Architecting Intermediate Layers for Efficient Composition in End-to-End Data Science Pipelines

Supun Abeysinghe

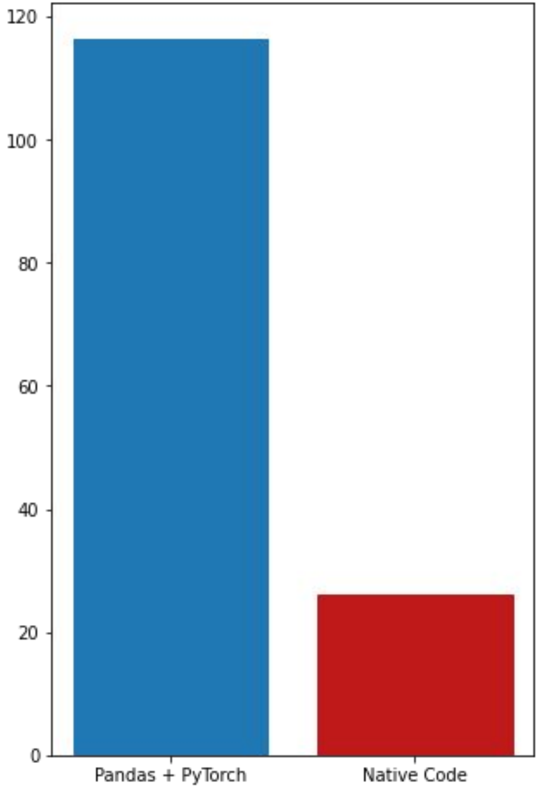
Motivation

Typical end-to-end data science pipelines are created using multiple libraries/frameworks/systems

e.g. Pandas + PyTorch

Motivation

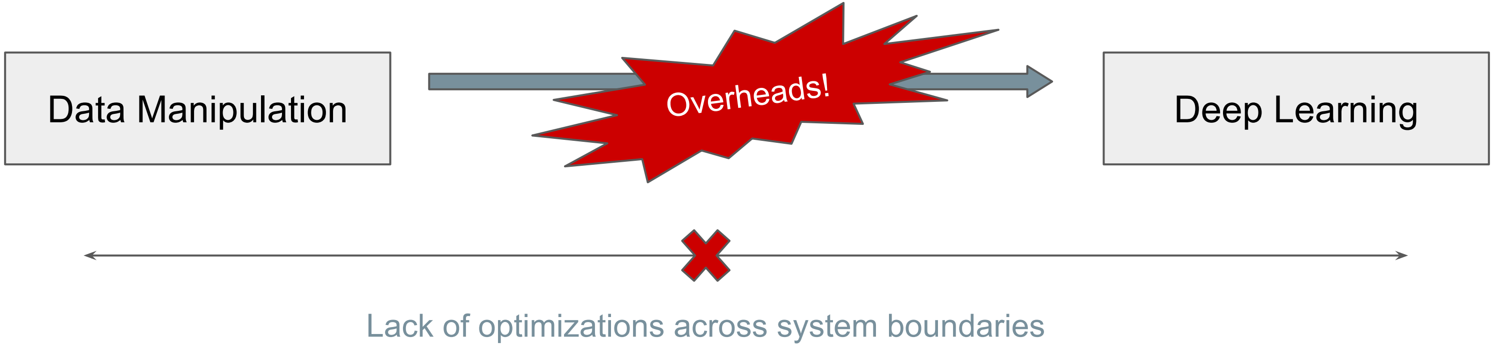
These systems go to great lengths to optimize the performance of their specific workloads, however, end-to-end performance is suboptimal



End-to-end performance for a ETL + Deep Learning Workload

Motivation

1. Data copying/transformation at system boundaries
2. Lack of global optimization



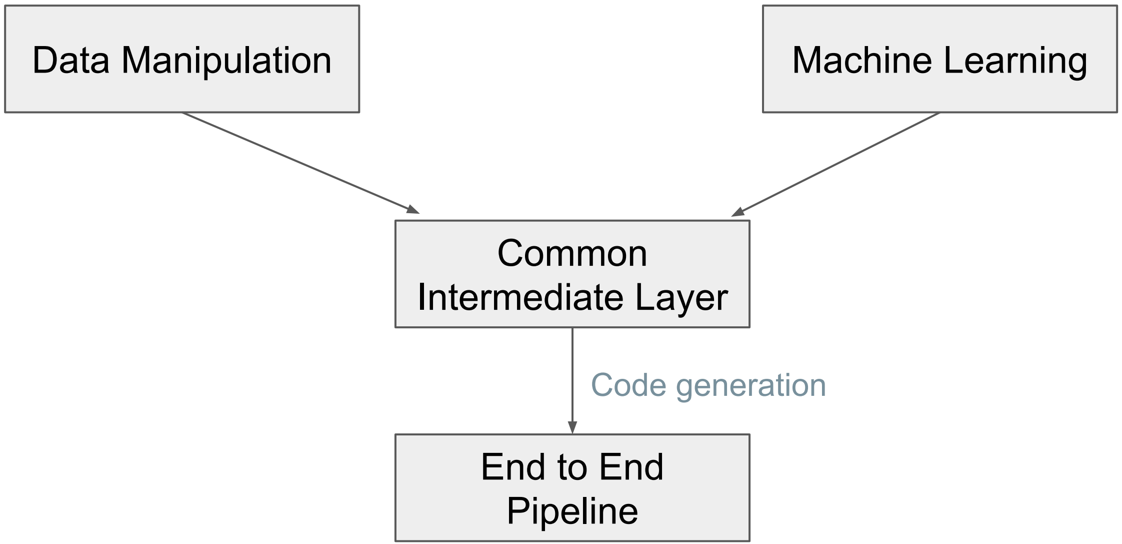
Problem

How can we build efficient end-to-end data science pipelines that

1. avoid performance overheads at subsystem boundaries
2. do not preclude global optimizations
3. retain familiar high-level interfaces to the end user

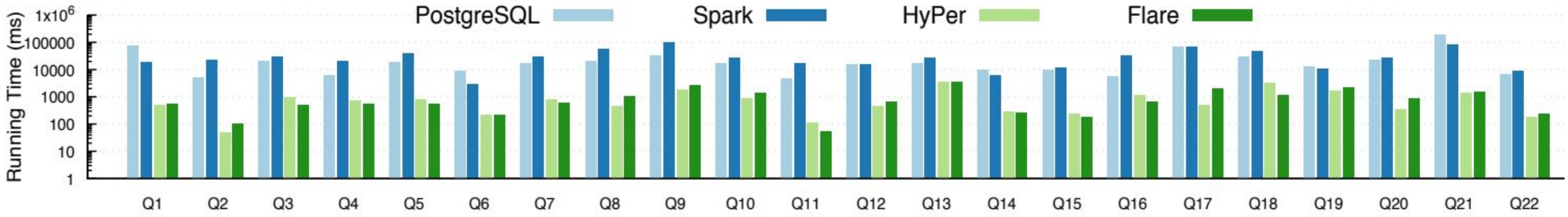
Intermediate Layers

A key idea to remove these bottlenecks is to reorganize existing frameworks around a common intermediate layer



Flare

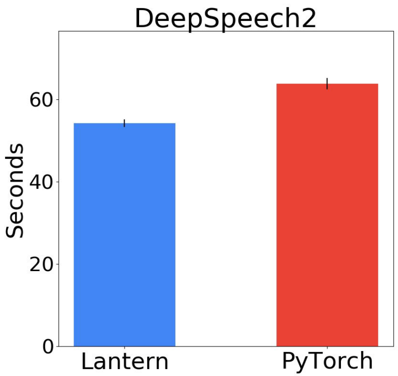
- Accelerator for Apache Spark
- Generates native code for optimized query plans from Apache Spark
- Leverages Lightweight Modular Staging (**LMS**)

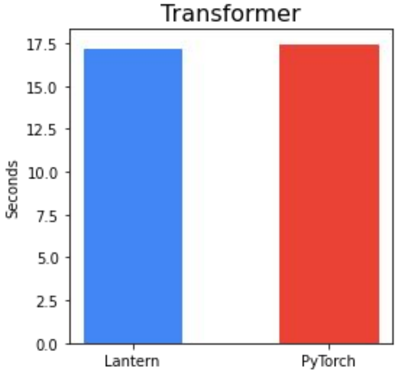


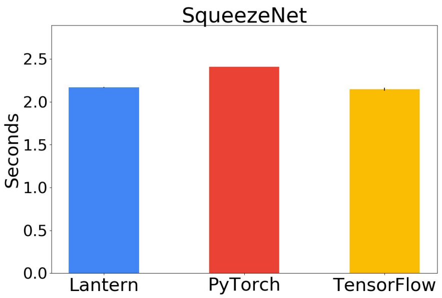
TPC-H Performance Benchmark

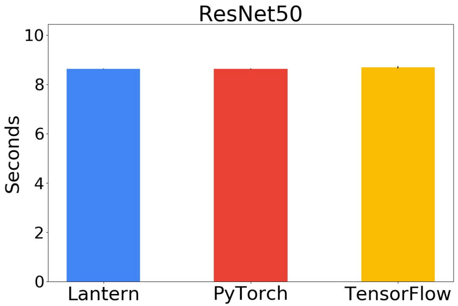
Lantern

- Deep Learning framework
- Performance of “define-then-run”
- Expressiveness of “define-by-run”
- Leverages Lightweight Modular Staging (**LMS**) to generate low-level (C/CUDA) code









Flern

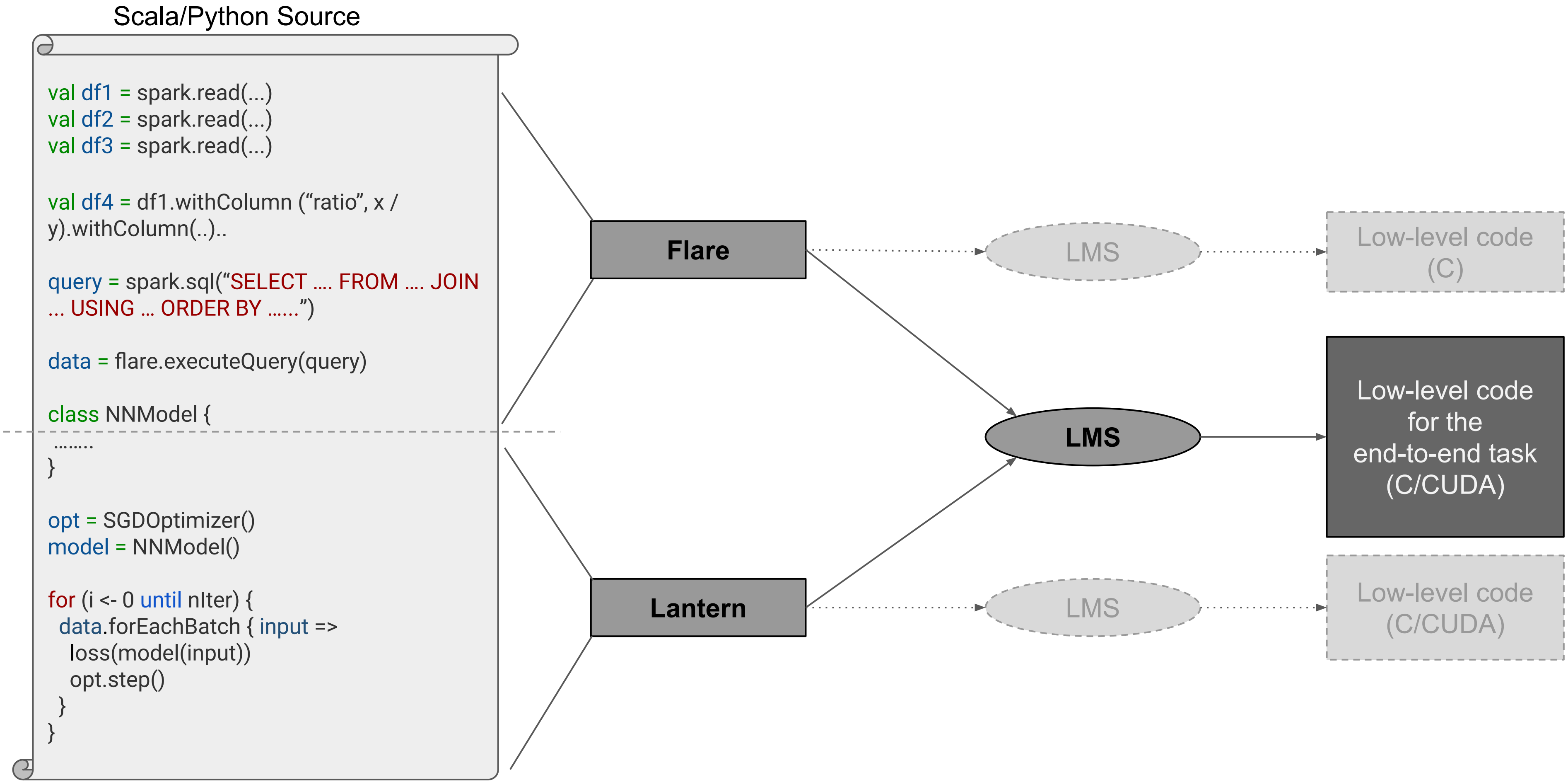
First intermediate-layer integration between systems that achieves,

1. best-of-breed performance individually

   competitive with best compiled query engines in **full relational benchmarks** (TPC-H)

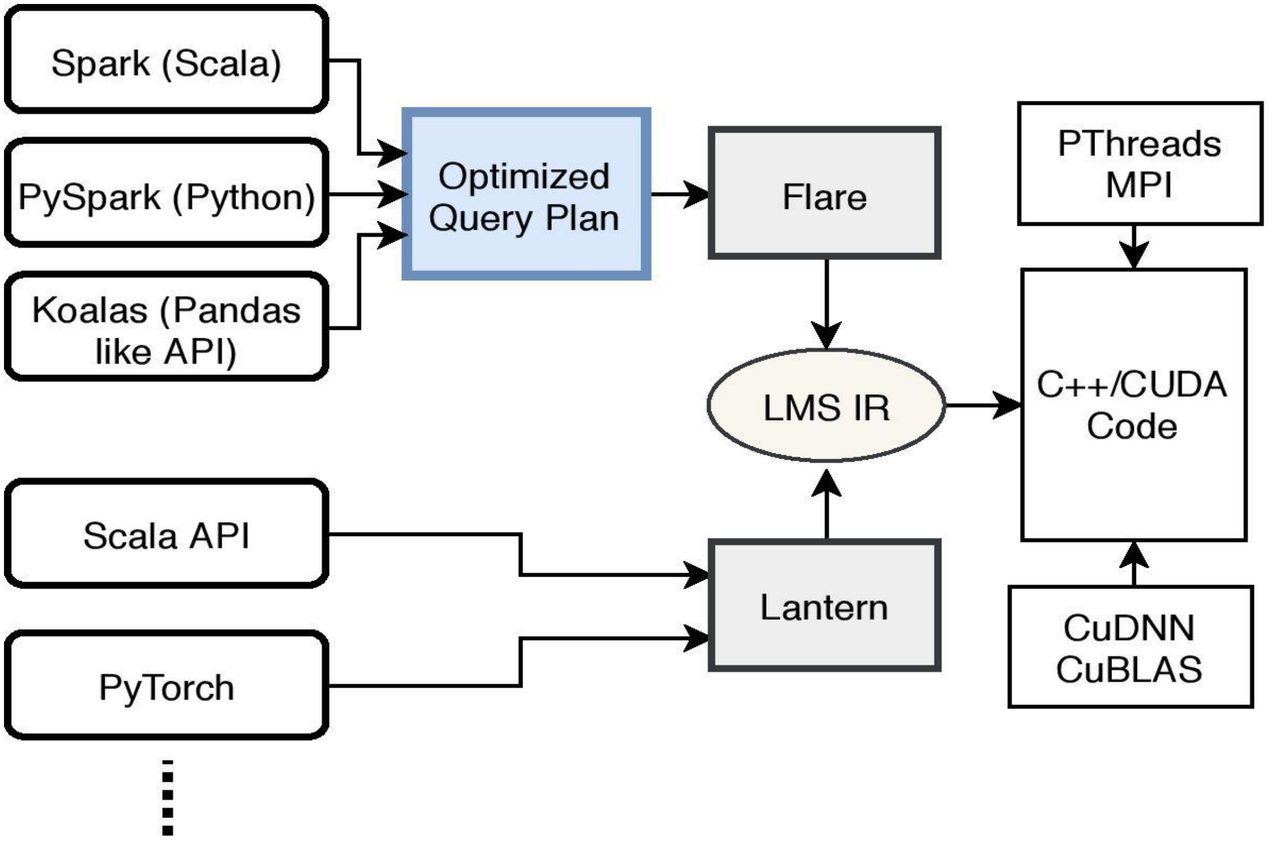
   competitive with Tensorflow, PyTorch in state-of-the-art machine learning models (**Transformer**, **DeepSpeech**)

2. competitive performance on end-to-end workloads



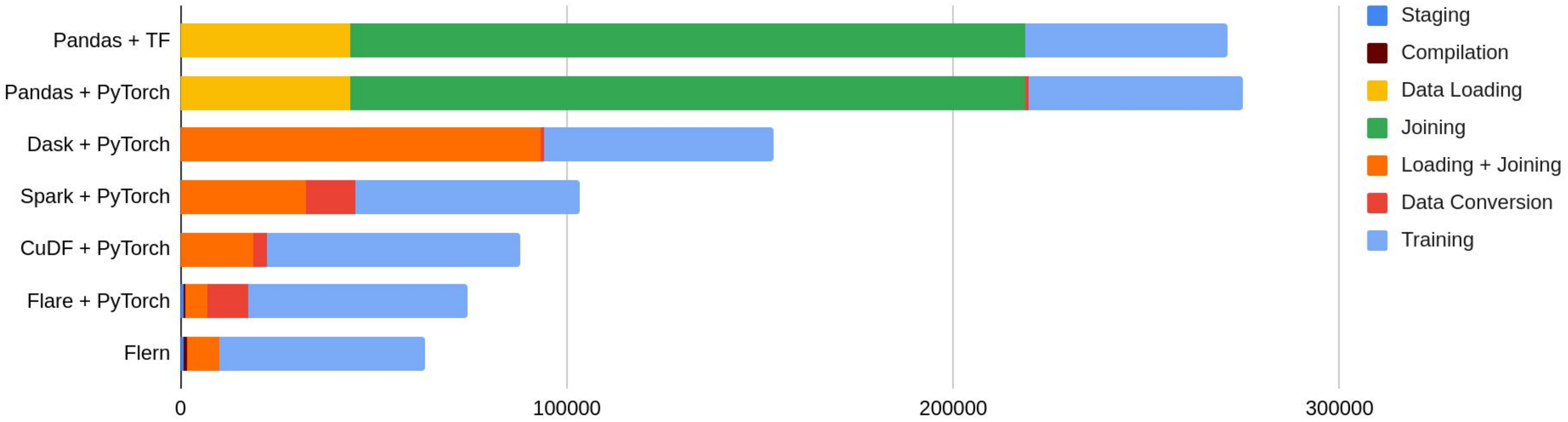
Scala/Python Source
val df1 = spark.read(...)
val df2 = spark.read(...)
val df3 = spark.read(...)
val df4 = df1.withColumn ("ratio", x / y).withColumn(..)..
query = spark.sql("SELECT .... FROM .... JOIN ... USING ... ORDER BY ......")
data = flare.executeQuery(query)
class NNModel {
....
}
opt = SGDOptimizer()
model = NNModel()
for (i <- 0 until nIter) {
data.forEachBatch { input =>
loss(model(input))
opt.step()
}
}
Flare
Lantern
LMS
LMS
LMS
Low-level code (C)
Low-level code for the end-to-end task (C/CUDA)
Low-level code (C/CUDA)

Overall Architecture



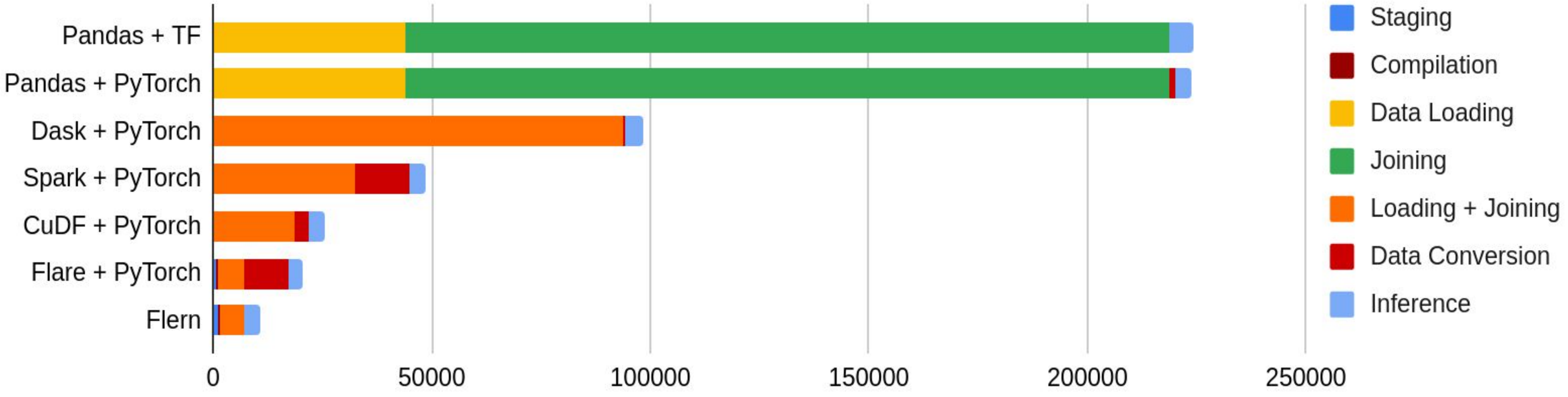
Performance - End-to-end Training

4x speedup!



Performance - End-to-end Inference

20x speedup!



Future

Beyond single node; a lot of interesting challenges at the integration point!

Pipeline parallelism across end-to-end workloads

End to end GPU execution (+ generating specialized cuda kernels)

Traditional machine learning models (large scale logistic regression, gradient boosted trees, etc.)