lineage graph for $b = (\mathbf{y}^\top \mathbf{X})^\top$, broadcasting $\mathbf{y}^\top$. The second transpose collects the vector $\mathbf{b}$ to the driver. The serialized $\mathbf{y}^\top$ remains in the driver's memory until the job finishes, which is dependent on other operators' linearization order.

GPU Execution Model and Memory Challenges: GPUs offer high peak performance and memory bandwidth, suit-

[1]SystemDS: https://github.com/apache/systemds



(a) Data Object Lifecycle (b) Spark Broadcast (c) RDD Reuse (d) Timings

Figure 2: Spark and GPU Backend Details.

able for regular data access (e.g., DNNs). In SystemDS, CUDA kernels execute eagerly and sequentially within a stream but asynchronously with respect to the host, enabling CPU–GPU overlap. However, device–host transfers and memory deallocation introduce synchronization barriers that stall the CPU until pending kernels complete. Traditional eager caching also forces synchronization and degrades performance. As shown in Figure 2(d), for an affine layer with ReLU over 10 epochs of 1K mini-batches (128 rows), memory allocation/free and data transfer incur 4.6x and 9x higher overhead than computation, highlighting that static cache memory and traditional eviction policies, which increase memory pressure and data copy overhead, are unsuitable for GPU pointer caching.

3. HIERARCHICAL LINEAGE CACHE

The basic architecture of MEMPHIS with lineage tracing and a hierarchical cache is shown in Figure 5. The driver/host process handles the lineage tracing, compiler optimizations, and eviction planning, whereas the actual cached objects reside in the respective backends. MEMPHIS builds upon LIMA's [71] basic lineage tracing (eager caching), and extends it with a hierarchical lineage cache and unified API for reuse across local, Spark, and GPU. This section describes our fine-grained lineage tracing, and the unified intermediate cache for backend-specific objects.

3.1 Supported API

MEMPHIS provides a set of system-internal methods to enable lineage-based reuse and simplify integration with ML systems across different backends, compilation, and operator placement. Specifically, `TRACE(inst)` captures operator lineage; `SERIALIZE(trace)` and `DESERIALIZE(log)` convert between in-memory lineage traces and lineage logs; `RECOMPUTE(log)` reproduces results from a log; `REUSE(trace)` skips execution by reusing cached results when available; `PUT(trace, object)` inserts instruction outputs into the backend-specific cache; and `MAKE_SPACE(object)` evicts cached objects to free backend memory.

3.2 Backend-agnostic Lineage Tracing

A lineage trace—incrementally built at runtime—is a DAG with nodes and edges representing operations and data dependencies. We invoke `TRACE` before executing each linear algebra instruction (see Figure 3) and maintain a hash map (`LineageMap`) that maps live variable names to lineage DAGs. Each lineage item captures the opcode, data items, and pointers to input lineage items, and every `TRACE` call creates a new item that is added to the map. To enable efficient reuse, lineage items implement `hashCode` and `equals`, where hashes are computed from input item hashes, opcodes, and data items, and equality checks use a non-recursive, queue-based traversal with sub-DAG memoization and early aborts. We further support serializing lineage traces and exact recomputation via `RECOMPUTE`, which re-applies the full compilation chain to regenerate identical results. Beyond reuse, fine-grained lineage tracing and these APIs provide a general infrastructure for debugging, interpretability, recomputation, model versioning, and lifecycle management, and integrate naturally with diverse data systems.

3.3 Multi-backend Lineage Cache

We leverage that our lineage uniquely identifies intermediates to enable reuse of previously computed results. The lineage cache serves as a repository for these lineage traces, maintaining the necessary data structures to efficiently map them to their corresponding backend-specific data objects.

Lineage Cache: The lineage cache is a hash map that maps lineage items to cached data objects (see Figure 5 middle). Figure 3 shows the pseudo-code of the reuse logic integrated in the main instruction execution path, which seamlessly applies to all instructions. We call `REUSE` for each instruction. If the output exists in the cache, we reuse the data object, assign it to the live variable, and skip the instruction. Otherwise, we execute the instruction and store the output in the cache via `PUT`. Additionally, the lineage cache entries hold metadata including compute costs, access counts, and status.

```
while (inst in insts)
  lt = TRACE (inst)
  if (!REUSE (lt))
    out = exec(inst)
    PUT (lt, out)
```

Figure 3: Reuse API.

Redundancy in Lineage Items: Generating a new lineage item for each instruction *before* reuse creates redundant lineage DAGs across `LineageMap` and lineage cache entries. To address this issue, upon successful probes, we replace the respective `LineageMap` entries with the lineage keys of cached objects. As Figure 4 shows, this compaction increases shared sub-DAGs (objects with identical references), which in turn improves probing efficiency and reduces the memory footprint.

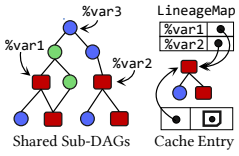


Figure 4: Compaction.

Backend-local Cached Objects: The lineage cache entries are wrappers around backend-specific pointers. These pointers refer to in-memory matrix blocks, scalars, distributed RDDs, GPU objects, and disk-evicted binaries. This design allows seamless data movement like broadcasting local inputs to remote backends, transferring Spark job results to

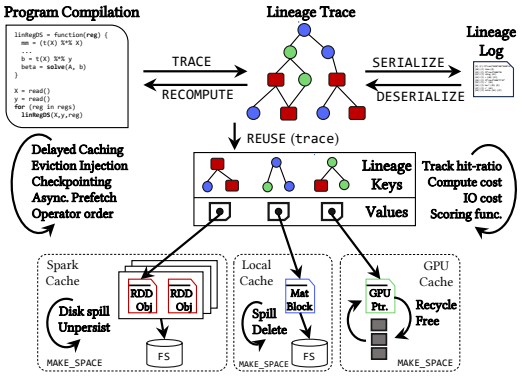


Figure 5: Hierarchical Lineage Cache & Reuse Overview.

the driver, and GPU-to-host copies. Moreover, the wrappers enable caching the same object in multiple backends (beneficial for data-local scheduling) and seamless extension to new backends. The centralized lineage tracing and reuse facilitates a reuse-aware compilation to increase reuse potential and reduce caching overheads. Figure 5 summarizes the lineage tracing lifecycle, reuse operations, backend-local evictions, and compiler extensions.

Multi-level Reuse: MEMPHIS reuses outputs of deterministic functions and basic blocks (code block w/o control flow) when called repeatedly with the same inputs, even when the inputs and outputs are scattered across different backends. We use a special lineage item containing the function name and inputs for each function output [71]. These items are managed alongside regular entries and are subject to eviction by the respective backends. Combining coarse-grained reuse (e.g., function reuse) with fine-grained reuse (e.g., operator-at-a-time reuse, as shown in Figure 3) is advantageous for hierarchically-composed data-centric ML pipelines. Multi-level reuse effectively reduces fine-grained remote operations, related data exchange, and cache pollution (avoids caching large distributed objects, or many small GPU objects) and hence is resilient against cache evictions.

4. RUNTIME MULTI-BACKEND REUSE

Backend-local caches require tailored cache management due to their diversity. This section describes reuse and memory management for Spark and GPU (more details [69]).

4.1 Reuse & Memory Management in Spark

We use Spark's caching API to cache RDDs in cluster memory. Furthermore, we introduce memory management and cache eviction tailored for Spark's lazy evaluation.

Reuse Spark Actions: Spark actions trigger job execution and return the results to the driver. ML systems implicitly leverage these actions within distributed physical operators (e.g., single-block aggregates calling `reduce()` instead of `reduceByKey()`). Additionally, operator placement decisions may explicitly move data to the driver using `collect()`. Before triggering a Spark job, we reuse the previously collected result if available in the driver's cache, and bypass the job execution. In Figure 6 (top entry), the `transpose` operator collects the vector $\mathbf{b}$ to the driver and stores it in the cache for future reuse. Reusing redundant actions in the driver eliminates distributed operations, unnecessary distributed caching, and data exchange such as shuffle and collect.

Reuse RDDs: RDDs are distributed data collections, which are cached and reused in the cluster. Before executing a Spark transformation, the `REUSE` API probes the lineage cache and reuses the RDD (serves as a pointer to the distributed collection) if a match is found. Otherwise, we mark the RDD for caching with `persist()` and store it in the cache along with related metadata. `persist` is a lazy operation—and thus, the cached RDD may not be materialized in Spark memory by the time of reuse. However, we reuse even unmaterialized RDDs to enhance compute-sharing across jobs, and enable shuffle-file reuse (data exchange optimization). Figure 6 (bottom entry) shows a cached RDD entry for $\mathbf{X}^\top\mathbf{X}$, where $\mathbf{X}$ is distributed.

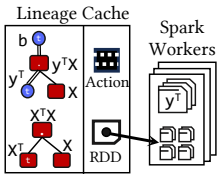


Figure 6: Spark Reuse.

Lazy Garbage Collection: To tackle memory overhead caused by dangling RDDs and broadcast references (Figure 2(b)), we track internal metadata for each RDD, including #child RDDs, #pending consumers, materialization status, and its memory overhead. During reuse, `MAKE_SPACE` recursively traverses child RDDs and broadcasts, updating the metadata and clearing stale RDDs (not in use and already materialized). For example, in Figure 6 (bottom), we delete $\mathbf{X}^\top$ and $\mathbf{X}$ once $\mathbf{X}^\top\mathbf{X}$ is materialized. We use Spark's `destroy` and `getRDDStorageInfo` for broadcast variable deletion and materialization checks. Furthermore, to mitigate caching overhead, the MEMPHIS compiler often defers RDD caching (*delayed caching* discussed in 5.2). Delayed caching along with Spark action reuse further increases dangling references. In Figure 6 (top), $\mathbf{y}^\top\mathbf{X}$ RDD caching is delayed and remains unmaterialized due to the reuse of the collected $\mathbf{b}$ at the driver. Such RDDs prevent the garbage collection of their child RDDs (i.e., $\mathbf{y}^\top$, $\mathbf{X}$). After k cache misses, we asynchronously trigger a Spark job (calling `count()`) to materialize such an RDD. Any subsequent reuse then cleans up the child RDDs. These lazy cleanups ensure periodic reclamation of driver and cluster memory without hindering reuse.

Example 1 (Grid Search Linear Regression). *We apply grid search hyper-parameter tuning on a direct-solve linear regression (`linRegDS`) with distributed features $\mathbf{X}$ and local responses $\mathbf{y}$. Figure 7 shows the `linRegDS` function, which is invoked for a list of regularization parameters (`reg`). The core operations $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$ are independent of `reg` and thus, reusable across calls. SystemDS compiles $\mathbf{X}^\top\mathbf{X}$ as a shuffle-based matrix multiplication and rewrites $\mathbf{X}^\top\mathbf{y}$ to $(\mathbf{y}^\top\mathbf{X})^\top$ for broadcast-based multiplication. The second transpose of $(\mathbf{y}^\top\mathbf{X})^\top$ and the `solve` trigger Spark jobs. Figure 7 shows the DAG for `linRegDS`. Rectangles represent distributed operations. Reusable distributed and local operations are colored red and blue, and data exchange is indicated by double lines. The first `linRegDS` call creates lineage cache entries for all operations and caches in-memory matrix outputs of local operations. The second transpose of $(\mathbf{y}^\top\mathbf{X})^\top$ caches the collected column vector $\mathbf{b}$ as an in-memory matrix in the driver. Our compiler enables delayed caching for RDDs, e.g., deferring caching until the second cache hit. Accordingly, the second `linRegDS` call marks the RDDs ($\mathbf{mm}$, $\mathbf{y}^\top\mathbf{X}$) for distributed caching, reuses the local operations and the collected $\mathbf{b}$. Reusing the*

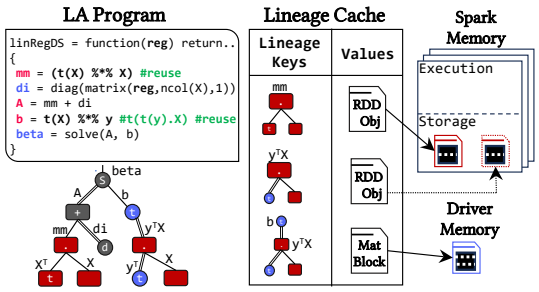


Figure 7: Spark Action and RDD Reuse Example.

Spark action result **b** *entirely eliminates the need to trigger the Spark job. However, not triggering the job keeps the* $\mathbf{y}^\top\mathbf{X}$ *RDD unmaterialized in the executors. The third call reuses the* **mm** *RDD, and subsequent calls clean up its child RDDs. After k (default three) cache misses for* $\mathbf{y}^\top\mathbf{X}$ *RDD, we asynchronously materialize the* $\mathbf{y}^\top\mathbf{X}$ *RDD (dotted line).*

4.2 Reuse & Memory Management in GPU

To manage limited GPU memory and allocation overhead, we combine reuse and recycling in a unified memory manager with moving boundaries that reuses `pointers`. Figure 8 depicts the memory management operations.

Live Variable Management: All pointers from allocation to deallocation are managed by the lineage cache. Figure 8(a) shows the lifecycle of a GPU pointer. We organize the allocated pointers in two lists: a `Live` and a `Free` list. The `Live` pointers correspond to variables which are still in use (i.e., pending consumers). The `Free` list comprises a hash map that maps sizes to a priority queues of free pointers of the respective sizes. Before executing an operation, we allocate memory for the output via `cudaMalloc` and place the allocated pointer in the `Live` list. After the last use, the pointers are moved to the `Free` list, as shown in Figure 8(b). This strategy of maintaining free memory pools benefits mini-batch processing with fixed batch sizes.

Reuse: Lineage cache entries encapsulate GPU pointers and metadata like data characteristics. Before executing an instruction (composed of one or more GPU kernels), the `REUSE` call reuses a cached output pointer, if available, and skips kernel launching. As Figure 8(c) shows, the reused pointer is moved from `Free` to `Live` list. Reusing a pointer may cause multiple variables referencing a single pointer. We track the *reference count* for each pointer indicating the #variables referencing it, and only when the reference count becomes zero, the pointer is returned to the `Free` list.

Memory Recycling: Pointers in the `Free` queues are ordered by a scoring function. Once the GPU memory is full, we start recycling the free pointers as a form of eviction (Figure 8(d)). We first look for a free pointer with the exact size to recycle. If unavailable, we free a pointer just larger than the required size. If all free pointers are smaller than the required size, we repeatedly free a pointer until `cudaMalloc` is successful. If allocation still fails, we clear all free pointers. Even then, the allocation may fail due to many live variables and memory fragmentation. In such cases, we initiate the device-to-host eviction process, and finally a full defragmentation, but this event is rare in practice. Memory recycling benefits typical DNN workloads with repeated operations on

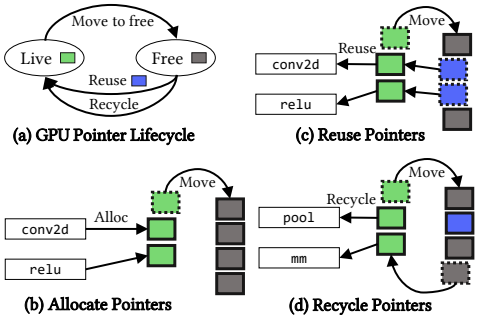


Figure 8: Reusing and Recycling of GPU Pointers.

fixed-sized intermediates, such as the forward and backward passes of mini-batch processing. The primary objective of these steps is to avoid memory allocation, deallocation, and fragmentation, without compromising the reuse potential.

4.3 Cache Eviction

We employ backend-specific, cost-based eviction policies to remove intermediates with low reuse potential. For Spark, we extend the Cost&Size policy [23, 71] to account for lazy evaluation by evicting cached RDDs, which is orthogonal to Spark's partition-level eviction. We heuristically allocate 80% (configurable) of Spark's storage memory for reuse, reserving the remainder for broadcast variables and compiler-placed checkpoints (Section 5.2). Each cached RDD o is ranked using analytical compute cost $c(o)$, estimated worst-case size $s(o)$, and reuse statistics (#hits r_h, #misses r_m, #jobs r_j), with the eviction scoring function

$$\arg\min_{o\in\mathbf{Q}} \quad (r_h(o) + r_m(o) + r_j(o)) \cdot c(o)/s(o), \qquad (1)$$

where $\mathbf{Q}$ is a priority queue of cached RDDs. The `MAKE_SPACE` method marks RDDs for asynchronous eviction and refreshes cache metadata using `getRDDStorageInfo`, with temporary overflows efficiently handled via Spark's partition spilling. For GPUs, the eviction policy determines the order in which pointers are recycled or freed from free queues and is tailored to mini-batch workloads. The eviction scoring function for a cached GPU object o is

$$\arg\min_{o\in\mathbf{Q}} \quad T_a(o) + 1/h(o) + c(o) \qquad (2)$$

where $T_a(o)$ denotes the normalized last access time to preserve recently reused pointers, $h(o)$ is the lineage DAG height favoring reuse of *input data pipelines* [55], and $c(o)$ is the estimated compute cost, prioritizing recycling of inexpensive intermediates (e.g., `ReLU` before `Conv2d`).

5. COMPILER INTEGRATION

Runtime-level reuse and memory management are valuable, but holistic *compiler- and runtime-level* handling yields additional benefits. In this section, we describe optimizations for operator parallelism, data exchange, workload-aware cache management and checkpoint placement, and an operator linearization strategy to increase parallel execution.

5.1 Asynchronous Remote Jobs

To enable inter-backend parallelism and asynchronous data transfer, we introduce new operators and rewrites.

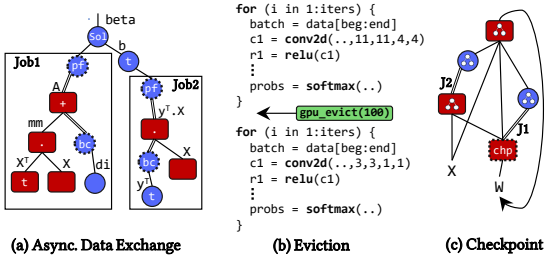


Figure 9: Compiler-assisted Reuse.

First, we add a `prefetch` operator that triggers remote (Spark/GPU) jobs and asynchronously fetches results without blocking the CPU instruction stream. The corresponding rewrite identifies operators that trigger remote jobs via `collect` or `cudaMemcpyDeviceToHost`, treats them as roots of remote operator chains, and inserts `prefetch` instructions after these roots. Additionally, this rewrite flags all other Spark actions for asynchronous execution. At runtime, the operator scheduler launches these operations asynchronously and returns `future` objects [58], enabling concurrent execution of Spark jobs, GPU kernels, and local operators. Figure 9(a) shows the DAG from Example 1 after `prefetch` placement (*pf*) for two jobs. Since immediate caching is infeasible for asynchronous execution, the main thread propagates lineage traces to prefetch threads, allowing fetched results to be cached (`PUT`) once available and reused across iterations. The `prefetch` instruction of Job2 in Figure 9(a) caches the fetched result of $\mathbf{y}^\top\mathbf{X}$. Second, we introduce an asynchronous `broadcast` operator to optimize local-to-remote data transfer, which the compiler places at the end of local operator chains (see *bc* operators in Figure 9(a)).

5.2 Workload-aware Cache Management

We introduce new cache management operators and placement rewrites to reduce caching overhead and improve the utility of reuse and memory management.

Eviction Injection: Our eviction logic, (in Section 4.3), operates incrementally, evicting one object-at-a-time to make space for new objects. This approach incurs high eviction overhead during allocation pattern shifts, common in GPU workloads. Figure 9(b), shows an ensemble learning script utilizing two models (AlexNet & VGG16) for joint prediction. In the first `for` loop, MEMPHIS efficiently recycles the less reusable pre-allocated pointers. However, the allocation pattern shifts in the second loop due to varying `conv2d` kernel sizes, leading to high allocation-deallocation overhead and memory fragmentation. To mitigate this issue, we introduce an `evict` instruction for cache cleanup. A program-level rewrite identifies these loop patterns and injects `evict` instructions with arguments of cache size to clean up. At runtime, the `evict` instruction utilizes the backend-specific eviction logic to free up space. The rewrite avoids full eviction if access patterns repeat. Other DNN frameworks require manual placement of such cleanup [18].

RDD Checkpoint Placement: Lineage-based reuse eliminates redundancy across Spark jobs, but Spark's lazy evaluation introduces another form of redundancy due to shared dataflow dependencies between jobs. Prior work [6, 14] proposes, rule-based checkpoint (`persist`) placements. Here, we introduce two workload-aware rewrites for checkpoint placement. (1) The first rewrite identifies overlapping Spark jobs within a basic block and injects a checkpoint after the last shared operator. (2) The second rewrite targets common iterative algorithms, where updated variables create increasingly large operator graphs. Figure 9(c) shows a simplified DAG of one factor matrix $\mathbf{W}$ of Poisson Non-negative Matrix Factorization [47]. Here, each node represents a Spark (red) or local (blue) operator chain. Every iteration updates $\mathbf{W}$ and triggers two jobs (J1, J2), both lazily executing all previous iterations. Our rewrite identifies the variables that are updated in each iteration and places checkpoints to reuse previous iterations' results. In this example, we cache $\mathbf{W}$ (indicated by *chp*) in each iteration. For a seamless integration, at runtime, the lineage cache eviction tracks these checkpointed RDDs and updates the cache metadata.

Delayed Caching: Caching incurs a probing cost and other backend-specific overheads, especially for long-running, complex workloads. Caching non-repeating large RDDs increases memory pressure and garbage collection overhead on both executors and drivers. Similarly, DNN training with no reuse potential in GPUs suffer from allocation overhead, memory fill-up, and fragmentation. Based on the observation that reusable operations repeat multiple times for hierarchically composed ML pipelines, we introduce *delayed caching*, that defers caching until the n-th (*delay factor*) cache hit. At runtime, the `PUT` call creates an empty lineage cache entry with status `TO-BE-CACHED` upon the first cache hit. If the operator repeats n times, we put the actual object in the cache and change the status to `CACHED`. Together with our cache eviction schemes, cache entry strategies such as delayed caching substantially reduce the overhead and cache misses.

Automatic Parameter Tuning: We introduce a program-level rewrite for tuning the *delay factor* n (no delay is $n = 1$) and the *Spark storage level* (for RDD caching) of each basic block, based on estimated reuse potential. This rewrite recursively traverses all program blocks, analyzing the execution frequency (nested loops, function calls) and the presence of loop-dependent operations (not reusable). A second pass then assigns the values for n ($n = 1$ if >80% reusable) and storage level, and stores them in the block headers. Figure 10 shows a *simplified* ML pipeline where, operations in block 1 (step-wise feature selection [2]) are loop-iteration-dependent ($\mathbf{X}_i$ varies with i) and thus, are deemed not reusable ($n = 4$). In block 3, cleaning and outlier removal methods `imputeMV` and `outlrIQR` are independent of λ and thus, reusable ($n = 1$). The training in block 4 is partially λ-dependent ($n = 2$). Similarly, we assign the storage level `MEMORY_AND_DISK` to block 2 and 3, and `MEMORY` (avoids spilling to disk) to 1 and 4, respectively.

```
# Feature selection
for (i in 1:ncol(X)) {
  Xi = cbind(X_g, X[,i])
1 ac = lm(Xi,..)
  ..check if best AIC..
  X_b = cbind(X_b, X[,i])
}                          n = 4
# Hyperparameter tuning
for (λ in lambdas)
2 model = TrainPipe(λ)     n = 1
# Clean and train
TrainPipe = function(λ) {
3 X_cl1 = imputeMV(X_b)
  X_ol2 = outlrIQR(X_cl1)  n = 1
  while(minimize)
4   ..training loop..      n = 2
}
```

Figure 10: Delay Factors.

5.3 Operator Ordering

Traditional backend-agnostic operator ordering is suboptimal for multi-backend systems. For a holistic inter-backend parallelism via asynchronous operators, we introduce a new operator ordering algorithm, MAXPARALLELIZE that aims to maximize opportunities for concurrent execution of local and

Algorithm 1 Operator linearization

Input: Operator DAG D with root R
Output: List of instructions **L**

```
1: if hasNoRemoteOps(D) then                    // All local OPs
2:     L = depthFirst(D)                        // Linearize depth-first
   // Step1: Identify OP chains & count operators
3: SRoots = collectSPRoots(D)
4: GRoots = collectGPRoots(D)
5: for each list Roots in SRoots, GRoots do
6:     for each root r in Roots do
7:         if r is Spark then
8:             nSPOps[r] = countSPOps(r)
9:         else if r is GPU then
10:            nGPOps[r] = countGPOps(r)
   // Step2: Sort and linearize remote jobs (longer jobs first)
11: Roots = SRoots + GRoots
12: Roots = sortRootsByOpCount(SRoots, nSPOps, nGPOps)
13: for each root r in Roots do
14:     if r is Spark then
15:         depthFirst(r, nSPOps[r], L)
16:     else if r is GPU then
17:         depthFirst(r, nGPOps[r], L)
   // Step3: Linearize the rest of the local OPs
18: depthFirst(R, L)                            // Place unvisited nodes of D
```

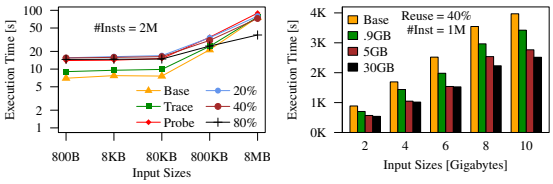
remote operator pipelines. As Algorithm 1 shows, we identify the operator chain roots (Spark action/prefetch/GPU-to-host copy) and linearize them in descending order of their lengths to maximize parallelism—longer operator chains allow for more concurrent execution, thus, increasing the degree of parallelism. The depthFirst method recursively processes sub-DAGs, with node memoization. This tight operator packing also improves memory usage by reducing the lifetime of dangling RDD references [42]. For Figure 9(a), maxParallelize linearizes Job1 followed by Job2. The `prefetch` operators trigger these jobs concurrently, where `solve` and transpose wait on the `future` objects for results.

5.4 Applicability and Future Work

We now summarize the larger scopes, limitations, and future work of MEMPHIS and its components.

Applicability: Our proposed techniques generalize across diverse data systems with varying scope. (1) MEMPHIS applies to compilation-based, multi-backend systems for unified data processing [12, 31], integrated data analysis [21], polyglot, and AutoML. (2) *Fine-grained lineage tracing* applies broadly to systems with heterogeneous compilation and execution strategies, including ML libraries and databases [54], however lineage deduplication (discussed in LIMA [71]) requires control-flow access that is limited in Python-hosted libraries. (3) *Fine-grained reuse* is most effective in compilation-based ML systems [57] and relational databases [5]. (4) *Memory management and reuse in Spark* generalize to lazily-evaluated distributed frameworks like Dask [74] and Ray [58]. (5)*Reuse and memory management in GPU* naturally extend to multi-GPU and other accelerators. (6) Our *compiler optimizations*—including prefetch, operator ordering, eviction, and checkpoint injection—also extend to compilation-based systems and standalone key-value stores [25]. Overall, our runtime and compiler techniques extend across workloads and system architectures.

Limitations and Future Work: Our lineage-based reuse framework makes several tradeoffs and opens opportunities for future work. (1) *Reuse potential* is workload-dependent, however, even without reuse, our lineage tracing improves interpretability and debuggability with negligible overhead. A future extension is enabling query processing over serialized lineage traces to analyze convergence and execution behavior. (2) Our cache is designed for process-wide sharing and requires extensions for *reusing across processes*. In addition, Python ML libraries may require instrumentation for tracing lineage [32]. (3) Despite handling non-determinism, reconstructed programs may yield slightly *different results* under varying configurations or versions. (4) Finally, a promising future work is a *cost-based, reuse-aware optimizer* that jointly optimizes query plan and operator scheduling.



(a) Reuse Overhead - Sizes (b) Cache Size Comparison

Figure 11: Reuse Overhead & Cache Size Influence.

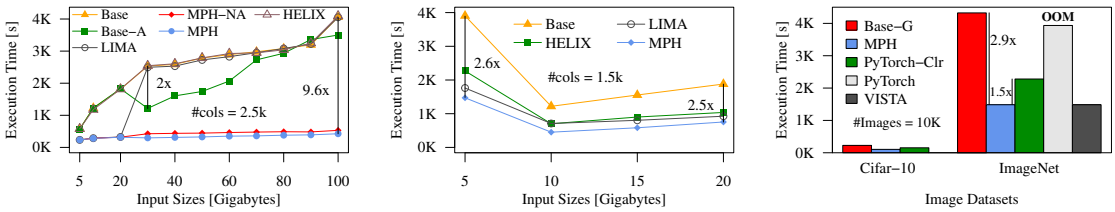
6. EXPERIMENTS

We present some highlights from our experimental study [69]. Overall, our evaluation systematically scales data size, task complexity, and instruction counts across diverse ML pipelines—including data pre-processing, hyper-parameter optimization, matrix factorization, and DNN—to show benefits of multi-backend reuse in realistic settings.

Experimental Setting: We ran all experiments on a cluster of 8+1 scale-out nodes and a scale-up node with two NVIDIA A40 GPUs with 48 GB. The software stack comprises Ubuntu 20.04.6, Hadoop 3.3, Spark 3.5, and CUDA 10.2. The Spark cluster uses 38GB driver memory and 230GB executor memory; operators exceeding 7GB are compiled to Spark. The lineage cache is configured to 5GB on the driver and 55GB per executor. We compare MEMPHIS with different SystemDS configurations, application-specific reuse frameworks such as LIMA [71] (fine-grained reuse), HELIX [87] (coarse-grained reuse), CoorDL [55] (*input data pipeline* reuse), Clipper [19] (prediction reuse), and VISTA [60] (reuse in transfer learning), and PyTorch [64] as a strong baseline for DNN workloads on GPUs.

Reuse Overhead: We analyze the overhead of lineage tracing and cache probing using a hyper-parameter tuning workload based on L2SVM, varying input sizes (800B–8 MB), and reuse rates. With a fixed 200 iterations (2M instructions), increasing input size scales compute cost per instruction while tracing and cache management overheads remain constant; reuse is simulated by randomly repeating hyper-parameters, yielding 20–80% reusable operations. Figure 11(a) compares *Base* (vanilla SystemDS) with *Trace* (tracing only) and *Probe* (reuse enabled but no hits). For small inputs, execution time is dominated by interpretation and bookkeeping overhead, with tracing and probing increasing runtime by 1.3x and 2x, and reuse providing no benefit. For larger inputs (8 MB), these overheads become negligible, and reuse yields speedups from 1.1x (20%) to 3x (80%).

Cache Size Comparison: We evaluate driver cache sizes using the same workload with larger inputs (2–10 GB), 1M instructions, and 40% reuse, comparing cache sizes of 900 MB,



(a) HCV (b) HBAND (c) TLVIS

Figure 12: Performance of End-to-end ML Pipelines.

5 GB, and 30 GB. Inputs of 8 GB and 10 GB trigger Spark operations. Figure 11(b) shows even a 900 MB cache consistently achieves 1.2x speedup. While 5 GB and 30 GB caches perform similarly for smaller inputs, the larger cache yields slightly higher speedups for large inputs (1.6x vs. 1.4x). These gains stem from reuse of local intermediates, Spark RDDs and actions (including `prefetch`), proving the robustness of our eviction policies even under repeated evictions.

Hyper-parameter Tuning of Linear Regression (HCV): We evaluate multi-backend reuse on an end-to-end hyper-parameter tuning pipeline that performs cross-validated linear regression (Example 1) on synthetic datasets with 10 regularization parameters, selecting the best model using R^2. We vary input sizes from 5-100GB and compare Base, LIMA, HELIX, and MEMPHIS (MPH), where Base-A and MPH enable asynchronous operators and Base and MPH-NA do not. As shown in Figure 12(a), MPH achieves up to 9.6x speedup over Base by reusing $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$ across folds and overlapping execution via `prefetch`. From 25 GB onward, these intermediates are placed in Spark; LIMA reuses only local intermediates up to 20 GB, while MPH reuses in all settings. HELIX performs similarly to Base due to the lack of fine-grained reuse, Base-A achieves 2x speedup from concurrency for smaller inputs, and MPH is about 20% faster than MPH-NA due to parallel operator execution.

Hyperband Model Selection (HBAND): This experiment evaluates model search using a Hyperband-like multi-armed bandit [45] with weighted ensemble learning on synthetic datasets. HBAND first applies successive halving to tune hyper-parameters of L2SVM and multi-class logistic regression (MLRG) via a grid search over regularization (`reg`) and intercept options, iterating over five brackets that halve the `reg` list (starting from 25 values) while doubling iterations (from 10). It then applies weighted ensemble learning to combine the best models, optimizing ensemble weights through a random search over 1K configurations. As shown in Figure 12(b), MEMPHIS (MPH) yields 2.6x/2.5x speedups for 5 GB/20 GB inputs over Base by reusing successive-halving iterations and **XB** computations in class probability estimation, reusing approximately 4K RDDs, 2K Spark actions, and 40K local matrices. Our compiler enabled *delayed caching* of RDDs for MPH. Overall, MPH outperforms HELIX and LIMA by about 40%, and remains 20% faster than LIMA even for local operations.

Transfer Learning for Vision Models (TLVIS): Transfer learning is a widely used alternative to training large models from scratch. We evaluate MEMPHIS for transfer learning, where practitioners test multiple pre-trained models to identify the most suitable model-layer pair for a downstream task [60, 73]. TLVIS uses AlexNet [40], VGG16 [80], and ResNet18 [34], extracting frozen feature layers ranging from Conv4–FC7 (AlexNet), Conv5–FC7 (VGG), and the last four residual blocks (ResNet), and ranks extracted features using a linear classifier proxy [63, 82]. Experiments run on CIFAR-10 and ImageNet test sets (10K images each) on GPUs. As shown in Figure 12(c), MEMPHIS achieves 2x and 3x speedups on CIFAR-10 and ImageNet by reusing intermediates across repeated feature extractions and efficient pointer recycling without degrading reuse (e.g., reusing and recycling 30K and 17.5K pointers for ImageNet). MEMPHIS compiles an `evict(100)` instruction between models (*eviction injection*) to free up the allocated pointers. VISTA performs similarly by applying CSE across pipelines. Compared to PyTorch, which uses frozen pre-trained models and GPU pre-loading, `torch.compile` suffers from out-of-memory errors and requires manual cache cleanup (`empty_cache()`) after each model (PyTorch-Clr). PyTorch—as a specialized system for DNNs—is 1.9x faster than Base-G (SystemDS-GPU) for TLVIS but 1.5× slower than MEMPHIS due to lack of reuse of repeated feature extractions.

7. CONCLUSIONS

After five years of developing this framework [68], we draw several key conclusions. As systems become increasingly heterogeneous across runtimes and backends, and ML workloads expand to multi-modal pipelines, LLMs, and AI agents, redundancy is both inevitable and progressively harder to manage. The diversity of workloads, execution models, and backends makes manual redundancy elimination infeasible and static cache management ineffective. We introduced MEMPHIS as a holistic framework for efficient, multi-backend reuse of intermediates and memory management. Our hierarchical lineage cache seamlessly allows reuse across heterogeneous backends and system architectures. We devised tailored eviction policies and memory management techniques for Spark and GPUs. Our compiler extensions improve reuse potential, reduce runtime overheads, and enable concurrent execution. Our experiments have shown robust improvements across diverse workloads. Overall, MEMPHIS eliminates redundancy in a principled and systematic manner while remaining broadly applicable to future workloads, backends, and evolving system architectures.

8. ACKNOWLEDGMENTS

We gratefully acknowledge funding from the German Federal Ministry of Research, Technology and Space (under research grant BIFOLD25B).

9. REFERENCES

[1] Martín Abadi et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, pages 265–283, 2016.

[2] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. Mind the Gap: Bridging Multi-Domain Query Workloads with EmptyHeaded. *PVLDB*, 10(12):1849–1852, 2017.

[3] Divy Agrawal et al. RHEEM: enabling cross-platform data processing - may the big data be with you! -. *PVLDB*, 11(11):1414–1427, 2018.

[4] Hani Al-Sayeh, Bunjamin Memishi, Muhammad Attahir Jibril, Marcus Paradies, and Kai-Uwe Sattler. Juggler: Autonomous cost optimization and performance prediction of big data applications. In *SIGMOD*, pages 1840–1854, 2022.

[5] Alex Hall. Caching & reuse of subresults across queries, 2024. URL: https://www.firebolt.io/blog/caching-reuse-of-subresults-across-queries.

[6] Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Felix Schüler, Lauritz Thamsen, Odej Kao, Tobias Herb, and Volker Markl. Implicit parallelism through deep language embedding. In *SIGMOD*, pages 47–61, 2015.

[7] Antreas Antoniou, Amos J. Storkey, and Harrison Edwards. Augmenting image classifiers using data augmentation generative adversarial networks. In *ICANN*, volume 11141 of *Lecture Notes in Computer Science*, pages 594–603, 2018. doi:10.1007/978-3-030-01424-7_58.

[8] Paul Barham et al. Pathways: Asynchronous distributed dataflow for ML. In Diana Marculescu, Yuejie Chi, and Carole-Jean Wu, editors, *MLSys*, 2022.

[9] Nirvik Baruah, Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia. Parallelism-optimizing data placement for faster data-parallel computations. *PVLDB*, 16(4):760–771, 2022. doi:10.14778/3574245.3574260.

[10] Sebastian Baunsgaard et al. Exdra: Exploratory data science on federated raw data. In *SIGMOD*, pages 2450–2463, 2021. doi:10.1145/3448016.3457549.

[11] Denis Baylor et al. TFX: A tensorflow-based production-scale machine learning platform. In *KDD*, pages 1387–1395, 2017.

[12] Kaustubh Beedkar, Bertty Contreras-Rojas, Haralampos Gavriilidis, Zoi Kaoudi, Volker Markl, Rodrigo Pardo-Meza, and Jorge-Arnulfo Quiané-Ruiz. Apache wayang: A unified data analytics framework. *SIGMOD Rec.*, 52(3):30–35, 2023.

[13] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Rev.*, 59(1):65–98, 2017.

[14] Matthias Boehm et al. Systemml: Declarative machine learning on spark. *PVLDB*, 9(13):1425–1436, 2016. doi:10.14778/3007263.3007279.

[15] Matthias Boehm et al. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *CIDR*, 2020. URL: http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf.

[16] Matthias Boehm, Arun Kumar, and Jun Yang. *Data Management in Machine Learning Systems*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.

[17] Dami Choi and George Dahl. Speeding up neural network training with data echoing, 2020. URL: https://ai.googleblog.com/2020/05/speeding-up-neural-network-training.html.

[18] PyTorch Contributors. Memory management of pytorch, 2023. URL: https://pytorch.org/docs/stable/notes/cuda.html#memory-management.

[19] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627, 2017.

[20] Ekin D. Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation strategies from data. In *CVPR*, pages 113–123, 2019. doi:10.1109/CVPR.2019.00020.

[21] Patrick Damme et al. DAPHNE: an open and extensible system infrastructure for integrated data analysis pipelines. In *CIDR*, 2022.

[22] Piali Das et al. Amazon sagemaker autopilot: a white box automl solution at scale. In Sebastian Schelter, Steven Whang, and Julia Stoyanovich, editors, *DEEM@SIGMOD*, pages 2:1–2:7, 2020.

[23] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. Optimizing Machine Learning Workloads in Collaborative Environments. In *SIGMOD*, pages 1701–1716, 2020. doi:10.1145/3318464.3389715.

[24] Apache Beam Developers. The Unified Apache Beam Model, 2024. URL: https://beam.apache.org/.

[25] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4):26:1–26:32, 2021. doi:10.1145/3483840.

[26] Shiqing Fan et al. DAPPLE: a pipelined data parallel approach for training large models. In *PPoPP*, pages 431–445, 2021. doi:10.1145/3437801.3441593.

[27] Haoyang Fang et al. Mlzero: A multi-agent system for end-to-end machine learning automation. In *Advances in Neural Information Processing Systems*, 2025. URL: https://neurips.cc/virtual/2025/poster/119504.

[28] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. Auto-sklearn: Efficient and robust automated machine learning. In *Automated Machine Learning - Methods, Systems, Challenges*, pages 113–134. 2019.

[29] Haralampos Gavriilidis, Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. In-situ cross-database query processing. In *ICDE*, pages 2794–2807, 2023.

[30] Victor Giannakouris, Nikolaos Papailiou, Dimitrios Tsoumakos, and Nectarios Koziris. Musqle: Distributed SQL query execution over multiple engine environments. In *IEEE BigData*, pages 452–461, 2016.

[31] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. Musketeer: all for one, one for all in data processing systems. In *EuroSys*, pages 2:1–2:16, 2015.

[32] Stefan Grafberger, Paul Groth, and Sebastian Schelter. Automating and optimizing data-centric

what-if analyses on native machine learning pipelines. *Proc. ACM Manag. Data*, 1(2):128:1–128:26, 2023. doi:10.1145/3589273.

[33] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *ATC*, pages 689–706, 2022.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[35] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *ASPLOS*, pages 875–890, 2020.

[36] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. An architecture for recycling intermediates in a column-store. In *SIGMOD*, pages 309–320, 2009. URL: https://doi.org/10.1145/1559845.1559879.

[37] Paras Jain et al. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *MLSys*, 2020.

[38] Abhinav Jangda et al. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *ASPLOS*, pages 402–416, 2022. doi:10.1145/3503222.3507778.

[39] Antonios Kontaxakis, Dimitris Sacharidis, Alkis Simitsis, Alberto Abelló, and Sergi Nadal. HYPPO: Using Equivalences to Optimize Pipelines in Exploratory Machine Learning. In *ICDE*, pages 221–234, 2024.

[40] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, pages 1106–1114, 2012.

[41] Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *OSDI*, pages 19–35, 2021.

[42] Lukas Landgraf, Wolfgang Lehner, Florian Wolf, and Alexander Boehm. Memory efficient scheduling of query pipeline execution. In *CIDR*, 2022.

[43] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: opening the black box of machine learning prediction serving systems. In *OSDI*, pages 611–626, 2018.

[44] Liam Li, Kevin G. Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. In *MLSys*, 2020.

[45] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18:185:1–185:52, 2017. URL: http://jmlr.org/papers/v18/16-558.html.

[46] Edo Liberty et al. Elastic machine learning algorithms in amazon sagemaker. In *SIGMOD*, pages 731–737, 2020. doi:10.1145/3318464.3386126.

[47] Chao Liu, Hung-chih Yang, Jinliang Fan, Li-wei He, and Yi-Min Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *WWW*, pages 681–690, 2010.

[48] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *NIPS*, pages 4765–4774, 2017.

[49] Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Towards automatically-tuned neural networks. In *(ICML*, volume 64, pages 58–65, 2016.

[50] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by gpu memory optimization on tensorflow. In *ML System@NeurIPS Workshop*, 2017.

[51] Xiangrui Meng et al. MLlib: Machine Learning in Apache Spark. *JMLR*, 17:34:1–34:7, 2016.

[52] Fraser Mince, Dzung Dinh, Jonas Kgomo, Neil Thompson, and Sara Hooker. The grand illusion: The myth of software portability and implications for ML progress. In *NeurIPS 2023*, 2023.

[53] Azalia Mirhoseini et al. Device placement optimization with reinforcement learning. In Doina Precup and Yee Whye Teh, editors, *ICML*, pages 2430–2439, 2017.

[54] Haneen Mohammed and Eugene Wu. Lineage capture trade-offs: A case study in duckdb. In *ProvenanceWeek@SIGMOD*, pages 32–36, 2025.

[55] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *PVLDB*, 14(5):771–784, 2021. doi:10.14778/3446095.3446100.

[56] Hubert Mohr-Daurat, Xuan Sun, and Holger Pirk. BOSS - an architecture for database kernel composition. *PVLDB*, 17(4):877–890, 2023.

[57] Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. Autograph: Imperative-style coding with graph-based performance. In *MLSys*, 2019. URL: https://proceedings.mlsys.org/book/272.pdf.

[58] Philipp Moritz et al. Ray: A distributed framework for emerging AI applications. In *OSDI*, pages 561–577, 2018.

[59] Derek Gordon Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. *PVLDB*, 14(12):2945–2958, 2021. doi:10.14778/3476311.3476374.

[60] Supun Nakandala and Arun Kumar. Vista: Optimized system for declarative feature transfer from deep cnns at scale. In *SIGMOD*, pages 1685–1700, 2020.

[61] Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Jinwoo Shin, Sercan Ö. Arik, and Tomas Pfister. MLE-STAR: machine learning engineering agent via search and targeted refinement. In *NeurIPS*, 2025.

[62] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for DNN training. In *SOSP*, pages 1–15, 2019.

[63] Cuong V. Nguyen, Tal Hassner, Matthias W. Seeger, and Cédric Archambeau. LEEP: A new measure to evaluate transferability of learned representations. In *ICML*, pages 7294–7305, 2020.

[64] Adam Paszke et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035, 2019.

[65] Fabian Pedregosa et al. Scikit-learn: Machine learning in python. *JMLR*, 12:2825–2830, 2011.

[66] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based GPU memory management for deep learning. In *ASPLOS*, pages 891–905, 2020. doi:10.1145/3373376.3378505.

[67] Tiago B. G. Perez, Xiaobo Zhou, and Dazhao Cheng. Reference-distance eviction and prefetching for cache management in spark. In *ICPP*, pages 88:1–88:10, 2018.

[68] Arnab Phani. *Fine-grained reuse and feature transformations in machine learning systems*. PhD thesis, Technical University of Berlin, Germany, 2025. URL: https://nbn-resolving.org/urn:nbn:de:101:1-2504300159545.425811098571.

[69] Arnab Phani and Matthias Boehm. MEMPHIS: holistic lineage-based reuse and memory management for multi-backend ML systems. In *EDBT*, pages 255–269. OpenProceedings.org, 2025. doi:10.48786/EDBT.2025.21.

[70] Arnab Phani, Lukas Erlbacher, and Matthias Boehm. UPLIFT: parallelization strategies for feature transformations in machine learning workloads. *PVLDB*, 15(11):2929–2938, 2022.

[71] Arnab Phani, Benjamin Rath, and Matthias Boehm. LIMA: fine-grained lineage tracing and reuse in machine learning systems. In *SIGMOD*, pages 1426–1439, 2021. doi:10.1145/3448016.3452788.

[72] Sergey Redyuk, Zoi Kaoudi, Volker Markl, and Sebastian Schelter. Automating data quality validation for dynamic data ingestion. In *EDBT*, pages 61–72, 2021. doi:10.5441/002/EDBT.2021.07.

[73] Cédric Renggli, Xiaozhe Yao, Luka Kolar, Luka Rimanic, Ana Klimovic, and Ce Zhang. Shift: An efficient, flexible search engine for transfer learning. *PVLDB*, 16(2):304–316, 2022.

[74] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *SciPy*, pages 130 – 136, 2015.

[75] Svetlana Sagadeeva and Matthias Boehm. Sliceline: Fast, linear-algebra-based slice finding for ML model debugging. In *SIGMOD*, pages 2290–2299, 2021.

[76] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Bießmann, and Andreas Grafberger. Automating Large-Scale Data Quality Verification. *PVLDB*, 11(12):1781–1794, 2018.

[77] Noam Shazeer et al. Mesh-tensorflow: Deep learning for supercomputers. In *NeurIPS*, pages 10435–10444, 2018.

[78] Shafaq Siddiqi, Roman Kern, and Matthias Boehm. Saga: A scalable framework for optimizing data cleaning pipelines for machine learning applications. *Proc. ACM Manag. Data*, 2024.

[79] Shafaq Siddiqi, Arnab Phani, Roman Kern, and Matthias Boehm. Saga++: A scalable framework for optimizing data cleaning pipelines for machine learning applications. *ACM TODS*, 2025.

[80] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[81] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *ICDE*, pages 535–546, 2017.

[82] Anh Tuan Tran, Cuong V. Nguyen, and Tal Hassner. Transferability and hardness of supervised classification tasks. In *ICCV*, pages 1395–1405, 2019.

[83] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. Fastflow: Accelerating deep learning model training with smart offloading of input data pipeline. *PVLDB*, 16(5):1086–1099, 2023.

[84] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Comput. Sci. Eng.*, 13(2):22–30, 2011.

[85] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. MISTIQUE: A system to store and query model intermediates for model diagnosis. In *SIGMOD*, pages 1285–1300, 2018.

[86] Yongji Wu, Matthew Lentz, Danyang Zhuo, and Yao Lu. Serving and optimizing machine learning workflows on heterogeneous infrastructures. *PVLDB*, 16(3):406–419, 2022. doi:10.14778/3570690.3570692.

[87] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya G. Parameswaran. Helix: Holistic optimization for accelerating iterative machine learning. *PVLDB*, 12(4):446–460, 2018.

[88] Luna Xu, Min Li, Li Zhang, Ali Raza Butt, Yandong Wang, and Zane Zhenhua Hu. MEMTUNE: dynamic memory management for in-memory data analytic platforms. In *IPDPS*, pages 383–392, 2016.

[89] Matei Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, pages 15–28, 2012.

[90] Ce Zhang, Arun Kumar, and Christopher Ré. Materialization optimizations for feature selection workloads. In *SIGMOD*, pages 265–276, 2014.

[91] Yanli Zhao et al. Pytorch FSDP: experiences on scaling fully sharded data parallel. *PVLDB*, 16(12):3848–3860, 2023.

[92] Lianmin Zheng et al. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *OSDI*, pages 559–578, 2022.

[93] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, pages 533–544, 2007. URL: https://doi.org/10.1145/1247480.1247540.