Flare & Lantern: Efficiently Swapping Horses Midstream

Grégory Essertel, Ruby Y. Tahboub, Fei Wang, James Decker, Tiark Rompf
Purdue University, West Lafayette, Indiana
{gesserte,rtahboub,wang603, decker31, tiark}@purdue.edu

ABSTRACT

Running machine learning (ML) workloads at scale is as much a data management problem as a model engineering problem. Big performance challenges exist when data management systems invoke ML classifiers as user-defined functions (UDFs) or when stand-alone ML frameworks interact with data stores for data loading and pre-processing (ETL). In particular, UDFs can be precompiled or simply a black box for the data management system and the data layout may be completely different from the native layout, thus adding overheads at the boundaries. In this demo, we will show how bottlenecks between existing systems can be eliminated when their engines are designed around runtime compilation and native code generation, which is the case for many state-of-the-art relational engines as well as ML frameworks. We demonstrate an integration of Flare (an accelerator for Spark SQL), and Lantern (an accelerator for TensorFlow and PyTorch) that results in a highly optimized end-to-end compiled data path, switching between SQL and ML processing with negligible overhead.

PVLDB Reference Format:
Grégory Essertel, Ruby Y. Tahboub, Fei Wang and Tiark Rompf. Flare & Lantern: Efficiently Swapping Horses Midstream. *PVLDB*, 12(12): 1910-1913, 2019.
DOI: https://doi.org/10.14778/3352063.3352097

1. INTRODUCTION

Machine learning, and especially deep neural networks, have been extraordinarily successful in fields such as game play, image recognition, speech processing, etc., and are having a similar impact on business intelligence and related domains. Productionizing ML applications and deploying them at scale often requires interfacing with big datasets stored in data management systems (DBMS) or data stores such as HDFS.

Interactions with ML systems include sampling, extracting, and pre-processing data from DB systems, and certain DB queries also use ML trained models as UDFs (User-Defined Functions), e.g in Figure 1.

```
for epoch in range(100):
  for batch, labels in sql("select ... from t1 join t2 ..."):
     y = model.train(batch, labels)

sql.register_udf("model", model)
sql("select * from input where model(x, y, z, ...) == Cluster1")
```

Figure 1: DBMS used for data loading with ML training algorithm and model used as UDF within a SQL query.

Many of these query execution pipelines mix data manipulation and domain-specific operations. For examples, given a dataset of images, a query pipeline may need to filter, join and mirror images.

The current state of affairs in ML processing represents a chasm between two classes of systems. Data management systems are highly optimized on the relational side but typically lacking in native ML support, especially regarding automatic differentiation, which is necessary for training via gradient descent. As a result, ML libraries are integrated as external user-defined functions UDFs e.g., a PyTorch or TensorFlow [1] classifier in Spark [2], Flare [5], etc. (See Figure 2). The performance of ML UDFs varies based on the nature of integration with DBMS evaluation (i.e., black box entirely or with some degree of cross-optimization).

On the other hand, ML frameworks e.g., TensorFlow, PyTorch, etc. provide high-level front-ends on the top of highly-optimized back-end kernels. Programming on such frameworks is heavily based on API calls. However ML frameworks are not optimized for data processing when data is local or when accessing distributed datasets.

Comparison with Recent Hybrid Compiler Frameworks. Earlier works that integrated data manipulation and ML processing into a common intermediate representation IR are Delite [11] and Weld [9]. Delite's Distributed Multiloop Language (DMLL) [3] and Weld's common runtime perform cross optimization on data processing and ML operations. While both systems achieve efficient ML processing on the hybrid scenario, the performance of individual cases (i.e., ML or data processing alone) does not match the performance of the best of breed systems yet. Weld's evaluations have only shown limited relational queries and small ML models. Delite does not yet support neural network construction and automated differentiation, which are the keys to modern deep learning. On the other hand, this work demonstrates the best of breed performance for both state of

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352097

```
# Define linear classifier using TensorFlow
import tensorflow as tf
# weights from pre-trained model elided
mat = tf.constant([[...]])
bias = tf.constant([...])
def classifier(c1,c2,c3,c4):
  # compute distance
  x = tf.constant([[c1,c2,c3,c4]])
  y = tf.matmul(x, mat) + bias
  y1 = tf.session.run(y1)[0]
  return max(y1)
# Register classifier as UDF: dumps TensorFlow graph to
# a .pbtxt file, runs tf_compile to obtain .o binary file
flare.udf.register_tfcompile("classifier", classifier)
# Use compiled classifer in PySpark query with Flare:
q = spark.sql("
  select real_class,
    sum(case when class = 0 then 1 else 0 end) as class1,
    sum(case when class = 1 then 1 else 0 end) as class2,
    ... until 4 ...
  from (select real_class,
               classifier(c1,c2,c3,c4) as class from data)
  group by real_class order by real_class")
flare(q).show()
```

Figure 2: Spark query using TensorFlow classifier as a UDF in Python.

the art deep learning models e.g., DeepSpeech or SqueezeNet and on a full relational benchmark e.g., TPC-H.

2. SYSTEM OVERVIEW

2.1 Flare

Flare [5] is a back-end accelerator for Apache Spark SQL that displays high performance that matches the state of the art in in-memory database systems. Flare has been designed around the data-centric model introduced in Hyper [8] and implements the code generation techniques that we developed for the LB2 system [12].

Flare is architected to bypass the default Spark back-end that is inadequate for single NUMA machine. Indeed, Spark is using resilient distributed datasets (RDDs) for its internal execution. While these RDDs are giving a good abstraction for map reduce operation, allowing them to be parallelized transparently and also are ensuring fault-tolerant computation, they come at a high price in term of performance when the system is used on in-memory data within a single machine. Figure 4 shows the performance benchmark of Flare, a traditional DBMS system (Postgres), Spark using its native RDD back-end layer and Hyper, the state of the art in in-memory data management system. The benchmark used is TPC-H with scale factor 10. We can see that Postgres and Spark are much slower than the two other systems and that Flare and Hyper are neck to neck in most of the queries.

Flare generates low-level C code using the LMS framework. Within Flare, LMS is used to partially evaluate the query interpreter with respect to the query plan given as input. This process, known as the first Futamura projection [6], generate a compiled version of the input. We provide more insight about LMS in Section 2.3

2.2 Lantern

Lantern [14, 16, 15] is a machine learning framework that performs automated differentiation via delimited continuations, and uses LMS to generate efficient low-level C++ and CUDA code

Machine learning relies on backpropagation to compute gradients and update model parameters. Traditional machine learning frameworks such as TensorFlow and PyTorch use axillary data structures (computation graphs or traces) to track forward computations for backpropagation. Lantern achieves a language level backprogation (without auxiliary data structures) through callbacks. That is to say, each computation operation has access to a callback (also called delimited continuation, that represents the rest of the forward propagation and the beginning of the backward propagation). Each computation operation is overloaded to compute the forward computation, trigger the callback with the result of the forward computation as the argument, and compute backward computation after the callback returns. When multiple such operations are stacked together, the forward propagation happens when the stack of callbacks are triggered, and the backward computation happens when the stack of callbacks return. Generation of callbacks is further automated via delimited continuations operators `shift` and `reset`, which in addition achieves backpropagation via code transformation.

After backpropagation via delimited continuations, Lantern stages the intermediate representations and generate low level C++/CUDA code via LMS. The generated code of Lantern has comparable performance compared with TensorFlow and PyTorch, we report some of the evaluation on different well-known models in Figure 5.

2.3 LMS

LMS (Lightweight Modular Staging) is a multi-stage generative programming framework that provides runtime compilation and code generation in Scala. LMS introduces a special staged type constructor called `Rep[T]` (where `T` is a type e.g., `Int`, `String`, etc.) in order to distinguish expressions to be evaluated in future stages. LMS provides typed API for staged `Rep[T]` that hides internal implementation and guarantees type and value correctness. This means that all operations or language constructs (if, while, ...) on Int are executed at compile time, and all operation on Rep[Int] will generate the equivalent code. Figure 6 shows a simple example of LMS.

2.4 Flare & Lantern

Figure 3 shows the integration of Flare and Lantern on the code generation level using lightweight modular staging (LMS). For ML applications, any of the ML front-ends integrated with Lantern e.g. TensorFlow can be used to write ML applications. It is also possible to use Lantern primitives directly. The computation graph is processed, staged and the code is generated to target the different back-end of Flare and Lantern. Thus, the code is optimized for both of data and ML processing. Moreover, SQL queries with ML UDFs are written in Spark SQL and optimized as part of Flare's evaluation [5]. Using the example in Figure 2, the result of the runs on multiple systems is shown in Figure 7.

3. DEMONSTRATION PROPOSAL

In our demonstration, we will show how our two systems, Lantern for machine learning and Flare for data management, can be combined to give the user powerful tools at the intersection of database and machine learning. The demo will illustrate how machine learning can be integrated with data management in two different aspects: using a full

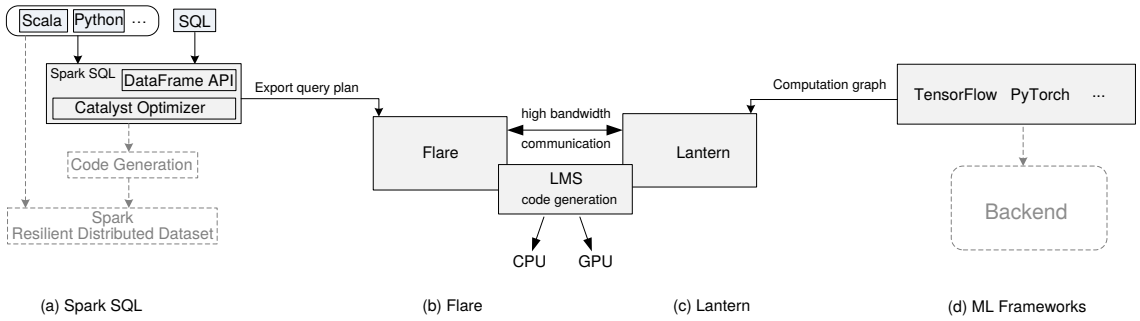
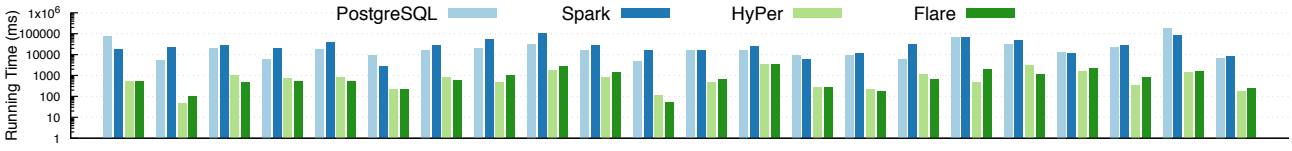


Figure 3: System overview: (a) architecture of Spark [2] (b)-(c) the integration of Flare (a Spark accelerator and query compiler) with Lantern (a machine learning framework) on the code generation level using lightweight modular staging (LMS) (d) ML frameworks



SF10	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Postgres	72599	5064	20790	6032	17904	9071	16408	20003	31728	16409	4729	15755	16152	9369	9607	5662	66254	30104	12852	22795	184736	6900
Spark SQL	17692	21062	28039	20693	40008	2826	28262	55346	98216	27585	16193	15431	25632	5913	11538	30973	66089	47542	11062	27448	78896	8687
HyPer	498	47	969	725	821	207	804	467	1782	846	111	460	3307	283	227	1112	485	3061	1666	345	1349	177
Flare	550	98	481	544	527	212	596	1016	2688	1429	53	643	3499	256	175	679	1924	1150	2236	836	1533	241

Figure 4: Performance comparison of Postgres, HyPer, Spark SQL, Flare in SF10

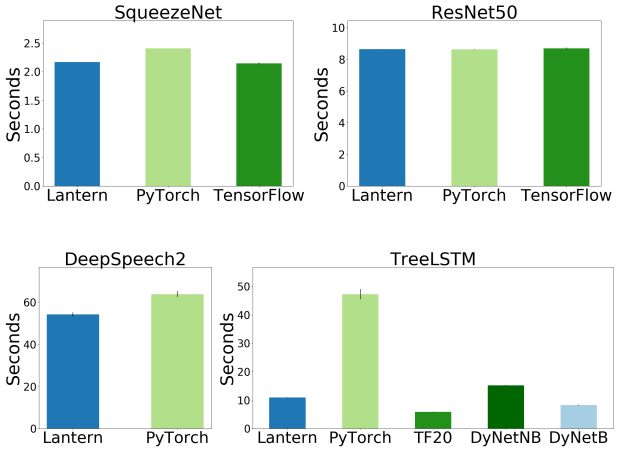


Figure 5: Runtime Performance of SqueezeNet, ResNet50, DeepSpeech2, and TreeLSTM in different frameworks. Bars in plots show the training time per epoch in seconds.

```
def power(x: Rep[Int], n: Int): Rep[Int] = {
  if (n == 0) 1
  else {
    val xx = power(x, n / 2)
    if (n % 2 == 0) xx * xx else x * xx * xx
  }
}
def main(args: Rep[String]) = println(power(args(0).toInt, 5))
```

(a)

```
int main(int argc, char** argv) = {
  int x0 = atoi(argv[0]);
  int x1 = 1;              // x^0
  int x2 = x0 * x1 * x1; // x^1 = x * x^0 * x^0
  int x3 = x2 * x2;        // x^2 = x^1 * x^1
  int x4 = x0 * x3 * x3; // x^5 = x * x^2 * x^2
  printf("%d\n", x4);
}
```

(b)

Figure 6: In the code snippet (a), the exponent n is of type Int, therefore LMS executes the branches. However the multiplications are done on Rep[Int], therefore it will generate code, the final result is displayed in (b).

DBMS system to load data for ML training, and using ML models through UDF within a query (data generation, selection function etc.).

The attendees will learn how to elegantly build a ML training pipeline using Flare and Lantern in a very high-level representation and see how the combined systems can generate a standalone, highly efficient binary from it. During the presentation, the attendees will have access to a Jupyter notebook with some predefined demos or interactively enter their own. Multiple datasets will be available: SNAP Memetracker [7], TPC-H or -DS [13], Imagenet [4]. We will show how we generate machine learning code that directly queries and loads training data from database within the training loop with various kinds of data filtering, data aggregation, joins, and data augmentation, thus optimizing the whole data path. The audience will see how using the full strength of a database system can open the exploration of new training strategies for neural networks, e.g. filtering poor samples, trying a different order to load the data, or investigate any strategy in order to improve the convergence speed. Then the audience will experience how we can integrate pre-trained machine learning models as UDFs for database queries, whether for a selection operator or within a GROUP BY operator. For example, the audience will see how to use speech-to-text algorithm using a stream of

#Data Points	Spark	Spark + JNI	Flare
200	11909	990	0.064
2000	522471	3178	0.503

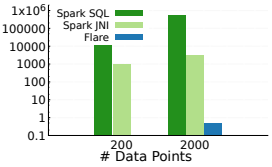


Figure 7: Running time (ms) of query in Figure 2 using TensorFlow in Spark and Flare.

recognized words from live spoken audio within the query:

```
select author, quote
from raw_audio, quotes
where quote contains speech2text(raw_audio.data)
```

Or a image recognition algorithm using a video input:

```
select name, count(*)
from imagenet, raw_camera
where label == resnet.classify(raw_camera.data)
group by name
```

The audience will be able to compare our system directly with Spark, PyTorch and Tensorflow by running the examples side by side and observe the difference in runtime. In addition they will have access to the code generated for the different queries that will be presented and see how the data path is optimized to avoid unnecessary processing.

4. RELATED WORK

Delite [11] is a general purpose compiler framework, implements high-performance DSLs (e.g., SQL, Machine Learning, graphs and matrices), provides parallel patterns and generates code for heterogeneous targets. The Distributed Multiloop Language (DMLL)[3] provides rich collections and parallel patterns and supports big-memory NUMA machines.

Weld [9] is another recent system that aims to provide a common runtime for diverse libraries e.g., SQL and machine learning. Weld is using an IR similar to DMLL that support nested parallel structures. The system is optimizing externally written libraries into a common IR.

Delite and Weld are earlier approaches of integrating ML and data management. Their performances come from the analysis of the IR and loop fusion operations. The current work is distinguished by demonstrating best of breed performance for both state of the art deep learning models e.g., SqueezeNet and relational benchmarks e.g., TPC-H. In addition, Delite and Weld differ from ours by their multi-pass compilation process. Flare and Lantern emit directly the same IR that can then be optimized in a single pass.

Lightweight modular staging (LMS) [10] is a library-based generative programming and compiler framework that uses generative programming abstractions, operator overloading and other features in regular general-purpose languages to generate code.

5. REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems, 2015.

[2] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in Spark. In *SIGMOD*, pages 1383–1394. ACM, 2015.

[3] K. J. Brown, H. Lee, T. Rompf, A. K. Sujeeth, C. De Sa, C. Aberger, and K. Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. CGO 2016, pages 194–205. ACM, 2016.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[5] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *OSDI*, pages 799–815. USENIX Association, 2018.

[6] Y. Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Transactions of the Institute of Electronics and Communication Engineers of Japan*, 54-C(8):721–728, 1971.

[7] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[8] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[9] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab. Weld: A common runtime for high performance data analytics. In *CIDR*, 2017.

[10] T. Rompf and M. Odersky. Lightweight Modular Staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[11] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *TECS*, 13(4s):134, 2014.

[12] R. Y. Tahboub, G. M. Essertel, and T. Rompf. How to architect a query compiler, revisited. In *SIGMOD Conference*, pages 307–322. ACM, 2018.

[13] The Transaction Processing Council. TPC-H Version 2.15.0.

[14] F. Wang, J. M. Decker, X. Wu, G. M. Essertel, and T. Rompf. Backpropagation with callbacks: Foundations for efficient and expressive differentiable programming. In *NeurIPS*, pages 10201–10212, 2018.

[15] F. Wang and T. Rompf. A language and compiler view on differentiable programming. *ICLR Workshop Track*, 2018.

[16] F. Wang, X. Wu, G. M. Essertel, J. M. Decker, and T. Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *CoRR*, abs/1803.10228, 2018.