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

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

Matthias Boehm
TU Berlin, 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 ML 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 for asynchronous exchange, workload-aware speculative cache management, and related operator ordering for concurrent execution. Our experiments across diverse ML tasks and pipelines show improvements 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 input data like text, images, and tabular data [100]. Data scientists hierarchically compose complex ML pipelines from black-box primitives [25]. The exploratory nature of these pipelines causes high computational redundancy [39, 66, 101, 125].

Sources of Redundancy: The high computational redundancy stems from various sources including incremental modifications of ML pipelines in AutoML and hyper-parameter tuning, as well as fine-grained redundancy in training, inference, and transfer learning [101]. AutoML and similar tools [37, 45, 78, 83, 114] combine tasks like data cleaning, feature engineering, hyper-parameter tuning, and model training [74, 75], and then explore various combinations with slight changes but, for instance, shared pre-processing. Deep neural network (DNN) workloads also execute data pipelines [53, 88, 92] and forward paths [93, 104] repeatably at a batch granularity. Similarly, inference frameworks for object detection and machine translation [30] encounter duplicate inputs [33, 73, 124].

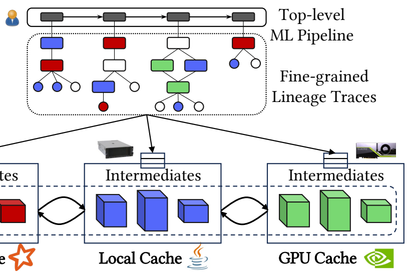


Figure 1: Multi-backend Reuse Cache for Intermediates.

Multi-backend ML Systems: ML pipelines with diverse data and workload characteristics necessitate multiple, specialized backends for achieving both efficiency and scalability, while utilizing available hardware resources and compute clusters. Example pipelines include (1) hybrid local/distributed runtime plans for large-scale ML [24, 85, 106, 129], (2) large-scale data validation [103, 109] and cleaning [114], (3) sampling, feature selection [25], and data augmentation [12, 34], which iteratively change data sizes by orders of magnitude, (4) exploratory ML algorithm research [86] (e.g., hybrid batch-minibatch training with large batch phases [15, 35]), (5) model training on datasets combining structured and unstructured features [93], (6) AutoML systems [24, 25, 78] supporting diverse ML algorithms, and (7) model debugging [80, 108] with configuration-dependent sizes of intermediates. These tasks are often combined into complex pipelines, increasingly fostering the development of ML systems with optimizing compilers, specialized operator placements [21, 25, 78], and multiple backends including local CPU/GPU/FPGAs, distributed MapReduce/Spark/Ray and federated backends [18, 69]. Example systems include PyTorch [96] and TensorFlow [2] (leverage CPU, GPU), MLlib [129] and Dask-ML [106] (utilize local and distributed), and SageMaker [78] (utilizes CPU, GPU, and Spark). Additionally, unified data analytics frameworks [21, 40, 49], polyglot [5, 47, 48], federated [18, 69], and composable [89] 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. To address this diversity, modern ML systems employ various techniques for memory management [8, 61, 99, 126], operator placement [16, 17, 87], data exchange [53, 62, 92, 120, 132], and parallelization [44, 94, 113, 134] tailored to specific workloads and backends. This heterogeneity necessitates a principled approach for efficient reuse and robust cache integration across diverse compilation, memory management, and operator scheduling techniques, which is currently lacking.

Table 1: Prior Work on Reuse & Memory Management.

System	Reuse	Multi-backend	Mem.Mgmt.	Workload
HELIX [125]	Coarse	No	No	ML Pipelines
Clippr [33]	Coarse	No	No	Inference
LIMA [101]	Fine	No	No	All
MEMTUNE [126]	Fine	RDD	Yes	Spark Jobs
PyTorch [96]	No	Recycle GPU Ptr.	Yes	DNN
Capuchin [98]	No	Activations (GPU)	No	DNN
Cachew [53]	Coarse	Distributed	No	Preprocessing
VISTA [93]	Coarse	DNN Layers	No	Feature Extract
MEMPHIS	Fine	RDD, GPU Ptr.	Yes	All

Existing Work on Reuse: Prior work on reuse in ML pipelines rely on coarse-grained lineage tracing at the pipeline level to reuse coarse-grained results of the top-level primitives (see Figure 1) through compile-time materialization [39, 66, 116, 123, 125, 130]. However, this black-box view of individual preprocessing steps, feature engineering, and ML algorithms fails to handle the ubiquitous fine-grained redundancy (e.g., repeated matrix multiplications inside/across primitives) and non-determinism (randomized primitives). The LIMA framework [101] introduced fine-grained reuse, leveraging lineage traces on individual operations and functions to uniquely identify reusable intermediates, but was limited to local CPU operations. Table 1 summarizes the previous work, highlighting their reuse type, multi-backend reuse support, memory management capabilities (e.g. dynamic cache size), and target workloads. Prior work on application-specific, multi-backend reuse includes heuristics-based Spark RDD caching [10, 24, 53], input data pipeline reuse [53, 88, 92], prediction caching [33, 73], and GPU-CPU activation offloading for DNNs [58, 84, 98]. These approaches are tailored to specific workload-backend combinations, and fail to eliminate redundancy in modern data-centric ML pipelines with diverse tasks.

MEMPHIS Overview and Contributions: We introduce MEMPHIS, a holistic framework for efficient, multi-backend reuse of intermediates and memory management *inside* ML systems. Key principles are (1) a unified cache abstraction with system-internal API and multi-backend data objects, (2) backend-specific cache management, and (3) a robust integration with ML system compiler, runtime, and memory management, supporting diverse workloads. MEMPHIS is fully integrated into Apache SystemDS[1] [25], and utilizes *three* representative backends: SystemDS for in-memory operations, Spark for distributed operations, and GPUs for hardware acceleration. MEMPHIS extends LIMA's [101] lineage-based reuse framework—which eagerly caches all in-memory intermediates—with novel compiler and runtime techniques for reusing Spark and GPU intermediates, handling Spark's lazy evaluation and small GPU memory. Our detailed technical contributions are:

- *Background:* We discuss some background of ML-system internals, Spark, GPUs and their challenges in Section 2.
- *Hierarchical Lineage Cache:* Then in Section 3, we introduce our multi-backend lineage tracing framework and its unified tracing and reuse API for easier system integration.
- *Multi-backend Reuse:* We describe runtime cache management for Spark and GPU backends in Section 4, where we reuse Spark actions, RDDs, GPU pointers; their cache evictions; and combined reuse and recycling for GPUs.
- *Compiler Integration:* For holistic integration, we introduce novel compiler optimizations for workload- and reuse-aware speculative cache management (e.g. delayed caching, eviction injection), new asynchronous operators that enable overlapping computation with data transfer, and an operator ordering algorithm to maximize concurrent pipeline execution in Section 5.
- *Experiments:* Finally, we share results of several micro-benchmarks and diverse end-to-end workloads including data cleaning, model search, matrix factorization, inference, hyperparameter tuning, and transfer learning in Section 6. Compared to PyTorch and existing reuse frameworks, MEMPHIS yields substantial improvements.

2 BACKGROUND

This section describes the necessary background of ML system internals, Spark and GPU backends, their execution models, memory management, caching primitives, and challenges.

2.1 ML Systems Background

ML systems employ a range of compilation and execution strategies. Here, we focus primarily on SystemDS's program compilation, and multi-backend operator scheduling.

Program/DAG Compilation: We categorize the optimization scope of ML systems into three main types [26].

- *Eager Execution:* Libraries like NumPy [122], PyTorch [96], and Scikit-learn [97] execute the operations directly and rely on Python to handle control flow and variable scoping.
- *DAG Compilation:* Systems like TensorFlow [2, 20] perform lazy evaluation of a DAG of operations (i.e., larger scope of optimization) but rely on the host language (Python) for control flow interpretation. However, recent work like TF AutoGraph [90] and TorchDynamo [11] also extract and integrate control flow into the computation graph.
- *Program Compilation:* Julia [22] and SystemDS [25] (previously SystemML [24]) compile a script to a hierarchy of program blocks, with every last-level block is compiled into a DAG of operations. Similarly, Wayang [21] and Musketeer [49] compile scripts into cross-platform plans.

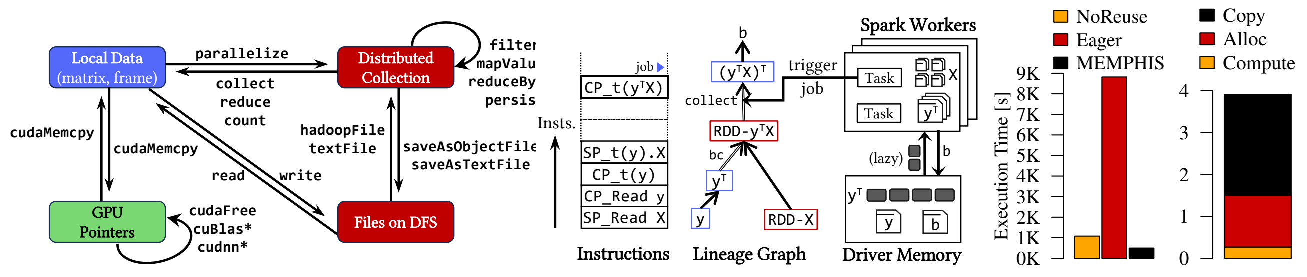
Compilation techniques like common subexpression elimination (CSE) and code motion fail to eliminate all redundancy, as the conditional control flow is often unknown during compilation.

Operator Scheduling: An operator scheduler converts operator DAGs into backend-specific instruction (kernel) streams and has two primary responsibilities: (1) *Operator placement* aims to minimize execution time of a multi-backend runtime and is done via global configurations, heuristics, or even reinforcement learning [87]. SystemDS places operations with higher memory estimates (than driver's memory) to Spark and compute-intensive, dense operations to GPU, both in a data locality-aware manner. Figure 2(a) shows the lifecycle of a data object in SystemDS across the host, Spark, and GPU, including backend-specific operations. (2) *Operator linearization* orders operator DAGs into instruction streams for sequential or parallel execution. The linearization

Table 2: Properties of Spark, GPU, and CPU.

	Exec.	Memory	Bandwd.	Cache-API	Workload
Spark	Lazy	Distrib.	15 GB/s	Yes	Large data
GPU	Async.	Small	6.1 GB/s	No	Mini-batch, DNN
CPU	Eager	Varying	—	No	ALL

[1]SystemDS: https://github.com/apache/systemds, and reproducibility package: https://github.com/damslab/reproducibility/tree/master/edbt2025-MEMPHIS



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

Figure 2: Spark and GPU Backend Details.

strategies affect parallelism and memory requirements. SystemDS linearizes DAGs to instruction streams in a depth-first manner and executes those on the respective backends.

Backends: The CPU, GPU, and Spark backends differ significantly in their execution model, memory management, and target workloads. Table 2 summarizes the key properties of Spark, GPUs, and CPUs, where the bandwidths are measured from the host (pageable host-to-device). Additional backends such as SystemDS federated [18, 19] or remote parfor [23, 24] create deeper hierarchies, where local lineage-based reuse directly apply.

2.2 Spark Backend and Challenges

Execution Model: Spark's execution model consists of a driver program for local operations and job scheduling, and multiple executor processes that run on worker nodes. RDDs (Resilient Distributed Datasets) [129] serve as a high-level programming abstraction, representing partitioned distributed collections of keyed matrix/frame tiles. Spark differentiates lazily evaluated *transformations* (distributed operations, which produce RDDs, see Figure 2(a)) and *actions*. An action (e.g. `count`, `collect`) triggers the `DAGScheduler` to construct and launch a Spark job. Each job is a DAG of stages, which comprise pipelined transformations separated by shuffle boundaries. The scheduler launches a task per partition, preferably in a data- or rack-local manner.

Memory Management: Spark's memory is divided into two regions: *execution* and *storage*. All Spark operations use the execution memory for computation and temporary partitions, while the storage is for caching and broadcasting. They share a unified memory (default 60% of heap) allowing execution to utilize unused storage memory and vice versa. There are two types of RDD caching. First, Spark implicitly caches shuffle files and broadcast data until destroyed. Caching shuffle files allows for avoiding recomputing the map side of a shuffle dependency [42]. Second, Spark provides the `persist()` API for explicitly caching RDDs (in different storage levels: deserialized/serialized and memory/disk) to avoid redundant lazy evaluation. The cached RDDs are lazily materialized in the executors' BlockManagers. Any subsequent job on a cached RDD then reads the partitions from memory or disk, thereby skipping prior operations. Spark evicts cached partitions if cached RDDs are too large or execution requires more memory. Evicted memory-only partitions or lost partitions on failures are recomputed based on Spark's lineage.

Broadcast: Broadcast-based operators like broadcast join are far less expensive than shuffle-based repartition joins. When a broadcast variable is created via `broadcast`, Spark creates a `TorrentBroadcast` that serializes and partitions the broadcast data into 4 MB chunks, and keeps them in the driver's `BlockManager` [43]. Individual chunks are then lazily transferred to selected executors, which parallelize the transfer to all executors on demand. Due to Spark's lazy evaluation, the broadcast data remains in the driver until the job completes.

Lazy Evaluation Challenges: Lazy evaluation presents several challenges for efficient reuse. First, traditional eager caching (e.g., LIMA, tf.Data [92], Cachew) eagerly materialize cached RDDs after each instruction, leading to performance drops due to repeated job executions. Figure 2(c) shows, eager materialization of 12K RDDs (4K reusable) is 10x slower than no caching at all. Second, runtime caching of all RDDs severely increases cluster memory usage (20x for experiments in Figure 2(c)), requiring speculative caching. Third, lazy evaluation delays the transfer of broadcast data to the cluster until the job execution triggers, requiring the retention of broadcast data and RDDs until the job completes. This creates *dangling references* that consume driver memory. For instance, 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. MEMPHIS employs workload-aware caching, lazy garbage collection and asynchronous job triggering, achieving a 2x speedup by reusing RDDs in Figure 2(c).

2.3 GPU Backend and Challenges

Execution Model: GPUs offer high peak performance and memory bandwidth, suitable for accelerating workloads with regular data access (e.g., DNN). SystemDS's GPU backend utilizes CUDA. Unlike Spark's lazy evaluation, CUDA kernel execution is eager and sequential within a single stream, but asynchronous for the calling host thread. Thus, the host thread continues executing other tasks while the kernels run concurrently. However, certain operations—such as device-to-host data transfer and memory deallocation—introduce synchronization barriers, which forces the CPU thread to wait until all pending GPU kernels have completed their execution. Traditional eager caching also forces synchronization barriers and slows down GPU execution.

Small Memory & Data Copy Challenges: To analyze GPU execution overhead, we ran an experiment with a single affine layer with ReLU activation for 10 epochs of 1K mini-batches of 128 rows. We force each kernel to allocate output memory, transfer the result to the host, and deallocate. Figure 2(d) shows that memory allocation/free and copy take 4.6x and 9x longer than the actual computation. This result highlights that static cache memory and traditional eviction policies, as seen in LIMA and Nectar [55], increase memory pressure and data copy overhead—making them unsuitable for GPU pointer caching. To mitigate these overheads, MEMPHIS employs dynamic cache sizes, memory recycling [132], and eviction injection optimizations.

3 HIERARCHICAL LINEAGE CACHE

The basic architecture of MEMPHIS with lineage tracing and a hierarchical cache—for seamlessly accommodating heterogeneous backends—is shown in Figure 3. The driver/host process handles the lineage tracing, compiler optimizations, and eviction planning, whereas the actual cached objects reside in the respective backends. 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. This API simplifies the integration of MEMPHIS into any ML systems, irrespective of their backends and underlying compilation and operator placement strategies.

- `TRACE(inst)`: Trace lineage for an operator (Section 3.2).
- `SERIALIZE(trace)` / `DESERIALIZE(log)`: Serialize an in-memory lineage trace to a lineage log and vice versa.
- `RECOMPUTE(log)`: Recompute the exact same results from the provided lineage log. The execution environment for `RECOMPUTE` may differ from the original environment.
- `REUSE(trace)`: Reuse the instruction output if available in the cache and skip the instruction execution.
- `PUT(trace,object)`: Put the instruction result in the cache with backend-specific pointers.
- `MAKE_SPACE(object)`: Iteratively evict cached objects to make space for a new object in the corresponding backend.

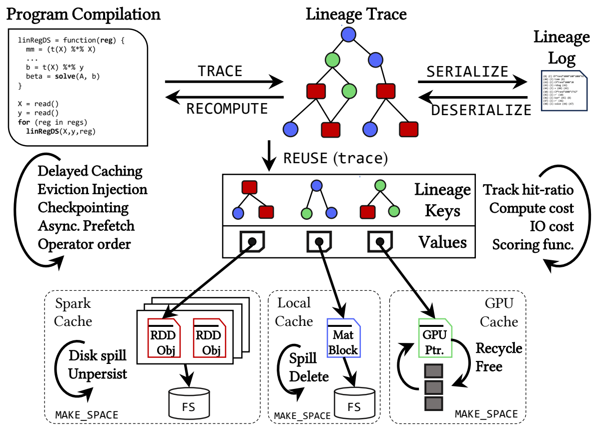
Improvements over LIMA: MEMPHIS builds upon LIMA's [101] basic lineage tracing (eager caching), and extends it with a hierarchical lineage cache and unified API for reuse across local, Spark, and GPU, robustness optimizations, and a holistic integration with ML systems. LIMA benefits moderate workloads, while MEMPHIS tackles complex multi-backend ML pipelines.

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.

Fine-grained Lineage Tracing: We call `TRACE` for each linear algebra instruction before its execution (see pseudo-code in Figure 4). MEMPHIS internally maintains a hash map (`LineageMap`) to map the live variable names to the respective lineage DAGs. A lineage item contains the opcode, data items, and pointers to the input lineage items. During tracing, each output generates a new lineage item from input items, which is then added to the `LineageMap`. For efficient lineage DAG comparisons (probing), which is a core operation for reuse, lineage items implement `hashCode` and `equals`. We calculate the hash by hashing the hashes of the input items, the opcode, and the data items. For equality check, we rely on a non-recursive, queue-based approach with sub-DAG memoization and early abort conditions based on hash mismatches, height differences, and shared sub-DAGs (object identity). This simple yet scalable backend-agnostic lineage tracing is a solid foundation for additional backends.

Recomputation for Debugging: To tackle interpretability and debuggability of complex ML tasks involving heterogeneous backends, we enable easy sharing of serialized lineage traces and exact recomputation of intermediates. The `RECOMPUTE` API first deserializes a lineage trace into an in-memory lineage DAG, followed by applying the full compilation chain to generate the instructions. Full re-compilation ensures the same results and flexibility regarding configurations and hardware environments.


```
linRegDS = function(reg) {
  mm = (t(X) %*% X)
  ...
  b = t(X) %*% y
  beta = solve(A, b)
}

X = read()
y = read()
for (reg in regs)
  linRegDS(X,y,reg)
```

Figure 3: Hierarchical Lineage Cache & Reuse Overview.

Future work includes query processing on lineage traces for model management [117].

3.3 Multi-backend Lineage Cache

We leverage the property that the lineage uniquely identifies an intermediate for reusing previously computed intermediates. Our 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 3 middle). Figure 4 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 4: Reuse API.

Redundancy in Lineage Items: Generating a new lineage item object for each instruction *before* reuse creates redundant lineage DAGs across `LineageMap` and lineage cache entries. To address this, upon successful probes, we replace the respective `LineageMap` entries with the lineage keys of cached objects. As Figure 5 shows, this compaction increases shared sub-DAGs (objects with identical references), which in turn improves probing efficiency and reduces the memory footprint.

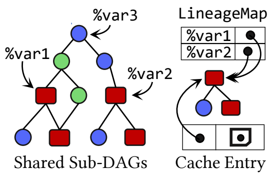


Figure 5: Compaction.

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 the driver, and GPU-to-host copies. Moreover, the wrappers enable caching the same object in multiple backends, which is beneficial for data-local scheduling. The centralized lineage tracing and reuse facilitates a reuse-aware compilation to increase reuse potentials and reduce caching overheads. Figure 3 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 [101]. 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 4) 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

The backend-local caches require tailor-made cache management strategies due to their diversity. In this section, we describe the reuse and memory management for Spark and GPU.

4.1 Reuse and Memory Management in Spark

We utilize Spark's caching API to cache RDDs in Spark cluster memory. Furthermore, we introduce memory management and cache eviction techniques 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 **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.

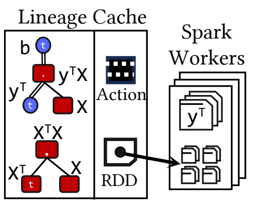


Figure 6: Spark Reuse.

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.

Lazy Garbage Collection: To tackle memory overhead caused by dangling RDDs and broadcast references (see Figure 2(b)), we track internal metadata for each RDD, including #child RDDs, #pending consumers, materialization status, and its memory overhead. During reuse, the `MAKE_SPACE` function recursively traverses child RDDs and broadcasts, updating the metadata and cleaning up stale RDDs (not in use and already materialized). For example, in Figure 6 (bottom), we clean up $\mathbf{X}^\top$ and $\mathbf{X}$ once $\mathbf{X}^\top\mathbf{X}$ is materialized. We use Spark's `destroy` and `getRDDStorageInfo` methods 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 **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.

Cache Eviction: We employ a cost-based eviction policy (orthogonal to Spark's partition-level eviction) to remove RDDs (via `unpersist`) with low reuse potential, preventing Out-of-Memory (OOM) errors. We heuristically utilize 80% (configurable) of Spark's storage memory for reuse, and the rest for broadcast variables and compiler-placed checkpoints (discussed in Section 5.2). We extend the prior Cost&Size policy [39, 101], which aims to preserve objects with high compute-cost-to-memory ratios, for Spark's lazy evaluation. In detail, we rank the operators based on analytical compute cost $c(o)$, and we collect statistics for each cached RDD o including the estimated worst-case size $s(o)$ and the number of references (#hits: r_h, #misses: r_m, #jobs: r_j), which are updated during every reuse and account for global reuse potential. The eviction scoring function for o is

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

where Q is the priority queue of RDDs. The `MAKE_SPACE` method marks RDDs for eviction via `unpersist` and refreshes cache metadata (e.g., available memory) with actual values (using `getRDDStorageInfo`). `unpersist` is an asynchronous operation, causing temporary overflow of the storage region, which is handled by Spark's partition spilling with minimum overhead. This eviction policy, combined with Spark's partition spilling, performs well in a wide variety of use cases.

Example 4.1 (Grid Search Linear Regression). In this example, we apply grid search hyper-parameter tuning on a direct-solve linear regression (`linRegDS`) for feature matrix **X** (distributed) and responses **y** (local). Figure 7 (comprising Figures 2(b) and 6) shows the `linRegDS` function, which is called by grid search 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 `linRegDS` calls. SystemDS compiles a shuffle-based matrix multiplication for $\mathbf{X}^\top\mathbf{X}$ and a broadcast-based multiplication for $\mathbf{X}^\top\mathbf{y}$ after rewriting it to $(\mathbf{y}^\top\mathbf{X})^\top$ (broadcasting $\mathbf{y}^\top$). The second transpose of $(\mathbf{y}^\top\mathbf{X})^\top$ and the `solve` trigger Spark jobs. Figure 7 shows

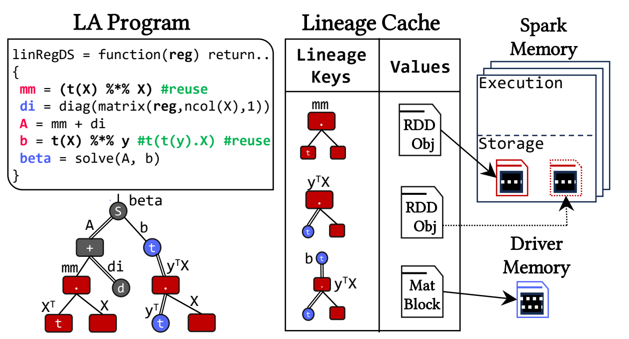

```
linRegDS = function(reg) return..
{
 mm = (t(X) %*% X) #reuse
 di = diag(matrix(reg,ncol(X),1))
 A = mm + di
 b = t(X) %*% y #t(t(y).X) #reuse
 beta = solve(A, b)
}
```

Figure 7: Spark Action and RDD Reuse Example.

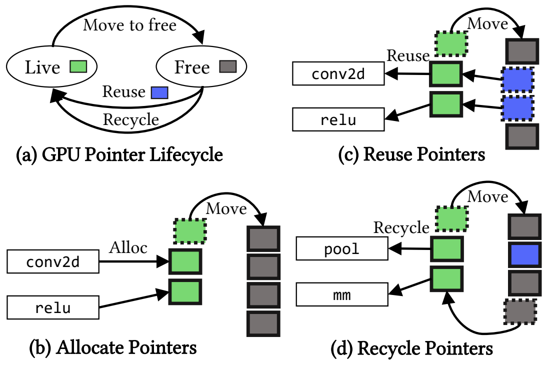


Figure 8: Reusing and Recycling of GPU Pointers.

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 **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 (**mm**, $\mathbf{y}^\top\mathbf{X}$) for distributed caching, reuses the local operations and the collected **b**. Reusing the Spark action result **b** entirely eliminates the need to trigger the Spark job. However, not triggering the job keeps $\mathbf{y}^\top\mathbf{X}$ 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 and Memory Management in GPU

To manage GPU's small memory and allocation overhead, we combine reuse and recycling in a unified memory manager with moving boundaries, where we reuse `pointers` to previous results. 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 related metadata like data characteristics. Before executing an instruction (composed of one or more GPU kernels), the `REUSE` call reuses the output pointer if available in the cache and skips launching the kernels. As Figure 8(c) shows, `REUSE` moves the reused pointer from `Free` to `Live` list. Reusing a pointer multiple times leads to multiple variables referencing a single pointer. We track the *reference count* for each pointer indicating the number of live variables referencing it, and only when the reference count becomes zero, the pointer is returned to the `Free` list.

Memory Recycling: All pointers in the `Free` queues are subject to eviction (preventing OOM). Pointers in each queue are ordered according to a scoring function. Algorithm 1 shows the

Algorithm 1 Allocate memory of size s

```
Input: Size $s$, Size-specific free pointer lists $FL$
Output: Allocated memory pointer A
 1: if CANALLOCATE(s) then                                   // GPU is not full
 2:     A = CUDAMALLOC($s$)                                  // Allocate new memory
 3: else if $FL$.CONTAINS($s$) then                          // Find free pointer of size s
 4:     A = $FL$.GETFREEPOINTEREXACT($s$)                    // Recycle
 5: else if $s \le FL$.LARGEST() then                       // Find pointer of size > s
 6:     $tmp = FL$.GETFREEPOINTERLARGERTHAN($s$)
 7:     CUDAFREE($tmp$)                                      // Deallocate
 8:     A = CUDAMALLOC($s$)                                  // Allocate new memory
 9: else if $s > FL$.LARGEST() then                         // s > all free pointers
10:     $freedSize = 0$                                      // Repeatedly deallocate
11:     while $freedSize < s$ | A is NULL do
12:         $tmp = FL$.GETFREEPOINTERNOTEXACT($s - freedSize$)
13:         $freedSize = freedSize + tmp$.SIZE()
14:         CUDAFREE($tmp$)
15:         if $freedSize \ge s$ then
16:             A = CUDAMALLOC($s$)                          // Allocate new memory
17: if A is NULL then
18:     $FL$.CLEARALL()                                      // Clear all free pointers
19:     A = CUDAMALLOC($s$)                                  // Allocate new memory
```

details for serving an allocation request. Once the GPU memory is full, as shown in Figure 8(d), we start recycling the free pointers as a form of eviction. We first look for a free pointer with the exact size to recycle. If not available, we free a pointer just larger than the required size using `cudaFree` (which may cause fragmentation). If all free pointers are smaller than the required size, we repeatedly free a pointer until `cudaMalloc` is successful. If these steps fail, we clean up 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 (not shown in Algorithm 1), and finally a full defragmentation—though this is rare in practice. Memory recycling benefits typical DNN workloads with repeated operations on fixed-sized intermediates, such as the forward and backward passes of mini-batch processing. We prioritize recycling exact-sized memory chunks over temperature-based approaches to prevent GPU memory fragmentation and to avoid the costly defragmentation process. The primary objective of these steps is to avoid memory allocation, deallocation, which triggers device synchronization, and fragmentation due to repeated deallocation, however, without compromising the reuse potential.

Eviction Policy: The eviction policy determines the order in which pointers are recycled or freed from each free queue. Our policy is devised to serve typical mini-batch workloads. The eviction scoring function for a cached GPU object o is

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

where Q is a priority queue of free pointers. $T_a(o)$ denotes the normalized last access timestamp, preserving recently reused pointers from recycling (e.g. reuse within a mini-batch). $h(o)$ denotes the height of the corresponding lineage trace. The $1/h(o)$ factor preserves objects with shorter lineage traces, allowing reuse of *input data pipelines* [88] that are applied to mini-batches, where a mini-batch is sliced from the input datasets before starting the forward pass. The lineage DAGs for operations in these data pipelines are shorter since they are directly sliced from the input (compared to the operations in forward/backward paths). Finally, $c(o)$ is the estimated compute cost of object o, which enables recycling the least expensive intermediates first (e.g., element-wise `ReLU` before `Conv2d` or fully-connected layers).

5 COMPILER INTEGRATION

Runtime-level reuse and memory management are valuable, but a holistic handling at *compiler- and runtime-level* yields additional improvements. In this section, we describe optimizations for operator parallelism, data exchange, workload-aware cache management and checkpoint placement, as well as 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 related rewrites.

Prefetching Remote Objects: We introduce a new `prefetch` operator and the corresponding compiler rewrite to trigger remote (Spark/GPU) jobs and asynchronously fetch the results without blocking the CPU instruction stream. This rewrite traverses the execution plan and identifies operators that trigger remote jobs through `collect` and `cudaMemcpyDeviceToHost` calls. These operators represent the roots of remote operator chains. After each such root, the rewrite inserts a `prefetch` instruction, wrapping the triggering call, and marks it for asynchronous execution. Additionally, this rewrite flags all other Spark actions for asynchronous execution. At runtime, the operator scheduler triggers these asynchronous operations and returns `future` objects [91], allowing concurrent execution. This compiler extension enables concurrent execution of Spark jobs, local operators, and GPU kernels. Figure 9(a) shows the DAG from Example 4.1 (Figure 7) after `prefetch` placement (indicated by *pf*) for two jobs that collects results of **mm** + **di** and $\mathbf{y}^\top\mathbf{X}$, respectively.

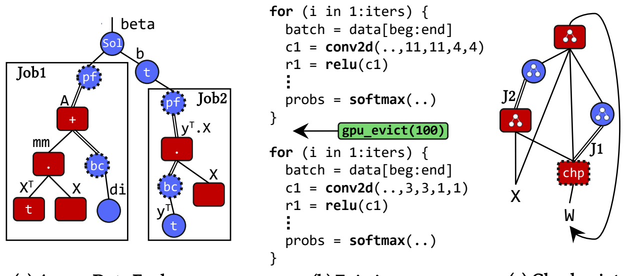
Reusing Prefetched Results: Caching results immediately after instruction execution is infeasible for asynchronous operators. To overcome, the main thread propagates the corresponding lineage trace to the spawned prefetch threads, enabling them to cache (`PUT`) the fetched data once available. The `prefetch` instruction of Job2 in Figure 9(a) caches the fetched result of $\mathbf{y}^\top\mathbf{X}$, which is then reused (e.g., in `prefetch`) in subsequent iterations.

Broadcasting Local Objects: We introduce another asynchronous operator, `broadcast`, to optimize local to remote data transfer. The corresponding rewrite places broadcast as the last operator of local operator chains. For Spark, the broadcast operator asynchronously partitions the input matrix and registers it as a broadcast variable, deferring the actual data transfer to the executors until the broadcast variable is used by Spark. Figure 9(a) shows an example of broadcast instruction (*bc*) placement.

5.2 Workload-aware Cache Management

We introduce new operators for cache management and rewrites for their placements. These operators aim to reduce caching overhead and improve the utility of reuse and memory management.

Eviction Injection: Our eviction logic, (in Section 4), operates in an incremental manner, 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 that utilizes two pre-trained 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 the `evict` instruction for cache cleanup. A program-level rewrite identifies these loop patterns and injects `evict` instructions with percentages of cache



```
for (i in 1:iters) {
  batch = data[beg:end]
  c1 = conv2d(..,11,11,4,4)
  r1 = relu(c1)
  ⋮
  probs = softmax(..)
}
for (i in 1:iters) {
  batch = data[beg:end]
  c1 = conv2d(..,3,3,1,1)
  r1 = relu(c1)
  ⋮
  probs = softmax(..)
}
```

(a) Async. Data Exchange (b) Eviction (c) Checkpoint

Figure 9: Compiler-assisted Reuse (incl. Asynchronous Data Exchange and Operations, Forced Eviction, and Checkpointing).

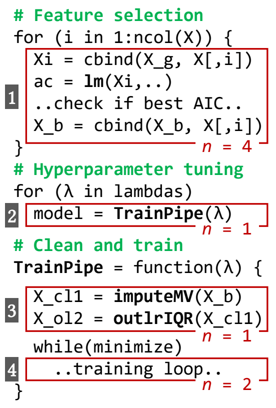
size to clean up as arguments. At runtime, the `evict` instruction utilizes the backend-specific eviction logic to free up space. The rewrite avoids full eviction if access patterns repeat. Other DNN frameworks require manual placement of such cleanup [31].

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 [10, 24] proposes simple, rule-based checkpoint (`persist`) placements. In this work, we introduce two workload-aware compiler 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 **W** of Poisson Non-negative Matrix Factorization [79]. Here, each node represents a Spark (red) or local (blue) operator chain. Every iteration updates **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 **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. Empirically, reusable operations repeat multiple times for hierarchically composed ML pipelines. Based on this observation, 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 [3]) are loop-iteration-dependent ($\mathbf{X}_i$ varies with i) and thus, is 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 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])
  ac = lm(Xi,..)
  ..check if best AIC..
  X_b = cbind(X_b, X[,i])
}                          n = 4
# Hyperparameter tuning
for (λ in lambdas)
  model = TrainPipe(λ)
                           n = 1
# Clean and train
TrainPipe = function(λ) {
  X_cl1 = imputeMV(X_b)
  X_ol2 = outlrIQR(X_cl1)
                           n = 1
  while(minimize)
    ..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 remote operator pipelines. As Algorithm 2 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 [70]. 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 Design Decisions

Our reuse and cache management for Spark apply to other distributed backends with lazy evaluation that offer primitives for distributed caching. For instance, Dask [106] and Ray [91] provide `Cache` [38] and `Checkpoint` [102] interfaces, respectively. Similarly, our memory management and reuse for GPU can be extended to other hardware accelerators like FPGAs and TPUs [64]. Here, we summarize the applicability and future work.

- *Other ML Systems:* MEMPHIS applies to other compilation-based, multi-backend systems for unified data processing [21, 49], integrated data analysis [36], polyglot, and AutoML. Our compiler and runtime optimizations like RDD reuse, cleanup, and eviction and checkpoint injection also apply to DAG compilation in TensorFlow and PyTorch.
- *Multi-GPU/stream:* SystemDS does not support multi-GPUs/GPU-streams. However, the memory manager and *prefetch* logic apply to multiple GPUs with separate caches.
- *Deeper Hierarchies:* For hierarchically-structured backends (e.g. federated workers with local GPUs), local lineage-based reuse directly applies. Prior work added lineage-based reuse to multi-tenant federated workers [19]. Future work includes efficient transfer of lineage traces.
- *No Exchange of Cached Objects:* While we implemented the mechanics for evicting cached objects from one backend to another, we observed that moving cached objects cause movements of live variables for maintaining data locality, which in turn can severely affect overall execution time.
- *Operator Scheduling:* Our scheduling strategies provide holistic memory management, but in a heuristic manner. Devising a guaranteed optimal, cost-based and reuse-aware operator placement is interesting future work.

Algorithm 2 Operator linearization

```
Input: Operator DAG $D$ with root $R$
Output: List of instructions $\mathbf{L}$
 1: if HASNOREMOTEOPS(D) then                              // All local OPs
 2:     $\mathbf{L}$ = DEPTHFIRST($D$)                          // Linearize deapth-frist
    // Step1: Identify OP chains & count operators
 3: $SRoots$ = COLLECTSPROOTS($D$)                           // Identify Spark jobs
 4: $GRoots$ = COLLECTGPROOTS($D$)                       // Identify GPU OP chains
 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($r$)             // Spark OP count/job
 9:         else if $r$ is GPU then
10:             $nGPOps[r]$ = COUNTGPOPS($r$)        // GPU OP count/chain
    // 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($r, nSPOps[r], \mathbf{L}$)          // Depth-first Spark jobs
16:     else if $r$ is GPU then
17:         DEPTHFIRST($r, nGPOps[r], \mathbf{L}$) // Depth-first GPU OP chains
    // Step3: Linearize the rest of the local OPs
18: DEPTHFIRST($R, \mathbf{L}$)                          // Place unvisited nodes of D
```

6 EXPERIMENTS

Our experiments study MEMPHIS's performance by systematically scaling data sizes, task complexity, and instruction counts across various workloads. We first conduct micro benchmarks for understanding the overhead of lineage tracing and cache probing in Spark and GPUs. Subsequently, we investigate a range of end-to-end ML pipelines, including data pre-processing, hyper-parameter optimization, matrix factorization, and DNN, for exploring the benefits of multi-backend reuse in realistic settings.

6.1 Experimental Setting

HW Environment: We ran all experiments on a cluster of 8+1 scale-out nodes and a single scale-up node. Each scale-out node has an AMD EPYC 7443P CPU @ 2.8-4.0 GHz (24 physical/48 virtual cores), and 256 GB DDR4 RAM. The scale-up node has two Intel Xeon Gold 6338 CPUs @ 2.2-3.2 GHz (64 physical/128 virtual cores), 1 TB DDR4 RAM, and two NVIDIA A40 GPUs with 48 GB and PCIe 4.0. The software stack comprises Ubuntu 20.04.6, Hadoop 3.3, Spark 3.5, OpenJDK 11, CUDA 10.2, and CUDNN 7.6.

Memory Configurations: For the Spark cluster, we use 38 GB driver memory and 230 GB executor memory. The buffer pool and operation memory are set to 20 GB and 7 GB i.e., operators requiring more than 7 GB memory are compiled to Spark instructions. The lineage cache size is set to 5 GB on the driver and 55 GB (80% of storage) on each executor. The scale-up node has 200 GB heap with 10 GB host memory for lineage cache and the full 48 GB GPU memory for the unified memory manager.

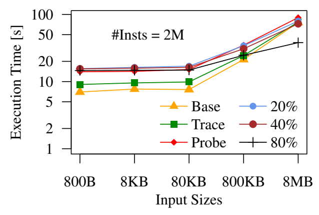
Baselines: For a comprehensive evaluation, we compare MEMPHIS with different SystemDS configurations, application-specific reuse frameworks, and state-of-the-art ML systems.

- *SystemDS:* **Base** is SystemDS without reuse. **MPH** denotes MEMPHIS with multi-level, multi-backend reuse and all optimizations. Different configurations of SystemDS (**Base-x**) and MEMPHIS (**MPH-x**) are introduced later.
- *Reuse Frameworks:* We compare with several reuse frameworks: LIMA [101] for fine-grained reuse, HELIX [125] for coarse-grained reuse, CoorDL [88] for *input data pipeline* reuse in CPU, Clipper [33] for prediction reuse, and VISTA [93] for reuse in transfer learning. For fair comparisons, we hand-optimized the ML pipelines via script-level common subexpression elimination (CSE) to reuse outputs of built-in functions (which may contain multi-backend operations), as well as data pipeline results, predictions, and extracted features for transfer learning, mimicking the capabilities of these frameworks. Script-level reuse ensures the maximum possible reuse benefits without overheads and configuration complexities from these frameworks.
- *ML System:* We compare with PyTorch 2.1 [96], a strong baseline for DNN workloads on GPUs. PyTorch's caching memory allocator also recycles memory blocks [31, 76].

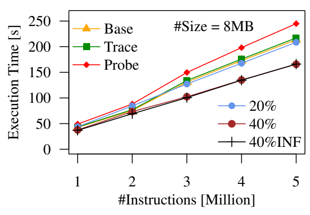
6.2 Micro Benchmarks

We first conduct micro benchmarks to study various aspects of MEMPHIS in isolation. We use representative but simplified scripts. These experiments focus on lineage tracing and reuse overhead, cache size configuration, and memory management of Spark and GPU. For the micro benchmarks, we set SystemDS's buffer pool to 100GB while keeping the operation memory constant at 7GB to prevent buffer pool eviction. While evaluating the impact of varying cache sizes in the driver, we keep remote caches in Spark and GPU unchanged due to their unified memory managers and impact on execution memory.

Reuse Overhead: To understand the overhead of lineage tracing and cache probing, we explore a hyper-parameter tuning scenario. We execute the core logic of L2SVM with varying input sizes, iteration counts, and percentages of reusable instructions. First, we fix the iteration count at 200 (total 2M instructions) and vary input sizes in [800B, 8MB]. While increasing input sizes scales the compute cost per instruction, the overhead from lineage tracing and cache management remains constant. To simulate reuse, we adjust the fraction of reusable instructions from 20% to 80% by randomly repeating hyperparameters, where the reusable instructions contain primarily binary matrix-vector computations. Figure 11(a) compares Base with different configurations. The *Trace* setting enables only tracing, introducing tracing overhead. *Probe* refers to reuse enabled but with no reusable instructions (i.e., maximum overhead with no reuse benefits). Here, for input sizes smaller than 8MB, the total execution time of Base is dominated by interpretation overhead, as well as variable and statistics management. Lineage tracing and probing further increase this overhead by 1.3x and 2x. For small input data, reuse does

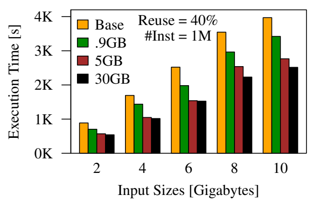


(a) Reuse Overhead - Sizes

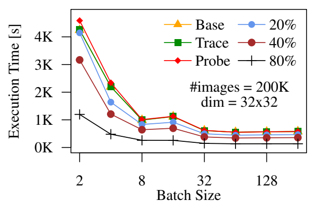


(b) Reuse Overhead - #Instructions

Figure 11: Basic Lineage Tracing and Reuse Overhead.



(a) Cache Size Comparison



(b) Tracing Overhead GPU

Figure 12: Influence of Cache Sizes on Reuse Potential and GPU Backend Tracing Overhead.

not help due to relatively low compute cost. However, for larger inputs (8MB), the tracing and probing overheads become negligible, and the reuse settings show improvements from 1.1x (20% reuse) to 3x (80% reuse). Second, to further study this scenario, we vary the instruction count from 1M to 5M for the fixed 8MB input. Increasing instruction count scales the caching related overheads. Figure 11(b) shows the probing overhead increases with instruction count, reaching 15% for 5M instructions, while the tracing overhead remains moderate. However, already 20% reuse amortizes these overheads by reusing intermediates and applying compaction (see Figure 5), while 40% reuse improves by 1.5x. We further compare a reuse setting of a larger cache with no cache eviction (40%INF). This setting does not yield any further speedup as MEMPHIS's caching policy maintains the objects with high reuse potential even in a small 5GB cache.

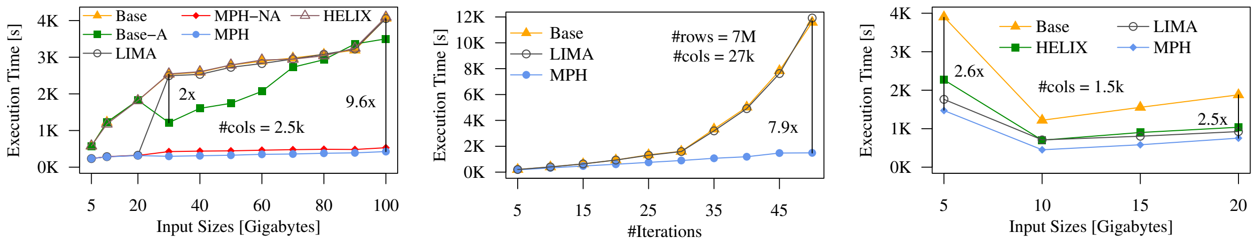
Cache Size Comparison: To evaluate the impact of cache sizes in the driver, we utilize the same experiment for varying input sizes in [2GB, 10GB], (with 1M instructions and 40% reusable), and three cache sizes: 900MB, 5GB, and 30GB. The 8GB and 10GB input matrices are distributed and produce Spark operations. Figure 12(a) shows, that even a 900MB cache consistently achieves 1.2x speedup. For smaller input sizes, 5GB and 30GB show similar speedup, while for larger inputs, the 5GB cache yields slightly less speedup compared to the 30GB cache (1.4x vs. 1.6x). The speedup comes from reusing local intermediates and Spark RDDs and actions (including `prefetch`), even after multiple evictions—proving the robustness of our eviction policies.

GPU Cache Eviction: To evaluate the GPU memory management, we utilize an ensemble convolutional neural network (CNN) scoring of 200K 32×32 images with varying batch sizes and different reuse settings (randomly repeated images identified by pixel-encoded IDs). We use two CNN models with distinct memory allocation patterns: one with two `conv2d` layers (#output channels = 64, 128) and ReLU activations, and the other with three `conv2d` layers (#output channels = 64, 192, 256) and ReLU activations. Both CNNs employ two fully-connected layers with ReLU and Softmax activation after the `conv2d` layers for classification. Figure 12(b) shows that the overhead of probing remains moderate, at 8%, even for a small batch size of 2 (2.6M GPU instructions), and is offset by only 20% reuse. From batch size 4, despite a large number of evictions (255K/139K recycled/reused pointers for batch size 4), reuse settings of 20%, 40%, and 80% yield consistent improvements of 1.3x, 1.6x, and 4x, respectively.

Conclusions: These micro benchmarks show that MEMPHIS's lineage tracing and probing overhead remain relatively low even for a large number of instructions. The robustness optimizations, such as the compaction of lineage DAGs, effectively amortize the probing overhead even when reuse is low. Moreover, the backend-specific eviction policies retain reuse benefits, despite frequent evictions and smaller cache sizes.

Table 3: Overview of ML Pipeline Use Cases & Datasets

Name	Use Case	Dataset	# Rows	# Columns	Influential Techniques
HCV	Grid Search / Cross Validation	Synthetic	[270K, 2.7M]	2.5K	Async. OPs, local & RDD reuse
PNMF	Non-negative Matrix Factorization	MovieLens	7M	27K	Checkpoint placement
HBAND	Hyperband Model Selection	Synthetic	[425K, 1.7M]	1.5K	Multi-level reuse, delayed caching
CLEAN	Data Cleaning Pipelines	APS	[60K, 7.2M]	170	Large #intermediates & #evictions
HDROP	Dropout Rate Tuning	KDD 98	95K	469	Local and GPU ptr. reuse
EN2DE	Machine Translation Inference	WMT1́4	200K	1	Recycle & reuse GPU ptrs.
TLVIS	Transfer Learning Feature Extraction	ImageNet, CIFAR-10	10K images	224 × 224, 32 × 32	Evictions & mem. management



(a) HCV (b) PMNF (c) HBAND

Figure 13: Performance of End-to-end ML Pipelines, part I.

6.3 ML Pipelines Performance

We now describe the performance impact of multi-backend reuse on end-to-end ML pipelines. For a balanced view, we evaluate a diverse set of workloads with different characteristics.

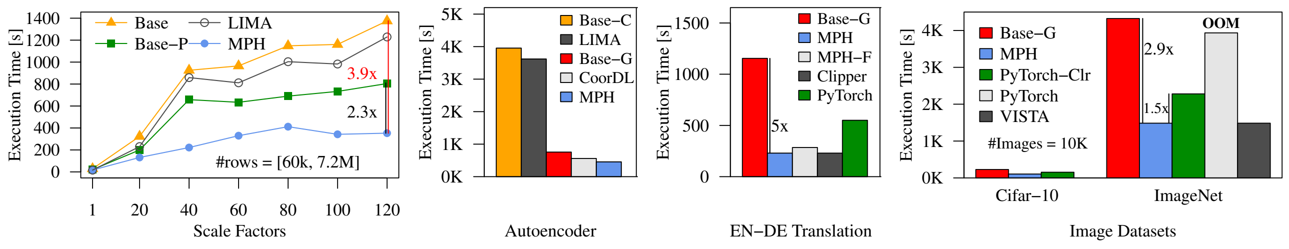
Pipeline Summary: Table 3 provides an overview of the used ML pipelines and datasets. The pipelines include: (1) Grid search hyper-parameter optimization of cross-validated linear regression (HCV); (2) Poisson non-negative matrix factorization (PNMF); (3) model search via Hyperband-like multi-armed bandit [74] and weighted ensemble learning (HBAND); (4) pipeline enumeration for data cleaning [114] (CLEAN) (5) dropout-rate optimization for Autoencoder (HDROP); (6) a pre-trained scoring network for English to German translation (EN2DE); and (7) transfer learning for computer vision models (TLVIS). All pipelines leverage SystemDS's built-in functions, with all compiler and runtime optimizations enabled for MEMPHIS. The *Influential Techniques* column highlights the most impactful optimizations and key workload characteristics for each use case.

Dataset Description: Lineage-based reuse is largely independent of data skew. We combine real and synthetic datasets to evaluate varying data characteristics. MovieLens [54] is a movie rating dataset containing 20M ratings (138K unique users, 27K unique movies). For pre-processing, we integer encode and replicate rows. APS [1] is a classification dataset, collected from components of SCANIA trucks, for classifying Air Pressure System (APS) failures. This dataset has 60K entries and 0.6% missing values. KDD98 is a regression dataset for the return from donation campaigns. We perform recoding on categorical and binning (10 equi-width bins) on numerical features. The WMT2014 dataset [27, 81] for English-to-German translation comprises news and web crawls. We use a 200K-word subset of this dataset. ImageNet [4] and CIFAR-10 [67] are popular image datasets. For pre-processing, we resized the ImageNet images to 224×224×3 and CIFAR-10 images to 32×32×3, then linearized these images.

Cross-Validation: HCV calls a cross-validated linear regression with Example 4.1 at its core, for 10 different regularization parameters. HCV uses R^2 to find the best parameters. We vary the input size in [5 GB, 20 GB] and compare Base, LIMA, HELIX, and MEMPHIS, where Base-A and MPH are with and Base and MPH-NA are without the asynchronous operators. Figure 13(a) shows MPH is 9.6x faster than Base by reusing $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$ for each fold and executing concurrent jobs through `prefetch`. Although, MPH uses 2x more cluster memory compared to Base for storing the cached RDDs, the extra memory usage does not impact the execution performance. Starting from 25 GB input size, $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$ are placed in Spark. LIMA reuses only local intermediates up to 20 GB, whereas MPH reuses in all settings. HELIX performs similar to Base as HCV has no coarse-grained reuse opportunities (i.e., no repeating top-level ML tasks). Base-A yields 2x speedup due to concurrent execution, but the speedup decreases for larger inputs (#partitions > #cores). MPH is 20% faster than MPH-NA due to parallel operator execution.

Matrix Factorization: Poisson Non-negative Matrix Factorization (PNMF) [79] factorizes a matrix $\mathbf{X}$ into factors $\mathbf{W}$ and $\mathbf{H}$ such that $\mathbf{X} \approx \mathbf{WH}$. PNMF iteratively updates $\mathbf{W}$ and $\mathbf{H}$ via its update rules. For PNMF on the MovieLens dataset with rank 100, $\mathbf{W}$ (7M × 100) becomes distributed—as shown in Figure 9(c)—while $\mathbf{H}$ (100 × 27K) remains local. We vary the number of iterations to study the performance impact of the compiler-placed checkpoints (`persist`). Figure 13(b) shows that, as the iteration count exceeds 30, Base and LIMA significantly slow down due to the increasing size of the Spark jobs re-executing all previous iterations. However, MPH yields a 7.9x speedup (with 4x more memory usage) by persisting $\mathbf{W}$ in each iteration (see Section 5.2).

Model Search: HBAND has two phases. We first utilize successive halving to fine-tune the hyper-parameters of L2SVM and multi-class logistic regression (MLRG). The inner loop performs a grid search over regularization parameters (`reg`) and intercept options (0, 1, or 2), while the outer loop iterates through five brackets, each halving the `reg` list (starting with 25 values) and doubling the iteration count (starting from 10). We then apply weighted ensemble learning to combine the best L2SVM and MLRG models. The ensemble weights are optimized via a random search over 1K weight configurations. Figure 13(c) shows the results for varying data sizes. MPH yields 2.6x/2.5x speedups for 5 GB/20 GB inputs over Base by reusing the previous iterations of successive halving and the **XB** multiplication in weighted class probability computations. Reused objects include many RDDs (≈ 4K), Spark actions (≈ 2K), and local matrices (≈ 40K). Our compiler enabled *delayed caching* for RDDs for MPH. MEMPHIS



(a) CLEAN (b) HDROP (c) EN2DE (d) TLVIS

Figure 14: Performance of End-to-end ML Pipelines, part II.

is 40% faster than HELIX (reuses **XB** operators) and LIMA. Even for local operations (5 GB), MPH is 20% faster than LIMA.

Data Cleaning: CLEAN enumerates data cleaning pipelines, leveraging the downstream ML algorithm for feedback. We construct 12 pipelines combining primitives for missing value imputation (`imputeByMean`, `imputeByMode`), outlier detection (`outlierByIQR`), normalization (`standard scaling`, `min-max`), class imbalance (`underSampling`), and dimensionality reduction (PCA) for a downstream L2SVM task and return the top-3 pipelines. The order of the primitives in each pipeline is data-dependent, (e.g., imputation and outlier removal before normalization). These pipelines can be auto-generated using AutoML [114]. We vary the data size [60K, 7.2M] using a scale factor, which utilizes row append to replicate the input data, and compare Base, Base with parallel feature processing using multi-threading [23] (Base-P), and MEMPHIS. Figure 14(a) shows, for scale factor 120, MPH yields 3.9x/3.5x/2.3x speedups over Base/LIMA/Base-P by reusing the repeating primitives. Most cleaning primitives operate feature-wise, producing many intermediates, which leads to repeated cache spills. However, our cache eviction logic continues to reuse the local and Spark operator outputs.

Dropout-Rate Tuning: HDROP tunes the dropout rate from 5% to 50% of an Autoencoder (AE) using grid search. The AE has two hidden layers of sizes 500 and 2 and a dropout layer. For each dropout rate, we train the AE for 10 epochs with batch size 256. We construct an input data pipeline (IDP) with normalization and a feature transformation map of binning, recoding, one-hot encoding, and apply this pipeline batch-wise in every iteration. Figure 14(b) shows the results of comparing Base with CPU instructions (Base-C), Base with CPU and GPU instructions (Base-G), LIMA, CoorDL, and MEMPHIS on the KDD98 dataset. Although HDROP has modest reuse opportunities, MPH achieves 1.7x speedup over Base-G by reusing the batch-wise IDP across epochs. The feature transformation is reused on the host, while the normalization is reused on the GPU. CoorDL, as a representative of IDP reuse frameworks, only reuses the CPU component of the IDP and is 24% slower than MPH. This experiment is another example of efficient fine-grained reuse across multiple backends.

Language Translation: The EN2DE pipeline comprises GPU scoring with a pre-trained language translation model, and exhibits fine-grained prediction caching potential [33]. We use pre-trained word embeddings for English and German and their dictionaries, along with a pre-trained model, comprising four fully-connected layers with ReLU and Softmax for translation. The word embeddings for both English and German are trained on Wikipedia and 300-dimensional. The input is a 200K-word sequence of news content. Figure 14(c) compares Base with GPU instructions (Base-G), operator-at-a-time reuse (MPH-F), which disables multi-level reuse (see Section 3.3), Clipper, and MEMPHIS. MPH yields a 5x speedup over Base-G by reusing scoring results at the host—completely eliminating GPU computations for the reused words. In contrast, MPH-F reuses GPU pointers for the reusable instructions, yielding a 4x speedup, despite frequent evictions including 325K recycled pointers. Clipper performs similar to MPH by reusing the predictions at the host.

Transfer Learning: Transfer learning is a widely used alternative to training large models from scratch. Practitioners often evaluate different pre-trained models to identify the most suitable model-layer pair for the downstream task and input dataset [93, 104]. TLVIS utilizes three pre-trained CNN models: AlexNet [68], VGG16 [115], and ResNet18 [57], and extracts the following layers (by applying the pre-trained weights on the test images) for transfer learning: `Conv2d` of layer4 (Conv4) to fully-connected layer7 (FC7) from AlexNet, Conv5 to FC7 layers from VGG and the last four residual blocks from ResNet. The pre-trained layers are frozen during feature extraction, and a proxy of a linear classifier is used to rank the extracted features [95, 119]. We split the CIFAR-10 and ImageNet datasets into non-overlapping training and test sets, with 10K images in each test set, and we execute the pipelines in the GPU. Figure 14(d) shows that MEMPHIS yields 2x and 3x speedups for CIFAR-10 and ImageNet test sets by reusing the intermediates during repetitive feature extractions. MEMPHIS compiles an `evict(100)` instruction between models (*eviction injection*) to free up the allocated pointers, and the GPU memory manager efficiently recycles the intermediates without compromising the reuse benefits. For instance, 30K and 17.5K pointers are reused and recycled for ImageNet, respectively. VISTA, as a specialized system for transfer learning, performs similar to MPH by applying CSE across pipelines.

Comparison with PyTorch: For comparison with another ML system, we compare MEMPHIS with PyTorch for EN2DE and TLVIS. All intermediates are represented as `torch` tensors. For TLVIS, we utilize PyTorch's pre-trained models and API to freeze layers. Additionally, we transfer the model parameters and datasets to the GPU before starting the mini-batch processing to enable PyTorch's compiler optimizations and speculative GPU memory management [76]. PyTorch with `torch.compile` (PyTorch) fails with out-of-memory and requires manual cleanups of the allocation cache (`empty_cache()`) after each model [31, 32] (PyTorch-Clr). Figures 14(c) and 14(d) show that PyTorch—as a specialized system for DNN training—is 2x/1.9x faster than Base-G for EN2DE and TLVIS. However, PyTorch fails to reuse predictions and repeated feature extractions and is 2.4x and 1.5x slower than MPH for EN2DE and TLVIS, respectively.

Conclusion: Overall, our workload-aware compiler-assisted reuse, optimizations, inter-backend parallelism, and memory management show competitive performance compared to other systems, which include ML systems as well as specialized application-specific reuse frameworks, demonstrating our framework's effectiveness in a broader range of scenarios.

7 RELATED WORK

Our compiler-assisted, multi-backend reuse in ML systems is related to the reuse of ML and query intermediates, RDD caching, GPU memory management, and operator scheduling.

Reuse of ML Pipeline Intermediates: Prior work has recognized significant reuse opportunities in exploratory data science workflows. Columbus [130] and KeystoneML [116] automatically materialize and reuse selected intermediates within and across ML pipelines. Subsequent work—including Alpine Meadow [112], HELIX [125], collaborative optimizer [39], HYPPO [66] and, result sharing for what-if analysis [51]—apply CSE and cost-based materialization for reusing ML task results such as data preparation, feature engineering, and model training. Other work focuses on reuse within specific workloads: tf.Data [92], Cachew [53], CoorDL [88], and TensorSocket [105] reuse pre-processed mini-batches in DNN training; Clipper [33] and PRETZEL [73] apply CSE and caching to predictions; MISTIQUE [123] enables caching and querying model intermediates; as well as OneAccess [65], Ease.ml [77], and SystemDS federated [19] enable multi-tenant resource sharing in ML clusters. Most of these systems only support coarse-grained reuse (of top-level primitives) and apply reuse in an isolated manner (independent of multiple backends, operator scheduling, and memory management).

Reuse of Query Intermediates: Our work on reuse is inspired by the extensive research on reusing query intermediates in database systems. Reuse is used for various aspects including buffer pool page caching in CPU/GPU memory [82], scan sharing [13, 121], request batching [72], subexpression reuse [63], materialized views [6, 63], as well as multi-query optimization [107]. Work on recycling intermediates in MonetDB [60], transient materialized views [135], and Firebolt [9] reuse previously materialized in-memory or spool intermediates. CoGaDB [28] leverages a similar reuse of GPU intermediates, exploiting data locality. In contrast to this line of work, MEMPHIS provides holistic multi-backend reuse for linear algebra programs.

RDD Caching and Reuse: Prior work on RDD caching largely falls into three categories. First, heuristics-based caching utilizes known data dependencies to select intermediates for caching. SystemML [24] injects `persist()` directives after persistent reads (text to binary) and before loops for read-only variables. Emma [10] caches RDDs that are referenced multiple times. Second, DAG-aware eviction policies such as Least Reference Count (LRC) [127, 128], and Most Reference Distance (MRD) [99] extend Spark's default LRU to prioritize frequently-used RDDs. LRC monitors the number of consumers for each RDD and evicts partitions of those with the least active consumers. MRD considers both the number and distance (number of stages) of consumers, evicting RDDs with fewer active consumers farther away in the DAG. Third, caching frameworks such as Juggler [8], Agile-Ant [7], and MEMTUNE [126] optimize distributed caching decisions in a cost-based manner by analyzing data dependencies and access patterns for individual Spark jobs. In contrast, MEMPHIS identifies fine-grained redundancy at runtime and reuses RDDs across jobs and conditional control flow programs.

GPU Memory Management: GPUs are widely used for DNN workloads and many specialized techniques for GPU memory management exist. GPU memory is a limiting factor in DNN design [50, 57, 68], and many DNN job failures are caused by running out of GPU memory due to large batch sizes, skewed memory usage across layers, memory fragmentation, and mispredicting intermediate sizes [131]. TensorFlow [118] and JAX [14] preallocate nearly all GPU memory to avoid memory fragmentation and frequent calls to `cudaMalloc` and `cudaFree`. PyTorch [132] uses a pool allocator [71], which recycles free memory pointers. Prior work [29, 61, 111, 133] on materialization also discards activations in the forward pass to reduce memory pressure and recomputes them during the backward pass. Further, asynchronous swapping [58, 84, 98] allows offloading activations from GPU memory to host, freeing up memory for ongoing computations and re-loading them on demand. Another line of work proposes more accurate estimators for the size of intermediates [46, 110]. These techniques are designed for single DNN training and are independent of reusing intermediates in ML pipelines with multiple training and inference tasks.

Operator Scheduling: The performance impact of operator scheduling has been investigated before. GPipe [59] and DAPPLE [44] explore pipeline parallelism across mini-batches, and partition the operators to increase parallelism and remove pipeline bubbles (overlapping computation and communication). PipeDream [94] extends this work for asynchronous convergence by scheduling the mini-batches based on a brief GPU profiling and multiple versions of weights. Alpa [134] combines data and model parallelism for large DNNs by automatically compiling cost-optimal plans for data and model parallelism. CoCoNet [62] introduced a domain-specific language for specifying combinations of computation and communication, along with a compiler that generates optimized custom kernels. Additionally, NVIDIA offers higher-level programming abstractions [41] and a unified memory manager including pre-fetching capabilities [56]. In contrast, MEMPHIS's operator ordering and asynchronous operators exploit parallelism across multiple backends, integrate lineage-based reuse, and handle data-centric ML pipelines ranging from pre-processing to training.

8 CONCLUSIONS

To summarize, we introduced MEMPHIS as a holistic framework for efficient, multi-backend reuse of intermediates and memory management in ML systems. Our hierarchical lineage cache seamlessly allows reuse across heterogeneous backends. We devised tailor-made eviction policies for Spark and GPUs, along with related memory management techniques. Our compiler extensions improve reuse potential, reduce runtime overheads, and enable concurrent execution. Our experiments have shown robust improvements across diverse workloads. In conclusion, the increasing complexity of ML pipelines, coupled with diverse data modalities and heterogeneous backends, inevitably introduces redundancy across different backends. This growing complexity and heterogeneity poses challenges for library developers and users, as manual redundancy elimination becomes infeasible. Our compiler-assisted, runtime-based multi-backend lineage cache effectively addresses these challenges. The underlying concepts and optimization techniques are broadly applicable in modern ML systems. Interesting future work includes (1) exploring cost-based operator scheduling for reuse-aware operator placement, (2) query processing over lineage traces for model debugging, and (3) extending lineage tracing for fairness constraints [52].

9 ACKNOWLEDGMENTS

We gratefully acknowledge funding from the German Federal Ministry of Education and Research (under research grant BIFOLD24B).

REFERENCES

[1] 2017. APS Failure at Scania Trucks. UCI Machine Learning Repository. https://doi.org/10.24432/C51S51

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

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

[4] Wendy Kan Addison Howard, Eunbyung Park. 2018. ImageNet Object Localization Challenge. https://kaggle.com/competitions/imagenet-object-localization-challenge

[5] Divy Agrawal et al. 2018. RHEEM: Enabling Cross-Platform Data Processing - May The Big Data Be With You! -. *PVLDB* 11, 11 (2018), 1414–1427. https://doi.org/10.14778/3236187.3236195

[6] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*. 496–505. http://www.vldb.org/conf/2000/P496.pdf

[7] Hani Al-Sayeh, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2024. AgileAnt: Self-managing Distributed Cache Management for Cost Optimization of Big Data Applications. *PVLDB* 17, 11 (2024), 3151–3164.

[8] Hani Al-Sayeh, Bunjamin Memishi, Muhammad Attahir Jibril, Marcus Paradies, and Kai-Uwe Sattler. 2022. Juggler: Autonomous Cost Optimization and Performance Prediction of Big Data Applications. In *SIGMOD*. 1840–1854. https://doi.org/10.1145/3514221.3517892

[9] Alex Hall. 2024. Caching & Reuse of Subresults across Queries. https://www.firebolt.io/blog/caching-reuse-of-subresults-across-queries

[10] Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Felix Schüler, Lauritz Thamsen, Odej Kao, Tobias Herb, and Volker Markl. 2015. Implicit Parallelism through Deep Language Embedding. In *SIGMOD*. 47–61. https://doi.org/10.1145/2723372.2750543

[11] Jason Ansel. 2022. *TorchDynamo*. https://github.com/pytorch/torchdynamo

[12] Antreas Antoniou, Amos J. Storkey, and Harrison Edwards. 2018. Augmenting Image Classifiers Using Data Augmentation Generative Adversarial Networks. In *ICANN (Lecture Notes in Computer Science, Vol. 11141)*. 594–603. https://doi.org/10.1007/978-3-030-01424-7_58

[13] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Leopoldo Perez. 2010. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD*. ACM, 519–530. https://doi.org/10.1145/1807167.1807224

[14] The JAX Authors. 2023. GPU Memory Allocation. https://jax.readthedocs.io/en/latest/gpu_memory_allocation.html

[15] Reza Babanezhad, Mohamed Osama Ahmed, Alim Virani, Mark Schmidt, Jakub Konečný, and Scott Sallinen. 2015. StopWasting My Gradients: Practical SVRG. In *NeurIPS*. 2251–2259. https://proceedings.neurips.cc/paper/2015/hash/a50abba8132a77191791390c3eb19fe7-Abstract.html

[16] Paul Barham et al. 2022. Pathways: Asynchronous Distributed Dataflow for ML. In *MLSys*, Diana Marculescu, Yuejie Chi, and Carole-Jean Wu (Eds.).

[17] Nirvik Baruah, Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia. 2022. Parallelism-Optimizing Data Placement for Faster Data-Parallel Computations. *PVLDB* 16, 4 (2022), 760–771. https://doi.org/10.14778/3574245.3574260

[18] Sebastian Baunsgaard et al. 2021. ExDRa: Exploratory Data Science on Federated Raw Data. In *SIGMOD*. 2450–2463. https://doi.org/10.1145/3448016.3457549

[19] Sebastian Baunsgaard, Matthias Boehm, Kevin Innerebner, Mito Kehayov, Florian Lackner, Olga Ovcharenko, Arnab Phani, Tobias Rieger, David Weissteiner, and Sebastian Benjamin Wrede. 2022. Federated Data Preparation, Learning, and Debugging in Apache SystemDS. In *CIKM*. 4813–4817. https://doi.org/10.1145/3511808.3557162

[20] Denis Baylor et al. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *KDD*. 1387–1395. https://doi.org/10.1145/3097983.3098021

[21] Kaustubh Beedkar, Bertty Contreras-Rojas, Haralampos Gavriilidis, Zoi Kaoudi, Volker Markl, Rodrigo Pardo-Meza, and Jorge-Arnulfo Quiané-Ruiz. 2023. Apache Wayang: A Unified Data Analytics Framework. *SIGMOD Rec.* 52, 3 (2023), 30–35. https://doi.org/10.1145/3631504.3631510

[22] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

[23] Matthias Boehm et al. 2014. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB* 7, 7 (2014), 553–564. https://doi.org/10.14778/2732286.2732292

[24] Matthias Boehm et al. 2016. SystemML: Declarative Machine Learning on Spark. *PVLDB* 9, 13 (2016), 1425–1436. https://doi.org/10.14778/3007263.3007279

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

[26] Matthias Boehm, Arun Kumar, and Jun Yang. 2019. *Data Management in Machine Learning Systems*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00895ED1V01Y201901DTM057

[27] Ondřej Bojar et al. 2014. Findings of the 2014 Workshop on Statistical Machine Translation. In *Workshop on Statistical Machine Translation*. 12–58.

[28] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust Query Processing in Co-Processor-accelerated Databases. In *SIGMOD*. 1891–1906. https://doi.org/10.1145/2882903.2882936

[29] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). http://arxiv.org/abs/1604.06174

[30] Dami Choi and George Dahl. 2020. Speeding Up Neural Network Training with Data Echoing. https://ai.googleblog.com/2020/05/speeding-up-neural-network-training.html

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

[32] PyTorch Contributors. 2023. PyTorch out of memory error. https://pytorch.org/docs/stable/notes/faq.html#my-model-reports-cuda-runtime-error-2-out-of-memory

[33] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *NSDI*. 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[34] Ekin D. Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. 2019. AutoAugment: Learning Augmentation Strategies From Data. In *CVPR*. 113–123. https://doi.org/10.1109/CVPR.2019.00020

[35] Ashok Cutkosky and Róbert Busa-Fekete. 2018. Distributed Stochastic Optimization via Adaptive SGD. In *NeurIPS*. 1914–1923. https://proceedings.neurips.cc/paper/2018/hash/5c936263f3428a40227908d5a3847c0b-Abstract.html

[36] Patrick Damme et al. 2022. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. In *CIDR*.

[37] Piali Das et al. 2020. Amazon SageMaker Autopilot: a white box AutoML solution at scale. In *DEEM@SIGMOD*, Sebastian Schelter, Steven Whang, and Julia Stoyanovich (Eds.). 2:1–2:7. https://doi.org/10.1145/3399579.3399870

[38] Dask Development Team. 2023. *Dask: Opportunistic Caching*. https://docs.dask.org/en/stable/caching.html

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

[40] Apache Beam Developers. 2024. *The Unified Apache Beam Model*. https://beam.apache.org/

[41] Thrust developers. 2023. *NVIDIA Thrust*. https://docs.nvidia.com/cuda/thrust/index.html

[42] Spark development team. 2023. Spark DAGScheduler. https://github.com/apache/spark/blob/6b7527e381591bcd51be205853aea3e349893139/core/src/main/scala/org/apache/spark/scheduler/DAGScheduler.scala#L86.

[43] Spark development team. 2024. Spark TorrentBroadcast. https://github.com/apache/spark/blob/9a1fc112677f98089d946b3bf4f52b33ab0a5c23/core/src/main/scala/org/apache/spark/broadcast/TorrentBroadcast.scala#L41.

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

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

[46] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. 2020. Estimating GPU memory consumption of deep learning models. In *ESEC/FSE*. 1342–1352. https://doi.org/10.1145/3368089.3417050

[47] Haralampos Gavriilidis, Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. In-Situ Cross-Database Query Processing. In *ICDE*. 2794–2807. https://doi.org/10.1109/ICDE55515.2023.00214

[48] Victor Giannakouris, Nikolaos Papailiou, Dimitrios Tsoumakos, and Nectarios Koziris. 2016. MuSQLE: Distributed SQL query execution over multiple engine environments. In *IEEE BigData*. 452–461. https://doi.org/10.1109/BIGDATA.2016.7840636

[49] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. 2015. Musketeer: all for one, one for all in data processing systems. In *EuroSys*. 2:1–2:16. https://doi.org/10.1145/2741948.2741968

[50] Aidan N. Gomez, Mengye Ren, Raquel Urtasun, and Roger B. Grosse. 2017. The Reversible Residual Network: Backpropagation Without Storing Activations. In *NeurIPS*. 2214–2224. https://proceedings.neurips.cc/paper/2017/hash/f9be311e65d81a9ad8150a60844bb94c-Abstract.html

[51] Stefan Grafberger, Paul Groth, and Sebastian Schelter. 2023. Automating and Optimizing Data-Centric What-If Analyses on Native Machine Learning Pipelines. *Proc. ACM Manag. Data* 1, 2 (2023), 128:1–128:26. https://doi.org/10.1145/3589273

[52] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. 2021. MLINSPECT: A Data Distribution Debugger for Machine Learning Pipelines. In *SIGMOD*. 2736–2739.

[53] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. 2022. Cachew: Machine Learning Input Data Processing as a Service. In *ATC*. 689–706.

[54] GroupLens. 2016. MovieLens 20M Dataset. https://www.kaggle.com/datasets/grouplens/movielens-20m-dataset?select=rating.csv

[55] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and

Computation in Datacenters. In *OSDI*. 75–88. http://www.usenix.org/events/osdi10/tech/full_papers/Gunda.pdf
[56] Mark Harris. 2023. *Unified Memory for CUDA Beginners*. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/
[57] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778. https://doi.org/10.1109/CVPR.2016.90
[58] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems using Integer Linear Programming. In *ASPLOS*. 875–890. https://doi.org/10.1145/3373376.3378465
[59] Yanping Huang et al. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*. 103–112.
[60] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. 2009. An architecture for recycling intermediates in a column-store. In *SIGMOD*. 309–320. https://doi.org/10.1145/1559845.1559879
[61] Paras Jain et al. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *MLSys*.
[62] Abhinav Jangda et al. 2022. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *ASPLOS*. 402–416. https://doi.org/10.1145/3503222.3507778
[63] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. *PVLDB* 11, 7 (2018), 800–812. https://doi.org/10.14778/3192965.3192971
[64] Norman P. Jouppi et al. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *ISCA*. 82:1–82:14. https://doi.org/10.1145/3579371.3589350
[65] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. 2019. The Case for Unifying Data Loading in Machine Learning Clusters. In *HotCloud*. https://www.usenix.org/conference/hotcloud19/presentation/kakaraparthy
[66] Antonios Kontaxakis, Dimitris Sacharidis, Alkis Simitsis, Alberto Abelló, and Sergi Nadal. 2024. HYPPO: Using Equivalences to Optimize Pipelines in Exploratory Machine Learning. In *ICDE*. 221–234.
[67] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. *Technical Report TR-2009, University of Toronto, Toronto.*
[68] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NeurIPS*. 1106–1114.
[69] Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. 2021. Oort: Efficient Federated Learning via Guided Participant Selection. In *OSDI*. 19–35. https://www.usenix.org/conference/osdi21/presentation/lai
[70] Lukas Landgraf, Wolfgang Lehner, Florian Wolf, and Alexander Boehm. 2022. Memory Efficient Scheduling of Query Pipeline Execution. In *CIDR*. https://www.cidrdb.org/cidr2022/papers/p82-landgraf.pdf
[71] Chris Lattner and Vikram S. Adve. 2005. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *SIGPLAN*. 129–142. https://doi.org/10.1145/1065010.1065027
[72] Rubao Lee, Minghong Zhou, and Huaming Liao. 2007. Request Window: an Approach to Improve Throughput of RDBMS-based Data Integration System by Utilizing Data Sharing Across Concurrent Distributed Queries. In *PVLDB*. 1219–1230. http://www.vldb.org/conf/2007/papers/industrial/p1219-lee.pdf
[73] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *OSDI*. 611–626. https://www.usenix.org/conference/osdi18/presentation/lee
[74] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *J. Mach. Learn. Res.* 18 (2017), 185:1–185:52. http://jmlr.org/papers/v18/16-558.html
[75] Liam Li, Kevin G. Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2020. A System for Massively Parallel Hyperparameter Tuning. In *MLSys*. https://proceedings.mlsys.org/book/303.pdf
[76] Shen Li et al. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *PVLDB* 13, 12 (2020), 3005–3018. https://doi.org/10.14778/3415478.3415530
[77] Tian Li, Jie Zhong, Ji Liu, Wentao Wu, and Ce Zhang. 2018. Ease.ml: Towards Multi-tenant Resource Sharing for Machine Learning Workloads. *PVLDB* 11, 5 (2018), 607–620. https://doi.org/10.1145/3187009.3177737
[78] Edo Liberty et al. 2020. Elastic Machine Learning Algorithms in Amazon SageMaker. In *SIGMOD*. 731–737. https://doi.org/10.1145/3318464.3386126
[79] Chao Liu, Hung-chih Yang, Jinliang Fan, Li-wei He, and Yi-Min Wang. 2010. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *WWW*. 681–690. https://doi.org/10.1145/1772690.1772760
[80] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *NIPS*. 4765–4774.
[81] Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *EMNLP*. 1412–1421. https://doi.org/10.18653/V1/D15-1166
[82] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In *SIGMOD*. 1017–1032. https://doi.org/10.1145/3514221.3517911
[83] Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. 2016. Towards Automatically-Tuned Neural Networks. In *(ICML*, Vol. 64. 58–65. http://proceedings.mlr.press/v64/mendoza_towards_2016.html
[84] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. 2017. Training Deeper Models by GPU Memory Optimization on TensorFlow. In *ML System@NeurIPS Workshop*.
[85] Xiangrui Meng et al. 2016. MLlib: Machine Learning in Apache Spark. *JMLR* 17 (2016), 34:1–34:7.
[86] Fraser Mince, Dzung Dinh, Jonas Kgomo, Neil Thompson, and Sara Hooker. 2023. The Grand Illusion: The Myth of Software Portability and Implications for ML Progress. In *NeurIPS 2023*. http://papers.nips.cc/paper_files/paper/2023/hash/42c40aff7814e9796266e12053b1c610-Abstract-Conference.html
[87] Azalia Mirhoseini et al. 2017. Device Placement Optimization with Reinforcement Learning. In *ICML*, Doina Precup and Yee Whye Teh (Eds.). 2430–2439. http://proceedings.mlr.press/v70/mirhoseini17a.html
[88] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and Mitigating Data Stalls in DNN Training. *PVLDB* 14, 5 (2021), 771–784. https://doi.org/10.14778/3446095.3446100
[89] Hubert Mohr-Daurat, Xuan Sun, and Holger Pirk. 2023. BOSS - An Architecture for Database Kernel Composition. *PVLDB* 17, 4 (2023), 877–890.
[90] Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. 2019. AutoGraph: Imperative-style Coding with Graph-based Performance. In *MLSys*. https://proceedings.mlsys.org/book/272.pdf
[91] Philipp Moritz et al. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*. 561–577.
[92] Derek Gordon Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. 2021. tf.data: A Machine Learning Data Processing Framework. *PVLDB* 14, 12 (2021), 2945–2958. https://doi.org/10.14778/3476311.3476374
[93] Supun Nakandala and Arun Kumar. 2020. Vista: Optimized System for Declarative Feature Transfer from Deep CNNs at Scale. In *SIGMOD*. 1685–1700. https://doi.org/10.1145/3318464.3389709
[94] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *SOSP*. 1–15.
[95] Cuong V. Nguyen, Tal Hassner, Matthias W. Seeger, and Cédric Archambeau. 2020. LEEP: A New Measure to Evaluate Transferability of Learned Representations. In *ICML*. 7294–7305. http://proceedings.mlr.press/v119/nguyen20b.html
[96] Adam Paszke et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 8024–8035.
[97] Fabian Pedregosa et al. 2011. Scikit-learn: Machine Learning in Python. *JMLR* 12 (2011), 2825–2830.
[98] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *ASPLOS*. 891–905. https://doi.org/10.1145/3373376.3378505
[99] Tiago B. G. Perez, Xiaobo Zhou, and Dazhao Cheng. 2018. Reference-distance Eviction and Prefetching for Cache Management in Spark. In *ICPP*. 88:1–88:10.
[100] Arnab Phani, Lukas Erlbacher, and Matthias Boehm. 2022. UPLIFT: Parallelization Strategies for Feature Transformations in Machine Learning Workloads. *PVLDB* 15, 11 (2022), 2929–2938. https://doi.org/10.14778/3551793.3551842
[101] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. In *SIGMOD*. 1426–1439. https://doi.org/10.1145/3448016.3452788
[102] Ray Development Team. 2023. *Ray: Saving and Loading Checkpoints*. https://docs.ray.io/en/latest/train/user-guides/checkpoints.html
[103] Sergey Redyuk, Zoi Kaoudi, Volker Markl, and Sebastian Schelter. 2021. Automating Data Quality Validation for Dynamic Data Ingestion. In *EDBT*. 61–72. https://doi.org/10.5441/002/EDBT.2021.07
[104] Cédric Renggli, Xiaozhe Yao, Luka Kolar, Luka Rimanic, Ana Klimovic, and Ce Zhang. 2022. SHiFT: An Efficient, Flexible Search Engine for Transfer Learning. *PVLDB* 16, 2 (2022), 304–316. https://doi.org/10.14778/3565816.3565831
[105] Ties Robroek, Neil Kim Nielsen, and Pınar Tözün. 2024. TensorSocket: Shared Data Loading for Deep Learning Training. *CoRR* abs/2409.18749 (2024). https://arxiv.org/abs/2409.18749
[106] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *SciPy*. 130 – 136.
[107] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD*. 249–260. https://doi.org/10.1145/342009.335419
[108] Svetlana Sagadeeva and Matthias Boehm. 2021. SliceLine: Fast, Linear-Algebra-based Slice Finding for ML Model Debugging. In *SIGMOD*. 2290–2299. https://doi.org/10.1145/3448016.3457323
[109] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Bießmann, and Andreas Grafberger. 2018. Automating Large-Scale Data Quality Verification. *PVLDB* 11, 12 (2018), 1781–1794.
[110] Taro Sekiyama, Takashi Imamichi, Haruki Imai, and Rudy Raymond. 2018. Profile-guided memory optimization for deep neural networks. *CoRR* abs/1804.10001 (2018). http://arxiv.org/abs/1804.10001

[111] Aashaka Shah, Chao-Yuan Wu, Jayashree Mohan, Vijay Chidambaram, and Philipp Krähenbühl. 2021. Memory Optimization for Deep Networks. In *ICLR*. https://openreview.net/forum?id=bnY0jm4l59

[112] Zeyuan Shang et al. 2019. Democratizing Data Science through Interactive Curation of ML Pipelines. In *SIGMOD*. 1171–1188. 10.1145/3299869.3319863

[113] Noam Shazeer et al. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *NeurIPS*. 10435–10444.

[114] Shafaq Siddiqi, Roman Kern, and Matthias Boehm. 2024. Saga: A Scalable Framework for Optimizing Data Cleaning Pipelines for Machine Learning Applications. *Proc. ACM Manag. Data* (2024). https://doi.org/10.1145/3617338

[115] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*.

[116] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. 2017. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *ICDE*. 535–546. https://doi.org/10.1109/ICDE.2017.109

[117] Nils Strassenburg, Dominic Kupfer, Julia Kowal, and Tilmann Rabl. 2023. Efficient Multi-Model Management. In *EDBT*. 457–463. https://doi.org/10.48786/EDBT.2023.37

[118] TensorFlow Development Team. 2023. Use a GPU. https://www.tensorflow.org/guide/gpu#limiting_gpu_memory_growth

[119] Anh Tuan Tran, Cuong V. Nguyen, and Tal Hassner. 2019. Transferability and Hardness of Supervised Classification Tasks. In *ICCV*. 1395–1405. https://doi.org/10.1109/ICCV.2019.00148

[120] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. 2023. FastFlow: Accelerating Deep Learning Model Training with Smart Offloading of Input Data Pipeline. *PVLDB* 16, 5 (2023), 1086–1099. https://doi.org/10.14778/3579075.3579083

[121] Philipp Unterbrunner, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. 2009. Predictable Performance for Unpredictable Workloads. *PVLDB* 2, 1 (2009), 706–717. https://doi.org/10.14778/1687627.1687707

[122] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Comput. Sci. Eng.* 13, 2 (2011), 22–30.

[123] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. In *SIGMOD*. 1285–1300. https://doi.org/10.1145/3183713.3196934

[124] Yongji Wu, Matthew Lentz, Danyang Zhuo, and Yao Lu. 2022. Serving and Optimizing Machine Learning Workflows on Heterogeneous Infrastructures. *PVLDB* 16, 3 (2022), 406–419. https://doi.org/10.14778/3570690.3570692

[125] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya G. Parameswaran. 2018. Helix: Holistic Optimization for Accelerating Iterative Machine Learning. *PVLDB* 12, 4 (2018), 446–460. https://doi.org/10.14778/3297753.3297763

[126] Luna Xu, Min Li, Li Zhang, Ali Raza Butt, Yandong Wang, and Zane Zhenhua Hu. 2016. MEMTUNE: Dynamic Memory Management for In-Memory Data Analytic Platforms. In *IPDPS*. 383–392. https://doi.org/10.1109/IPDPS.2016.105

[127] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. 2017. LRC: Dependency-aware cache management for data analytics clusters. In *INFOCOM*. 1–9. https://doi.org/10.1109/INFOCOM.2017.8057007

[128] Yinghao Yu, Chengliang Zhang, Wei Wang, Jun Zhang, and Khaled Ben Letaief. 2022. Towards Dependency-Aware Cache Management for Data Analytics Applications. *IEEE Trans. Cloud Comput.* 10, 1 (2022), 706–723. https://doi.org/10.1109/TCC.2019.2945015

[129] Matei Zaharia et al. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*. 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[130] Ce Zhang, Arun Kumar, and Christopher Ré. 2014. Materialization optimizations for feature selection workloads. In *SIGMOD*. 265–276. https://doi.org/10.1145/2588555.2593678

[131] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An empirical study on program failures of deep learning jobs. In *ICSE*. 1159–1170. https://doi.org/10.1145/3377811.3380362

[132] Yanli Zhao et al. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *PVLDB* 16, 12 (2023), 3848–3860. https://doi.org/10.14778/3611540.3611569

[133] Bojian Zheng, Nandita Vijaykumar, and Gennady Pekhimenko. 2020. Echo: Compiler-based GPU Memory Footprint Reduction for LSTM RNN Training. In *ISCA*. 1089–1102. https://doi.org/10.1109/ISCA45697.2020.00092

[134] Lianmin Zheng et al. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *OSDI*. 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

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