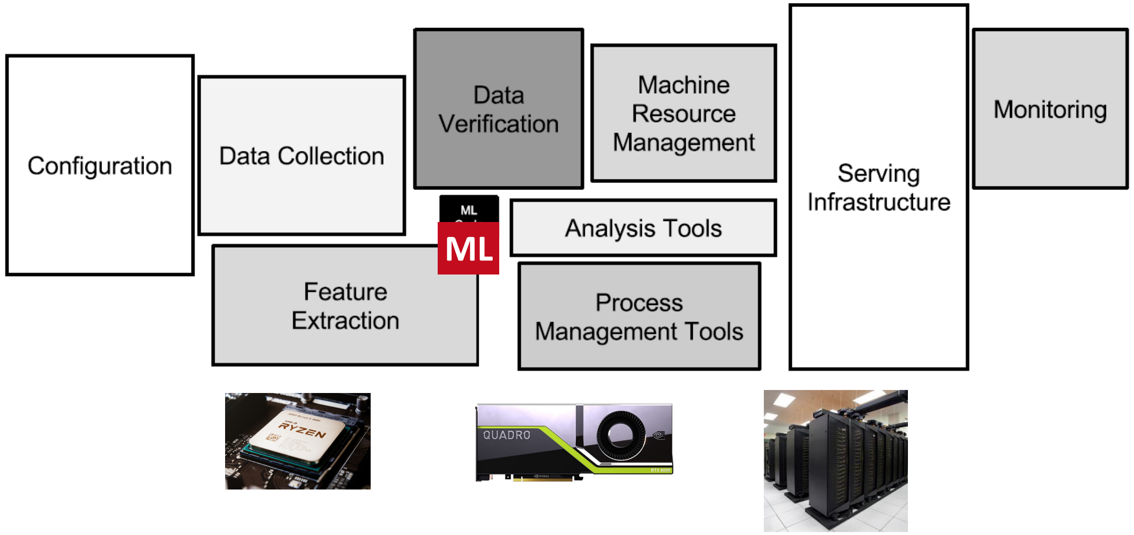
BIFOLD

TECHNISCHE UNIVERSITÄT BERLIN

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

Arnab Phani, Matthias Boehm
TU Berlin

Exploratory Data Science



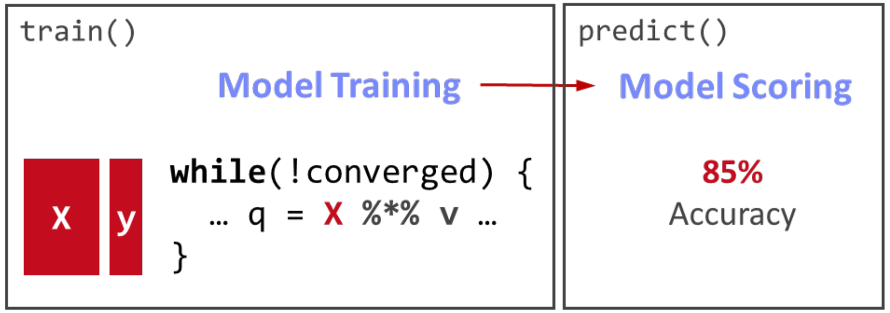
Data Integration & Data Cleaning

Data Programming & Augmentation

Model and Feature Selection

Hyper-parameter Tuning + CV

Hierarchical Composition as Library Functions on top of ML systems



Validation & Debugging

Deployment & Scoring

- **Problem**
  - Data science workflows are exploratory, **hierarchically composed** and complex
  - Hierarchy of ML tasks create high **computational redundancy** across ML tasks
  - Redundancy in various levels, from redundant function calls to LA operators
  - **Multiple backends** to server diverse data and applications.

Existing Work on Reuse

- **Coarse-grained Reuse**
  - **Black-box** view of ML algorithms (hidden sub-steps)
  - **Coarse-grained** reuse eliminates top-level redundancy
  - Cannot eliminate **fine-grained redundancy** (LA Ops)

- **Backend/Workload-specific Reuse**
  - Reuse tailored to specific workloads and backends

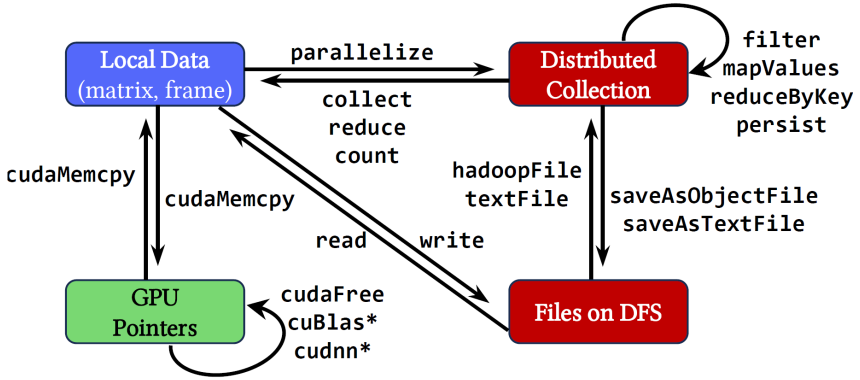
Framework	Reuse	Multi-backend	Memory Mgmt.	Workload
HELIX, CO, HYPPO	Coarse	No	No	ML Pipelines
Clipper, PRETZEL	Coarse	No	No	Inference
MEMTUNE, MRD	Fine	Reuse RDDs	Spark Storage	Spark Jobs
TensorFlow, PyTorch	No	Recycle GPU Ptrs.	GPU Memory	DNNs
Capuchin	No	Activations (GPU)	No	DNNs
Cachew	Coarse	Distributed Reuse	No	Preprocessing
VISTA, SHiFT	Coarse	Reuse DNN Layers	No	Transfer Learning
LIMA	Fine	No	No	All
MEMPHIS	**Fine**	**RDDs, GPU Ptrs.**	**Yes**	**All**

Specific focus limits applicability for hybrid workloads and diverse backend combinations.

MEMPHIS

Reuse in Multi-backend ML Systems

- **Hybrid Execution Plan of Multi-backend Operations**
- **Challenges**
  - Backends differs in execution model, memory bandwidth, memory management, target workloads
  - Reuse linked with Op scheduling and data exchange
- **MEMPHIS**
  - **Holistic framework** for multi-backend reuse of intermediates
  - A **unified cache abstraction** with system-internal API
  - Reuse Spark actions, **RDDs, GPU pointers**
  - Specialized cache management
  - Unified **memory management**
  - **Holistic integration** into compiler and runtime



Lifecycle of data objects

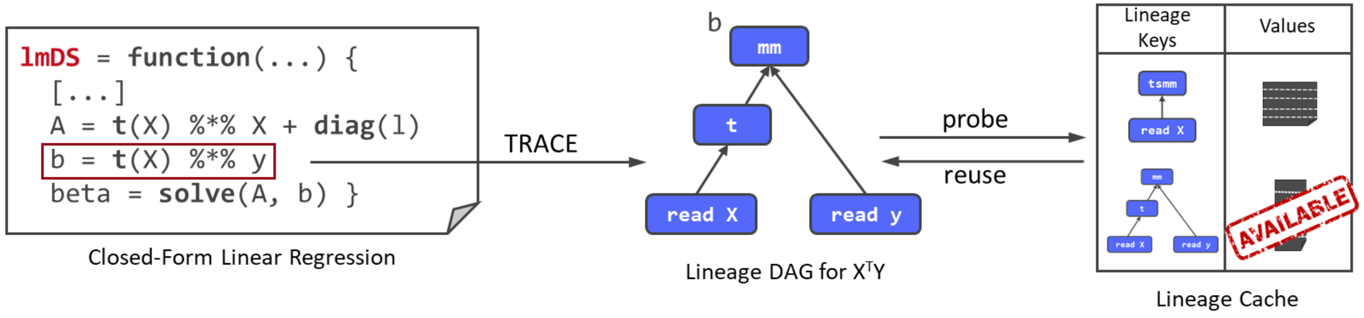
	Exec.	Memory	Cache-API	Workload
Spark	Lazy	Distrib.	Yes	Large data
GPU	Async.	Small	No	Mini-batch, DNN
CPU	Eager	Varying	No	All

Properties of Spark, GPU, CPU

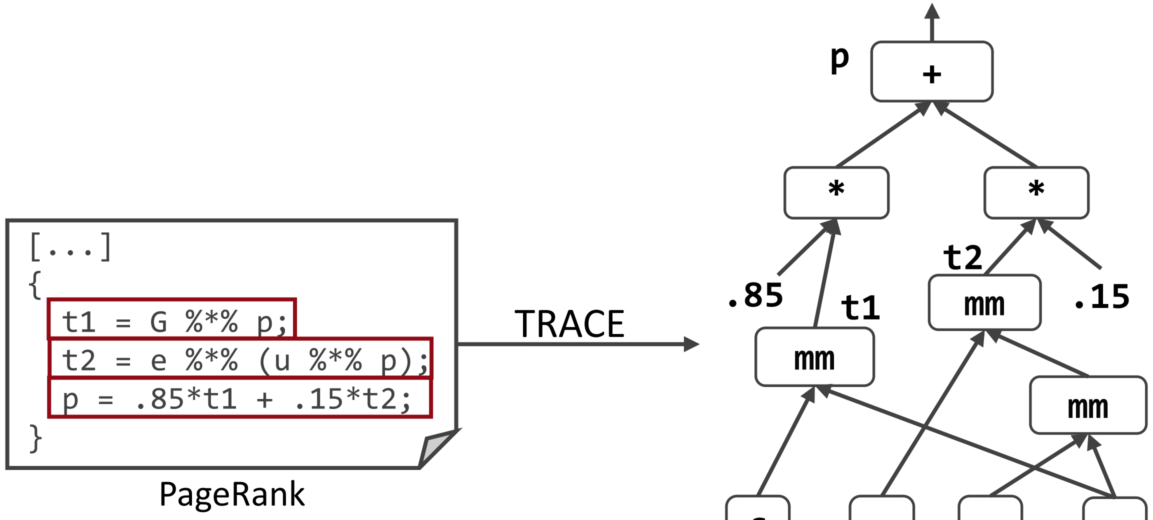
THE APACHE SOFTWARE FOUNDATION

Apache SystemDS ™

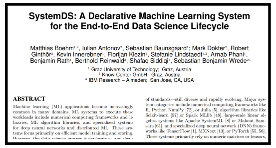
Background: Lineage Tracing and Reuse



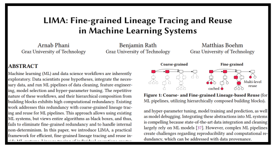
LIMA: Lineage Tracing



CIDR 2020



SIGMOD 2021

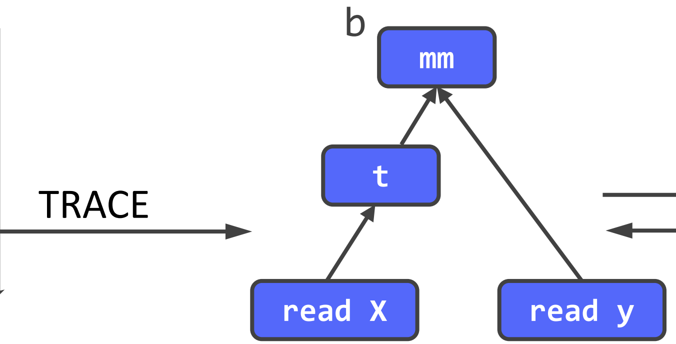


- **Lineage DAG**
  - Trace lineage and use lineage traces for reuse, debugging and recomputation
  - During runtime of an LA program, we maintain **lineage DAGs** for all live variables
  - Incrementally built as we execute instructions

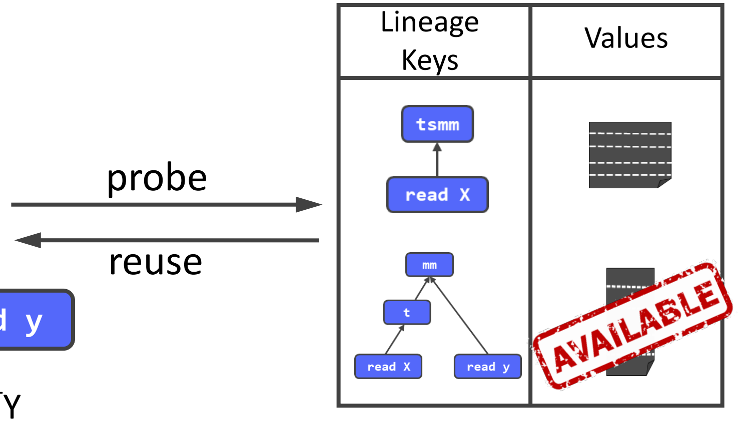
LIMA: Lineage-based Reuse

```
lmDS = function(...) {
  [...]
  A = t(X) %*% X + diag(l)
  b = t(X) %*% y
  beta = solve(A, b) }
```

Closed-Form Linear Regression



Lineage DAG for X^TY



Lineage Cache

- **Full Reuse**
  - Lineage of an intermediate uniquely identifies the intermediate
  - Lineage cache comprises a hash map, **`Map<Lineage, Intermediate>`**
  - Before executing instruction, **probe lineage cache** for outputs
  - Reuse of function calls and code blocks

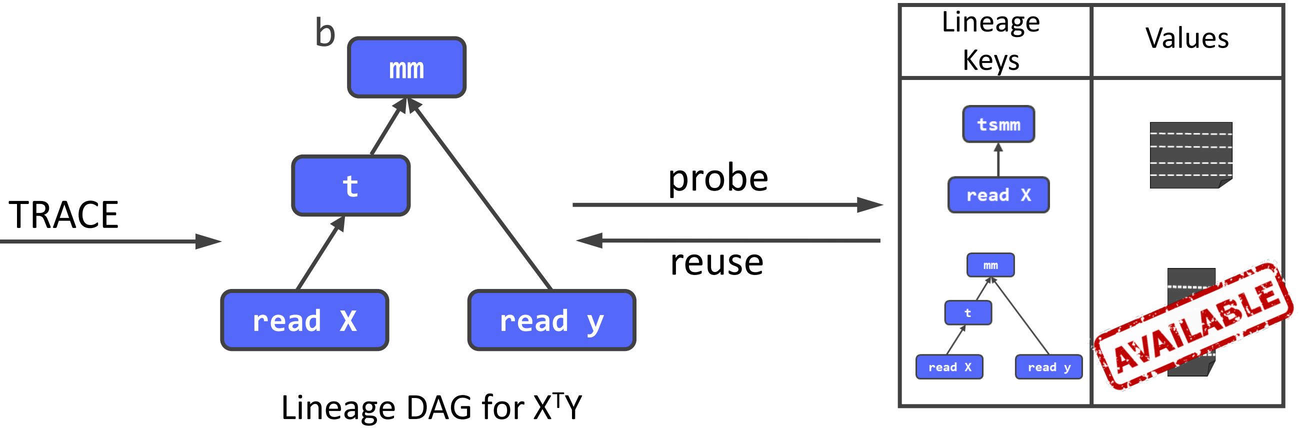
```
for (inst in insts)
  lt = TRACE (inst)
  if (!REUSE (lt))
    out = exec(inst)
    PUT (lt, out)
```

REUSE API Integration

LIMA: Lineage-based Reuse

```
lmDS = function(...) {
  [...]
  A = t(X) %*% X + diag(l)
  b = t(X) %*% y
  beta = solve(A, b) }
```

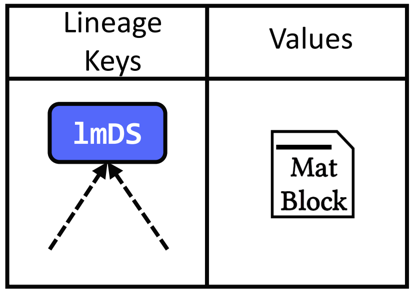
Closed-Form Linear Regression



Lineage DAG for X^TY

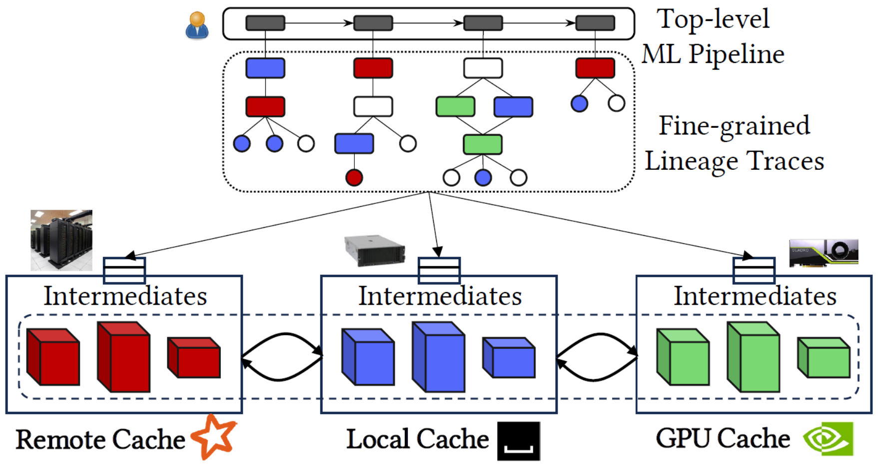
Lineage Cache

- **Full Reuse**
  - Lineage of an intermediate uniquely identifies the intermediate
  - Lineage cache comprises a hash map, **`Map<Lineage, Intermediate>`**
  - Before executing instruction, **probe lineage cache** for outputs
  - Reuse of function calls and code blocks



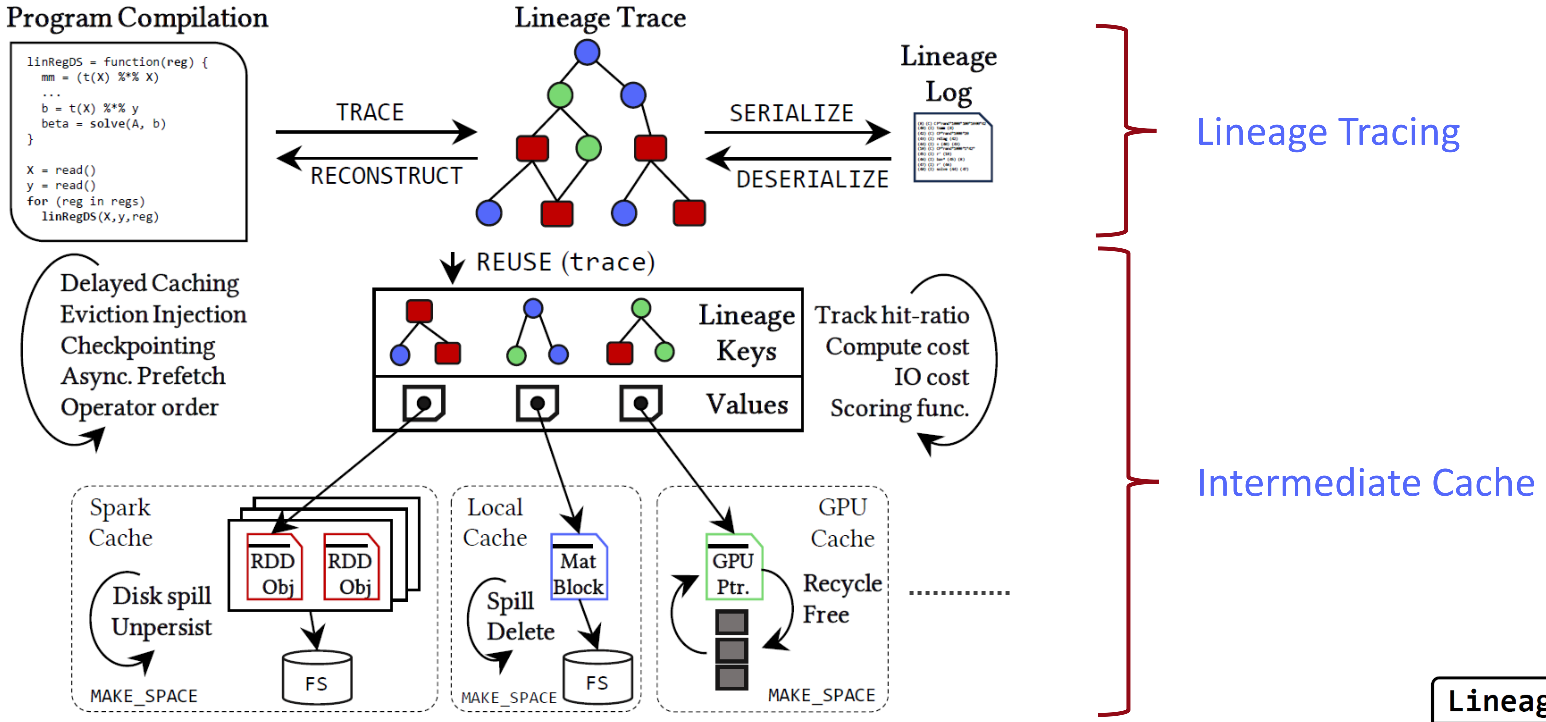
Reuse Function Outputs

Hierarchical Lineage Cache and Multi-backend Reuse



MEMPHIS

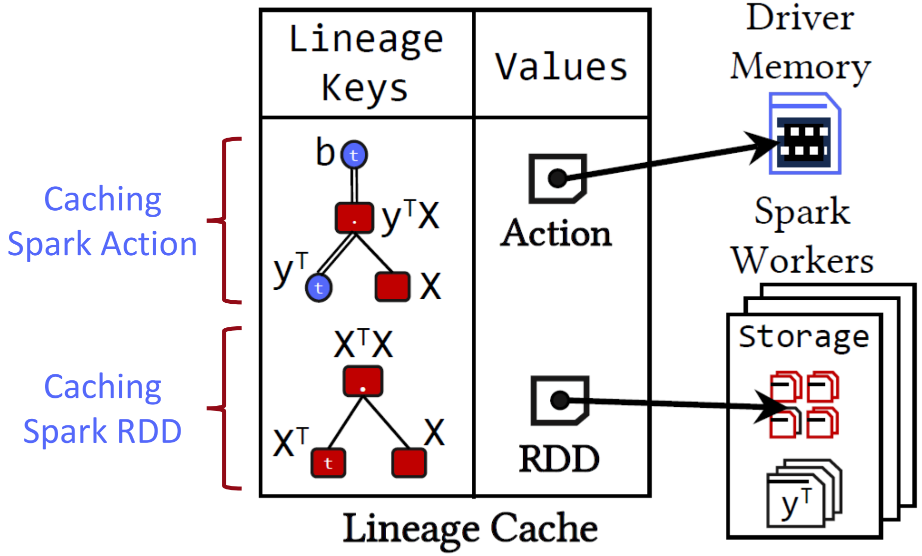
Hierarchical Lineage Cache



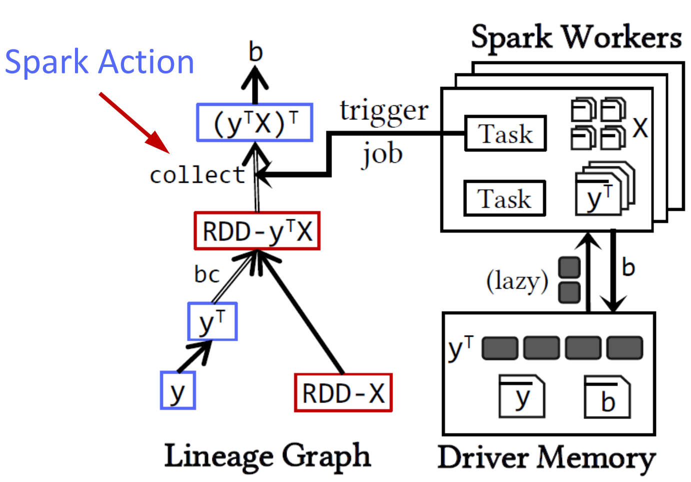
- **Overview**
  - Driver/host handles lineage tracing, reuse-aware compilation, eviction planning
  - Data objects reside in the backends
  - Lineage DAGs are **backend-agnostic**
  - **Hierarchical design** seamlessly accommodates **heterogeneous backends**

```
LineageCacheEntry
-MBval: MatrixBlock
-SOval: ScalarObject
-RDDObj: RDDObject
-GPUPtr: GPUPointer
-status: CacheStatus
-computeTime: long
-score: double
+getMBValue()
+getCacheStatus()
```

Reuse & Memory Management in Spark



Spark Action and RDD Reuse



Lazy Evaluation

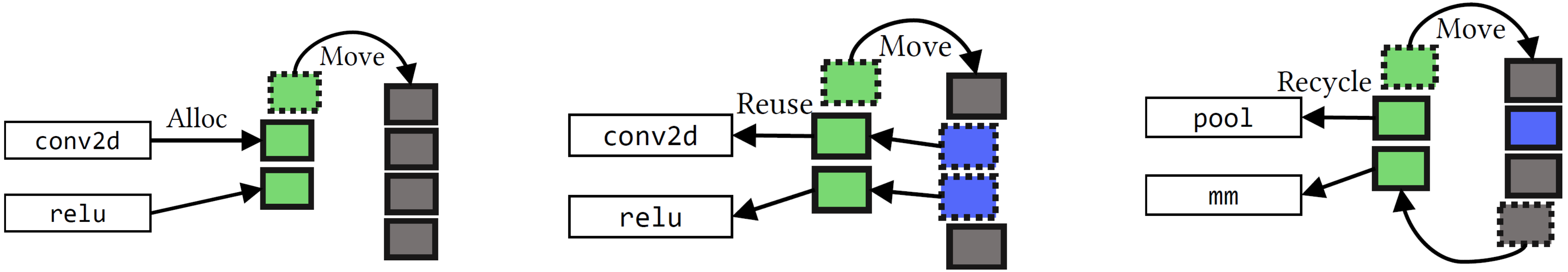
- **Reuse Spark Actions & RDDs**
  - Eager caching triggers jobs after instruction execution
  - Cache actions in driver, RDDs in cluster
  - Lazy materialization introduces **memory overhead**
  - Lazy garbage collection, asynchronous materialization

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

Eager caching

MEMPHIS

Reuse & Memory Management in GPU

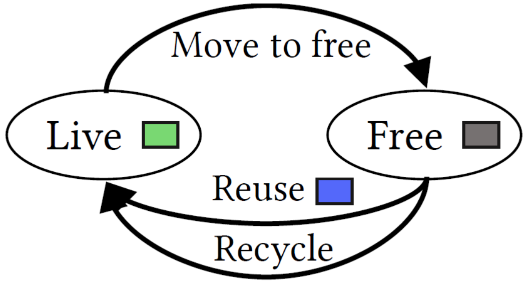


Allocate Pointers

Reuse Pointers

Recycle Pointers

- **Unified Memory Manager**
  - Small memory and data copy challenges
  - Pool allocator: **Live** and **Free** lists
  - **Reuse** GPU pointers
  - Once full, start recycling
  - Helps **mini-batch DNN,** avoid **alloc/dealloc overhead**



GPU Pointer Lifecycle

Cost-based Cache Eviction

Spark: $\underset{o \in Q}{\arg\min} \; (r_h + r_m + r_j) \cdot c(o)/s(o)$

#hits, #misses, #jobs

Compute cost

Estimated size

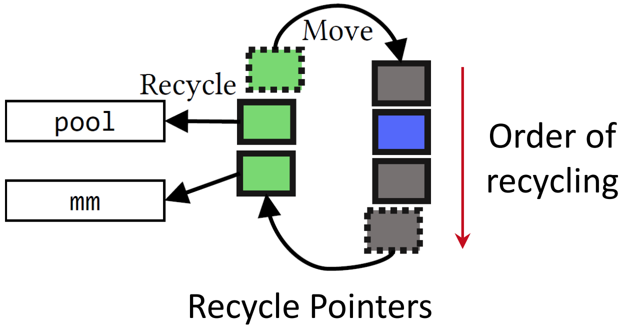
GPU: $\underset{o \in Q}{\arg\min} \; (T_a(o) + 1/h(o) + c(o))$

Last access timestamp

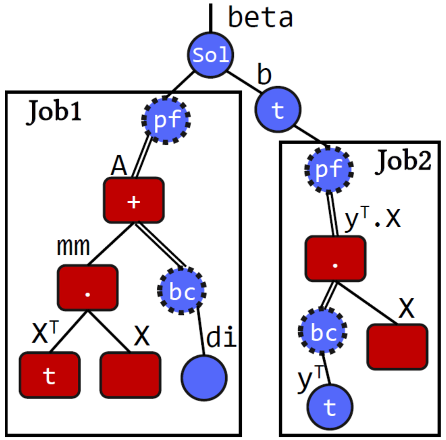
Lineage DAG height

Compute cost

- **Backend-specific Cache Eviction**
  - Preserve objects with **high reuse benefits**
  - Scoring functions determine the **order of eviction**
  - Statistics collection during execution
  - CPU, Spark: Cost&Size
  - GPU: Serve mini-batch workloads



Compiler Integration



Async. Data Exchange

```
# Feature selection
for (i in 1:ncol(X)) {
1   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)
2   model = TrainPipe(λ)
                                n = 1
# Clean and train
TrainPipe = function(λ) {
3   X_cl1 = imputeMV(X_b)
    X_ol2 = outlrIQR(X_cl1)
                                n = 1
    while(minimize)
4     ..training loop..
                                n = 2
}
```

Delay Factor Tuning

```
for (i in 1:iters) {     //AlexNet
  batch = data[beg:end]
  c1 = conv2d(..,11,11,4,4)
  r1 = relu(c1)
  ⋮
  probs = softmax(..)
}                                      Cannot
        gpu_evict(100)                 recycle
for (i in 1:iters) {     //VGG16
  batch = data[beg:end]
  c1 = conv2d(..,3,3,1,1)
  r1 = relu(c1)
  ⋮
  probs = softmax(..)
}
```

Eviction Injection

- **Compiler Optimizations**
  - Asynchronous OPs: **Prefetch**, Broadcast
  - **Delayed Caching**: cache after *n* hits
  - **Eviction Injection**: handles allocation shift
  - Operator Ordering: Maximize inter-backend parallelism

Experiments

Compiler Integration

- **Baselines**

LIMA	Fine-grained reuse
HELIX	Coarse-grained reuse
CoorDL	Input data pipeline reuse in CPU
Clipper	Prediction reuse
VISTA	Reuse in transfer learning

Apache SystemDS™

PyTorch

- **Datasets and Workloads**

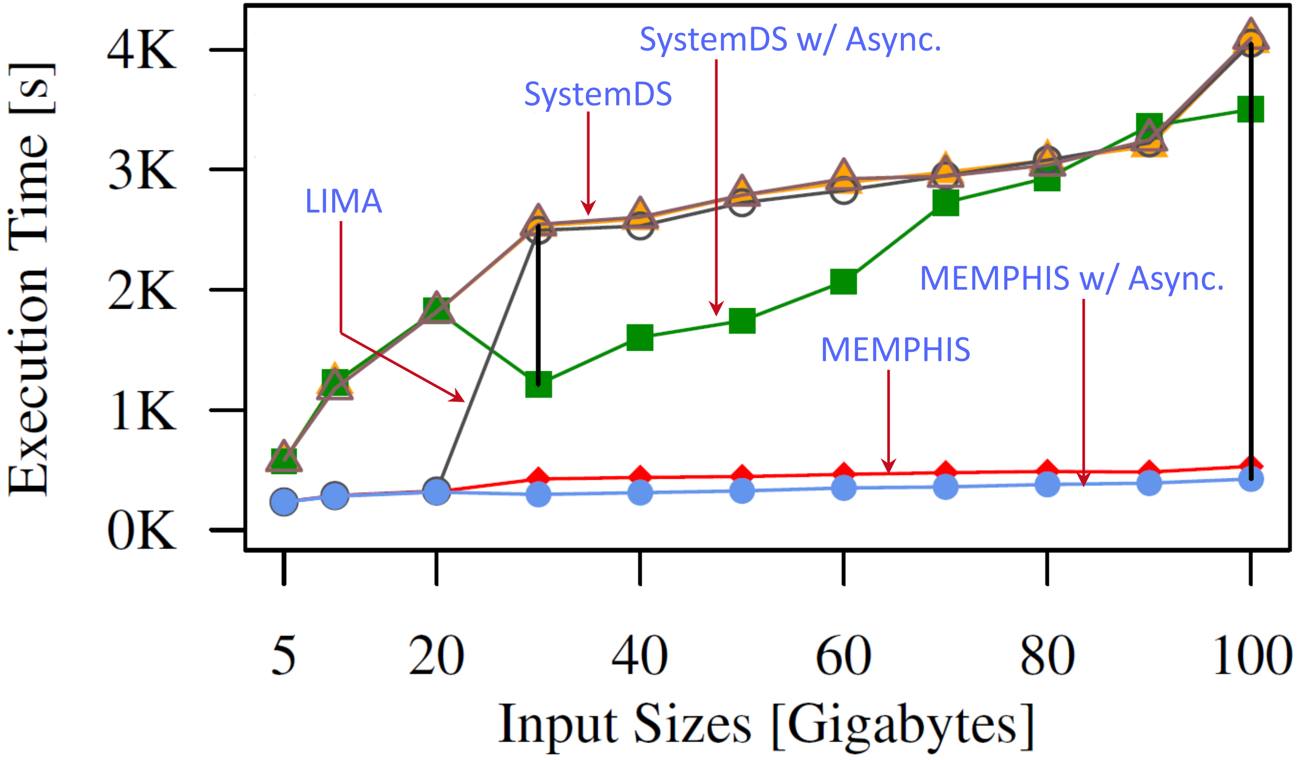
Name	Use Case	Dataset	Influential Techniques
HCV	Grid Search / Cross Validation	Synthetic	Async. OPs, local & RDD reuse
PNMF	Non-negative Matrix Factorization	MovieLens	Checkpoint placement
HBAND	Hyperband Model Selection	Synthetic	Multi-level reuse, delayed caching
CLEAN	Data Cleaning Pipelines	APS	Large #intermediates & #evictions
HDROP	Dropout Rate Tuning	KDD 98	Local and GPU ptr. reuse
EN2DE	Machine Translation Inference	WMT14	Recycle & reuse GPU ptrs.
TLVIS	Transfer Learning Feature Extraction	ImageNet, CIFAR-10	Evictions & mem. management

MEMPHIS

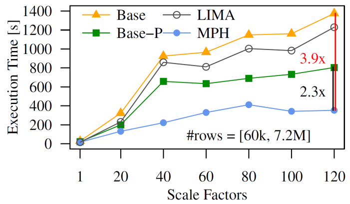
Experiments

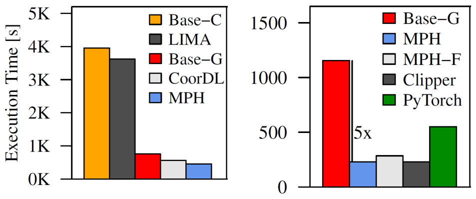
Seamless reuse **across backends and workloads**

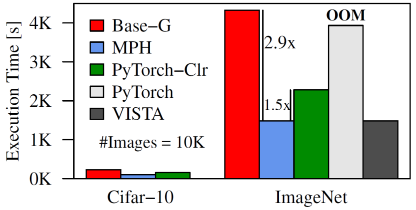
Grid Search LM / Cross Validation (w/ & w/o Prefetch)

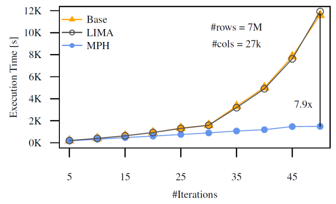


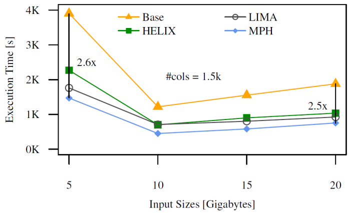
Reused Intermediates: RDDs, Spark Actions, Local Data Objects







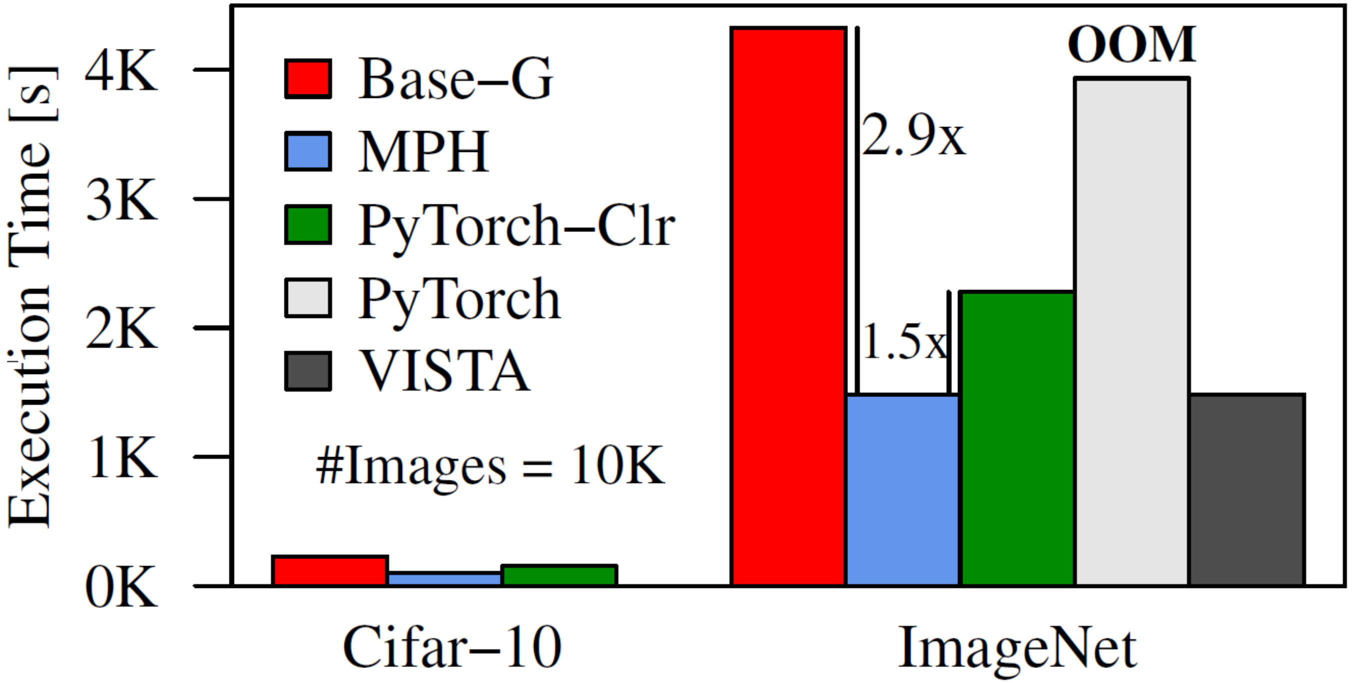




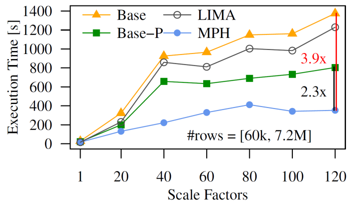
Experiments

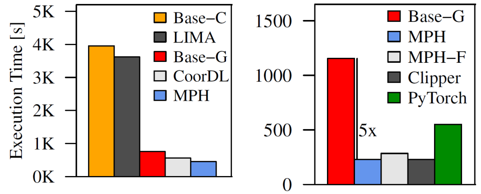
Seamless reuse **across backends and workloads**

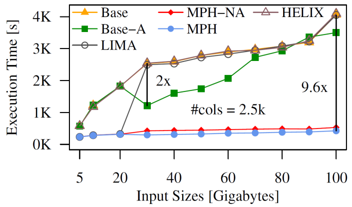
Feature Extraction for Transfer Learning

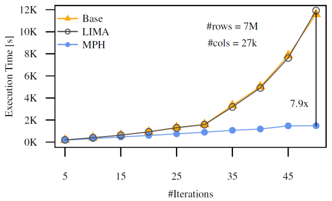


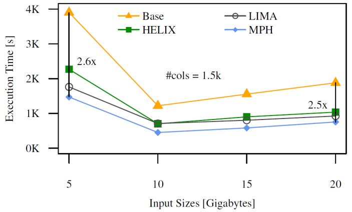
30K Reused Pointers and 17.5K Recycled Pointers











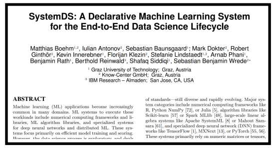
Conclusions

- Redundancy is inevitable in modern data-centric ML pipelines
- Diversity of backends, applications makes simple reuse harder
- Fine-grained view towards ML tasks enable reuse and parallelization
- A robust integration of a reuse framework in ML systems

SystemDS Repo

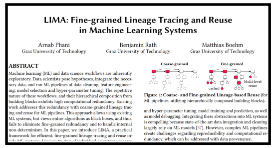
Reproducibility Repo

CIDR 2020



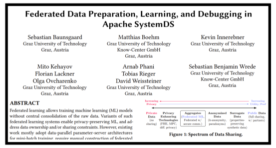
Vision of SystemDS and reuse of intermediates

SIGMOD 2021



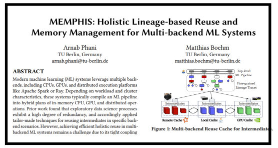
LIMA: Lineage tracing and lineage-based reuse and

CIKM 2022



Multi-tenant reuse in federated settings

EDBT 2025



MEMPHIS: Multi-backend reuse and memory management

Apache SystemDS™