Using Dataflow for Machine Learning Inference

Harikaran Subbaraj

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-75
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-75.html

May 28, 2020

Copyright © 2020, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Using Dataflow for Machine Learning Inference

by Harikaran Subbaraj

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Joseph Gonzalez
Research Advisor

May 27, 2020

(Date)

* * * * * * *

Professor Michael Ball
Second Reader

May 28, 2020

(Date)

UNIVERSITY OF CALIFORNIA, BERKELEY

MASTER PROJECT REPORT

Using Dataflow for Machine Learning Inference

Author:
Harikaran SUBBARAJ

Research Advisor:
Dr. Joseph GONZALEZ

Second Reader:
Michael BALL

*A project submitted in fulfillment of the requirements
for the degree of Master of Science Plan II*

in the

Department of Electrical Engineering and Computer Sciences

May 28, 2020

Abstract

The need for systems to support machine learning inference has grown as the importance of machine learning in production systems has increased. Serving pipelines of machine learning models comes with challenges of scaling, low-latency requirements for requests, and high computation costs for different stages of the pipeline. In this paper we propose that by taking a dataflow abstraction we can simplify and increase performance of serving these machine learning pipelines. The proposed system FLOWSERVEcombines this dataflow paradigm with Cloudburst, a stateful function as a service (FaaS) system to provide a framework to deploy and serve machine learning pipelines at scale. We provide several logical and physical optimizations that make FLOWSERVEoutperform currently used research and industry systems.

Contents

Abstract **i**

1 Introduction **1**

1.1 Machine Learning 1

2 Background **3**

2.1 Dataflow for Prediction Serving 3

2.1.1 Optimizing Prediction Pipelines 3

2.1.2 Cloudburst 4

3 Related Work **6**

3.1 Research Systems 6

3.1.1 Clipper 6

3.1.2 InferLine 6

3.1.3 ParM 7

3.1.4 Dataflow Abstractions 7

4 FlowServe **8**

4.1 Architecture and API 8

4.1.1 Dataflow API 8

4.1.2 Prediction Serving Control Flow 9

4.2 Optimizing Dataflows 11

4.2.1 Operator Fusion 11

4.2.2 Competitive Execution 12

4.2.3 Fine-Grained Autoscaling 12

4.2.4 Data Locality 13

4.2.5 Batching 14

5 Performance Analysis **15**

5.1 Evaluation 15

5.1.1 Optimization Microbenchmarks 15
Operator Fusion 15
Competitive Execution 16
Locality 17
Batching 19

6 Conclusion **21**
6.1 Future Work 21
6.2 Closing Summary 21

Acknowledgements **22**

Bibliography **23**

1 Introduction

1.1 Machine Learning

Machine learning has been an increasingly valuable tool used in a wide array of applications by many different industries. TV show recommendations on streaming services such as Netflix, the software behind self-driving cars, and language translation services are just a few of the many ways that users interact with machine learning on a daily basis. The machine learning cycle can be broadly separated into two categories: training and inference. Most of the systems side research in the machine learning discipline is focused on streamlining and improving the training side of machine learning. There are many production-grade libraries that are focused on making developing models easier, such as Tensorflow, and PyTorch. There are also tools that simplify training models on "Big Data" that cannot fit on a single machine, such as MLLib and Ray RLLib. However, taking these trained models and deploying them at scale for real-time inference (also known as prediction serving) offers a different set of challenges.

The challenges to machine learning inference can be broken down into four major categories:

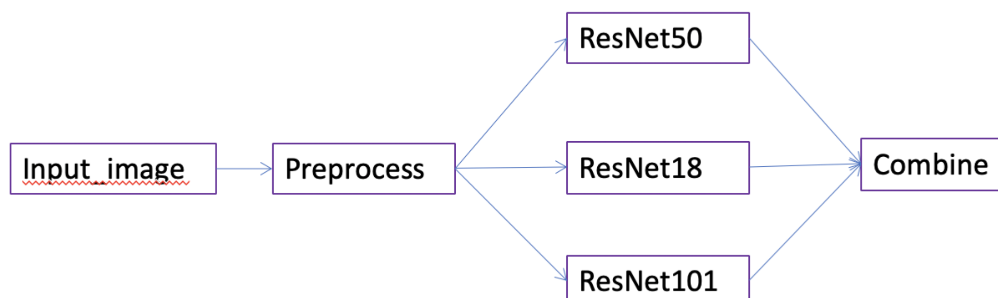

FIGURE 1.1: An example prediction serving pipeline to classify an image using an ensemble of three models. The models are run in parallel, and the results are combined after they finish.

- High computation costs.
- Low-latency requirements for real-time requests.
- Support for multiple model frameworks
- Composition of multiple models for single prediction. Figure 1.1 is an example of a machine learning inference pipeline that makes use of multiple different models to serve a prediction.

Today, most people build their machine learning inference solutions on top of a cloud provider based service such as AWS Sagemaker/Azure ML or use Flask servers deployed as microservices. Both of these approaches are limited and do not scale well, which is a major problem given the growth in interaction with machine learning applications. Building a series of Flask servers brings up problems of container orchestration and data shipping between containers to serve an entire pipeline. Sagemaker doesn't support running models in parallel and thus cannot support pipelines such as Figure 1.1
We simplify these seemingly complex pipelines by providing an abstraction that models machine learning inference pipelines as a series of dataflow operators. Pipelines can be constructed by using the following operators in combination: MAP, FILTER, JOIN, AGG, GROUPBY, LOOKUP.
Using this abstraction provides a familiar interface for engineers and data scientists who use Pandas. The pipeline in Figure 1.1 would be expressed as:

- MAP for preprocess stage
- Parallel MAPs for the three models
- JOIN to combine results
- MAP final prediction

We present FLOWSERVE, a dataflow system for predcition serving pipelines. FLOWSERVE provides and easy to use API for constructing pipelines, and applies common dataflow and prediction serving optimizations such as operator fusion and competitive execution to optimize those pipelines. These improvements are applied without user interaction and improve performance of the pipeline.

2 Background

2.1 Dataflow for Prediction Serving

Machine learning inference is now an integral part of many production applications, from picking which ads to show a consumer to predicting customer churn. As these use cases become more and more complex, however, the needs cannot be solved efficiently by a single model. Rather, these predictions are mad by creating a entire pipeline composed of different models that perform smaller tasks in parallel or sequence.

This model of a pipeline offers a nice comparison to a dataflow DAG, where each stage receives an input, performs a computation, and then sends the output to the next model in the graph. This DAG approach simplifies the deployment of the pipeline and efficient movement of the data.

2.1.1 Optimizing Prediction Pipelines

We categorize the optimizations to machine learning inference on pipelines into two categories: logical and physical. The logical optimizations are operator fusion and competitive execution, and the physical optimizations are fine-grained autoscaling, batching, and data locality. These optimizations are a combination of best practices from different dataflow and machine learning inference systems [2, 9]. Below is the description of each of the optimizations:

Operator Fusion. Dataflow systems models each operator in the pipeline as a separate computational stage. With large inputs, however, the cost shipping data and serialization can be expensive. With operator fusion, when there is a sequence of events without branching, multiple logical operators can be merged into a single physical operator, which avoids the cost of data movement but also maintains the logical separation of stages. For example, in a cascade pipeline, the preprocessing stage and model stage can be fused together.

Competitive Execution. Depending on inputs, machine learning models can have highly variable execution times. Google's MapReduce system [4] pioneered the idea of using competitive execution for straggler mitigation. This same idea has been used to improve the tail (e.g., 99th percentile) latencies of machine learning models. In short, we will trigger multiple copies of the model in parallel and return the first returned result.

Fine-Grained Autoscaling. The various stages of a prediction pipeline have different runtimes, computational characteristics, and require different resources,
We want the ability to scale effectively on a per model-level. For example, in pipeline 1, we would want to allocate more resources to the resnet since the preprocess stage is relatively quick. This level of autoscaling prevents bottlenecks and allows for efficient use of expensive resources like GPUs.

Data Locality. Model inference tasks sometimes involve large data, both in the form of large inputs and lookups from large tables. For example, consider pipeline that recommends shows from a vast but sparse dataset. By executing models on machines that already have this data located on them and exploiting data locality, we lower lookup costs and improve performance.

Batching. Most models today are optimized to take in a batch of inputs versus a single input at a time. By batching requests together, we can improve the throughput of the system without affecting the latency much. In addition, GPUs are highly efficient at computing on batches of data, so batching will take advantage of this hardware as well.

2.1.2 Cloudburst

Cloudburst [11] is a stateful Functions-as-a-Service (FaaS) platform. The key goal of the system is to enable stateful serverless programming by enabling three kinds of state sharing: function composition, message passing, and caches that are located on the same machines where code is run.
[11] describes the architecture and implementation of Cloudburst. The system is built on top of Anna [13, 12], a low-latency autoscaling key-value store. Layered on top of the KVS is a set of function executor nodes, each of which has threads which respond to user requests and a cache that intermediates on KVS reads and writes. The system heuristically optimizes for data locality by scheduling requests on nodes

where input data is likely to be cached. With this architecture, Cloudburst is able to outperform commodity FaaS platforms by orders of magnitude for stateful tasks and significantly cuts data transfers costs.

We chose Cloudburst as the execution engine for FLOWSERVE because it directly supported the goals of our system. Serverless systems are a natural fit for dataflow execution because they naturally encourage a functional programming model. And the low function composition and data retrieval costs support our latency constraints. Despite these benefits, Cloudburst had a number of key limitations that we had to overcome—we describe these in detail in Section 4.2.

3 Related Work

3.1 Research Systems

3.1.1 Clipper

Clipper is a research project out of the RISE lab that provides a low-latency system for machine learning inference. Clipper offers strong guarantees to meet a user defined latency SLO for deployed models. Clipper packages user-defined models and deploys them in a Docker container. It then exposes a REST API for the client production system to query. Clipper deploys a middle layer that contains a request queue for each deployed model. Incoming requests are placed on the queue and sent as a batch to the model when the model container polls the queue. This improves throughput of the system greatly, and was a inspiration for FLOWSERVE's batching architecture.

Clipper employs an adaptive batching scheme that adjusts the batch size to achieve maximum throughput while still meeting the latency SLO requirement. It uses an explore/exploit scheme to try different batch sizes until it violates the latency SLO. However, Clipper was built to serve queries to single models, not machine learning pipelines.

3.1.2 InferLine

Inferline[3] is a research system that was built on top of Clipper to do machine learning inference on pipelines. Inferline provided three major components on top of Clipper to serve machine learning pipelines: profiler, planner, and a reactive controller.

Offline, the profiler would take a user-defined inference pipeline and derive the DAG form of the pipeline. Using a user-provided example trace, the offline planner would estimate the end-to-end latency of the pipeline. This proactive planner

uses a cost-optimizer that takes into account batch size, replication factor, and cost of hardware to provide the optimal configuration for the given latency SLO.
Once the pipeline is live, the reactive controller takes into account incoming traffic and adjusts the replication factor on a model-level basis to improve performance and maintain latency SLOs of the system.
The DAG structure of machine learning pipelines was a useful inspiration for the dataflow paradigm. FLOWSERVE improves on the scalability of InferLine by using Cloudburst, a FaaS system that minimizes resource usage. Cloudburst also provides furthuer optimizations to reduce data shipping between stages of the pipeline.

3.1.3 ParM

ParM [8] is a research system that aims to reduce tail latencies and model failures in machine learning inference systems. In addition to sending queries to models deployed in Docker containers, similar to Clipper and Inferline, ParM encodes batches of queries using erasure codes and sends these "parity queries" to special replications of the models called "parity models". In the case of a unavailable or failed prediction, these models can construct an approximate prediction from the other values in the parity query. This allows ParM to greatly reduce the tail latencies that are common in machine learning inference systems. ParM is focused on reducing the p99 latencies and prediction failures and does not offer the optimizations that FLOWSERVE uses to reduce median latencies in prediction serving workloads. Adding replications of parity models to FLOWSERVE's architecture would be an interesting future optimization that would reduce tail latenices of certain workloads.

3.1.4 Dataflow Abstractions

In the past few years, dataflow has emerged as a common abstraction for constructing systems around streaming data. Systems such as Naiad[10], Flink[1], and Noria [5] implement stateful dataflow that targets scenarios where the input data continuously streams into the system. In addition, systems such as Spark[14] and Dryad[7] use this dataflow abstraction to deal with high throughput data processing. To adapt dataflow to work for low-latency, bounded queries in interactive machine learning inference pipelines, FLOWSERVE adds additional optimizations such as competitive execution and fine-grained autoscaling.

4 FlowServe

4.1 Architecture and API

Here we describe FLOWSERVE's API, it can be used simplify machine learning inference pipelines, and how FLOWSERVE pipelines are implemented on top of Cloudburst. As explained in Section 2.1.2, we use Cloudburst's serverless execution runtime because it enables stateful serverless programming.

4.1.1 Dataflow API

The FLOWSERVE API consists of simple dataflow operators that all can bes used with the core data abstraction, a `Table`. `Table` is an in-memory relational that has a predefined schema, in which each row represents a query to the pipeline and has a unique queryID. Every stage of the pipeline takes in a `Table` as an input and returns a `Table` as an output.

Queries to the same pipeline are added to the `Table`, after which the operators are applied to define the pipeline.

Table 4.1 provides a brief overview of the operator API. There are 6 key operators: `map`, `filter`, `groupby`, `agg`, `lookup`, and `join`. The `map`, `filter`, `groupby`, `agg`, and `join`operators are used to define the machine learning inference pipeline. The special `lookup`operator serves as the API for FLOWSERVE to query ANNA, the KVS that Cloudburst is built on top of. FLOWSERVE takes advantage of metadata from `lookup`to leverage cloudburst's locality-based scheduling. `lookup` operators retrieve data from Anna at runtime in one of two ways:

Executing Dataflows on Cloudburst. FLOWSERVE uses Cloudburst as its execution engine. Each FLOWSERVE operator is registered as a Cloudburst function and the whole dataflow is registered as a Cloudburst DAG—Section 2.1.2 has more details on the Cloudburst API. At execution time, Cloudburst schedules and executes the functions as described in [11], and results are stored in Anna, Cloudburst's underlying key-value store. `lookup` operators retrieve data from Anna at runtime in one of

API Name	Functionality
`map`	Apply function to each element in the `Table`
`filter`	Apply Boolean function to each element in the `Table` and keep only `true` results
`groupby`	Group elements in the `Table` by the value in the given column
`agg`	Apply a predefined aggregate function (count, sum, min, max, average) to the `Table`
`lookup`	Retrieve an object from the underlying KVS and insert into the `Table`
`join`	Join two `Tables` on a given key, using the automatically assigned query ID as a default

TABLE 4.1: An overview of the core operators supported by FLOWSERVE.

two ways: (1) query the KVS directly (2) create references, which Cloudburst uses to schedule functions on the same machine where data is cached to lower data shipping costs.

4.1.2 Prediction Serving Control Flow

Here we describe how FLOWSERVE's dataflow programming model simplifies the implementation of control flow constructs that are common in prediction serving pipelines. We briefly describe each construct and highlight its simplicity with a code snippet.

Ensembles. Ensemble pipelines have a branch where the same input is sent to multiple models to compute in parallel. The pipeline then combines the results by either taking a weighted vote of the results or returning the prediction with the highest confidence. There are also other aggregation types that common ensemble methods use.
Two models are evaluated in parallel (after a preprocessing stage), the results are joined, and a final prediction is selected by the `pick_best_prediction` function.

Cascades. In a model cascade, models of increasing complexity are executed in sequence. To avoid unnecessary computation and latency, if an earlier, simpler model returns a prediction with a confidence above a certain threshold, the latter models are not run. This way you get sufficiently confident results with much better latency. Figure 4.2 shows the FLOWSERVE implementation of a model cascade.

```python
flow = Flow('ensemble-flow', FlowType.PUSH, cloudburst)
img = flow.map(transform, init=transform_init, names=['img'])

anet = img.map(alexnet_model, init=alexnet_init, names=['alexnet_index', 'alexnet_perc'])
rnet = img.map(resnet_model, init=resnet_init, names=['resnet_index', 'resnet_perc'])
anet.join(rnet).map(ensemble_predict, names=['class'])

flow.deploy()
```

FIGURE 4.1: Script to execute FLOWSERVE ensemble pipeline.

```python
flow = Flow('cascade-flow', FlowType.PUSH, cloudburst)
rnet = flow.map(trans,
                init=transform_init,
                names=['img'],
                batching=gpu) \
          .map(resnet,
               init=resnet_cons,
               names=['img', 'resnet_index', 'resnet_max_prob'],
               high_variance=True,
               gpu=gpu,
               batching=gpu)

incept = rnet.filter(low_prob) \
          .map(incept,
               init=incept_cons,
               names=['incept_index', 'incept_max_prob'],
               gpu=gpu,
               batching=gpu)

rnet.join(incept, how='left') \
    .map(cascade_predict, init=cascade_init, names=['class'])
flow = optimize(flow, rules=optimize_rules)
flow.deploy()
```

FIGURE 4.2: Script to execute FLOWSERVE cascade pipeline.

```
flow = Flow('compete')
img = flow.map(preproc)
result = img.map_compete(simple_model, medium_model, complex_model)

result.map(pick_first)
```

FIGURE 4.3: Script to execute FLOWSERVE cascade pipeline.

After the simple model is executed, we filter out any predictions that are high confidence. We join these filtered outputs with the `img` row to bring the original input back into the `Table` and then apply the complex model. We join the simple and complex models results using a left outer join to ensure we include results that were filtered out by the low confidence filter as well as results from the complex model. We use a `ensemble-predict` function again to select the prediction with the highest confidence.

Competitions. This pipeline is similar to the competitive execution optimization that we will discuss later. In this pipeline, multiple models are executed in parallel, and the first returned model of high enough confidence is returned.
Figure 4.3 shows an implementation in FLOWSERVE. The `map_compete` macro takes multiple functions and applies each one in parallel. The function that is downstream from those parallel functions (in this case `pick_first`), will be configured to wait for *one* result rather than all results, but will only accept the result if it meets the confidence threshold.

4.2 Optimizing Dataflows

In this section, we describe how we implement each of the optimizations from Section 2.1.1: operator fusion, competitive execution, fine-grained autoscaling, locality, and batching. As we describe each optimization, we also describe extensions we made to Cloudburst to support those optimizations.

4.2.1 Operator Fusion

Operator fusion is the simplest optimization we implement: Multiple logical operators are merged into a single physical operator. FLOWSERVE supports fusion by implementing a `multi` operator, which is a meta-operator that wraps two or more

of the operators detailed in Section 4.1.1. The `multi` operator is treated as a single function that executes on a single location, saving costs of data shipping.

4.2.2 Competitive Execution

Competitive execution is used to reduce the tail latency (95th or 99th percentile) of operators that have highly variable execution times. In order to implement this, we create multiple parallel replicas of the operator. We then need to return the result of the first finished computation. By default Cloudburst's DAG execution API assumes that for a function to be executed, every function that precedes it in the DAG must have finished executing (*wait-for-all* semantics). To support competitive execution—both as an optimization and as a form of control flow (see Section 4.1.2—-we would like to execute many replicas in parallel and pick the first result that is returned (*wait-for-one*). We modified Cloudburst's DAGs to support a *wait-for-one* execution mode. In this mode, the Cloudburst executors will execute a function as soon as it receives *one* result from an upstream function, rather than waiting for *all* upstream functions.

Optionally, users can provide invalid responses for these functions—if the function returns an invalid response, it will be re-queued instead of being marked finished. The executor will then wait for another result from an upstream function before executing the function again. If all upstream results cause the function to return an invalid response, then the user will receive an error.

4.2.3 Fine-Grained Autoscaling

Each operator in a dataflow will naturally have different properties—memory consumption, resource requirements, variability, runtime, and so on. This is particularly pronounced in prediction serving, where for example a pipeline might have a CPU-intensive preprocessing stage followed by a GPU-based model evaluation stage. In this context, it is useful to be able to allocate resources in a fine-grained fashion, specifically to the stages of the pipeline that are performance bottlenecks. Continuing our example, if the preprocessing stage was serialized and slow, while the GPU stage was efficient and supported batching, it would be unwise (and expensive!) to scale all parts of the pipeline uniformly.

FLOWSERVE's dataflow model makes it easy to scale in a fine-grained way. Since each operator is registered and deployed as a separate Cloudburst function,

FLOWSERVE's pipelines are well-suited to take advantage of existing autoscaling techniques on serverless infrastructure. Furthermore, the dataflow model ensures that users can focus on pipeline logic, while FLOWSERVE is responsible for managing how pipeline stages are combined and deployed.

Cloudburst, like all FaaS systems, natively supports autoscaling by adding and removing replicas of individual functions as load changes. Since FLOWSERVE was designed to run on Cloudburst's DAG API, we are able to naturally leverage the autoscaling system it provides. We also extended the autoscaling system to support multi-class autoscaling for GPU and CPU resources (see Section 4.2.5 for more details).

4.2.4 Data Locality

An important part of many prediction pipelines are data retrievals—for example, a recommender system might first look at a user's recent history, then query the database for a set of candidate products and their weights before returning a set of recommended items. This is why we support `lookup` as a first-class operator in FLOWSERVE. However, simply gathering data and shipping it to downstream dataflow operators is slow and expensive. Cloudburst's API, however, requires users to specify data accesses before each request is executed, which conflicts with the dynamic nature of many prediction pipelines. To avoid data shipping, we implement two optimizations. First, we fuse the `lookup` operator with whatever is downstream from it to avoid shipping data between operators, and second we implement "continuations." Within a pipeline, FLOWSERVE automatically rewrites `lookup` operators to leverage continuations—users simply specify what `Table` column has their dynamic lookup key. FLOWSERVE automatically converts that into a KVS reference that Cloudburst can process and inserts a continuation that Cloudburst can interpret.

Cloudburst: Continuations. Cloudburst, by default, optimizes DAG schedules for data locality by attempting to schedule functions on machines that might have data accesses cached. As described above, these lookups are often dynamic, meaning Cloudburst's scheduler does not have the opportunity to optimize for locality.

To avoid this pitfall, we added support for *continuations* in Cloudburst. This enables multiple DAGs to be chained together as a part of a single logical request. After

one DAG finishes executing, rather than returning the result to the user, the executor that finished the DAG will send the result to the scheduler, which will use the result as the input to the continuation DAG. As usual, the scheduler will inspect the arguments to look for KVS references, which it will then use to optimize its function placement. This allows Cloudburst to ensure that a dynamically generated data lookup will still be executed on a machine where the data is likely to be cached.

4.2.5 Batching

Batching is a well-known optimization for prediction serving workloads, particularly when systems can take advantage of the extreme parallelism afforded by hardware accelerators like GPU. Clipper [2] pioneered the technique of loading single requests that arrive at the same time on a queue and shipping them as a batch to the model. FLOWSERVE's dataflow model makes batching particularly easy because it does not require any user program modifications; whether an operator receives one input or many, they can merged into a single `Table` and treated as a batch.

Cloudburst: Batching & GPUs. To support batching as a possible optimization—particularly for pipelines that use GPUs—we had to modify Cloudburst's executors. We extended the Cloudburst API to annotate functions that support batching as a part of a DAG. When an executor thread is executing a batching-enabled function, it dequeues all execution requests it has received (by default, Cloudburst threads only dequeue one request at a time). It treats this group of dequeued requests as a batch and executes all of them in a single function call. The maximum batch size is configurable and defaults to 20 requests. In addition, we modified the kubernetes configuration to add an nvidia-kubernetes-plugin that installs packages on the function executor nodes that allow the function pods to communicate with the GPUs. These optimizations together help FLOWSERVE to increase throughput greatly.

5 Performance Analysis

5.1 Evaluation

In this section, we study FlowServe's performance in detail. Section 5.1.1 first studies each of the optimizations discussed in Section 4.2 in isolation, using synthetic workloads.

We ran all experiments in the `us-east-1a` AWS availability zone. All Cloudburst function executors were run on `c5.2xlarge` instances, except machines with GPUs which were `p2.xlarge` instances.

5.1.1 Optimization Microbenchmarks

We first present microbenchmarks that study each of the optimizations detailed in Section 4.2—operator fusion, competitive execution, locality, batching

Operator Fusion

Our first microbenchmark studies the benefits of operator fusion on linear chains of functions. As discussed earlier, the main benefit of fusing operators is avoiding the cost of serializing and shipping data between compute locations. Correspondingly, the experiment varies two parameters: the length of the function chain and the size of data passed between functions. The functions themselves do not do any computation: They take an input of the given size and return an output of the same size. The output is passed downstream to the next function in the chain.

For each combination of chain length and size, we measure an optimized (fused) pipeline as well as an unoptimized pipeline. The fused pipelines execute all n functions in a single Cloudburst function, while the unfused pipelines execute each stage in a separate Cloudburst function. Figure 5.1 reports median (box) and 99th percentile (whisker) latencies for each setting.

As expected, for each input data size, the median latency of the optimized pipelines is constant, with minor variations. The 99th percentile latencies have slightly more

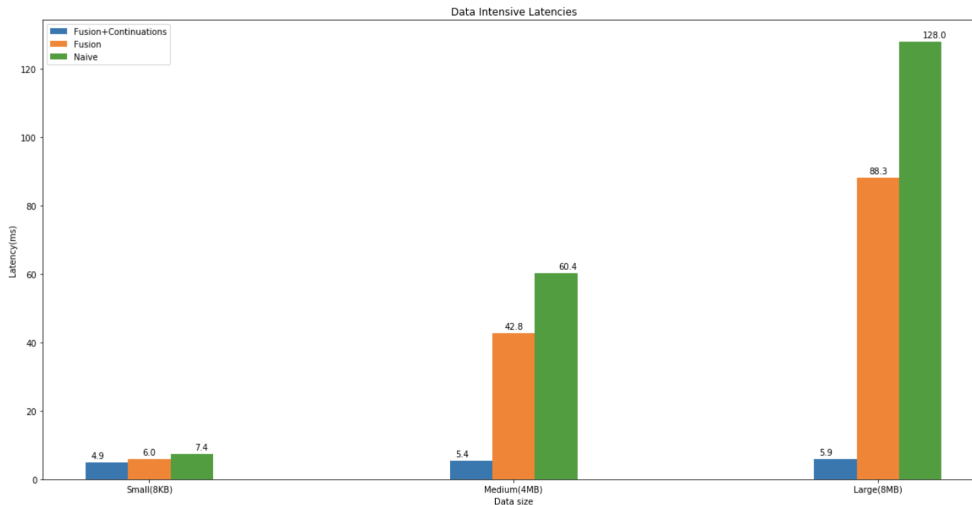

FIGURE 5.1: A study of the benefits of operator fusion as a function of chain length (2 to 10 functions) and data size (10KB to 10MB). We report median latency of each configuration In brief, operator fusion improves performance in all settings and achieves speedups of 3-5× for the longest chains of functions.

variation, which is generally expected of tail latencies, but there is no discernible trend in the measurements, and the variations are not significant. The latencies of the unoptimized pipelines in each data size increase roughly linearly with the length of the function chain. This is again as expected, as the cost of data movement will increase linearly with the length of the chain. While performance is relatively close for the smaller chains (improvements of of 20-40%), fusing longer chains of functions leads to improvements of 3-4×.

***Takeaway**: Operator fusion in* FLOWSERVE *can lead to improvements of up to 4× in latency by avoiding the overheads of data serialization and data movement between function executors.*

Competitive Execution

Next, we turn our attention to reducing tail latencies for operators that have high variance in runtime using competitive execution. As discussed in Section 4.2, the general approach to reducing tail latencies is to execute multiple replicas of the high variance operator in parallel and simply select the result from the replica that finishes executing first. To model such a workload, construct a 3-stage pipeline in which the first and third operators are pass-through operators that do no computation. The second function draws a sample from one of three Gamma distributions, with low, medium, and high variances, respectively. The function sleeps for the

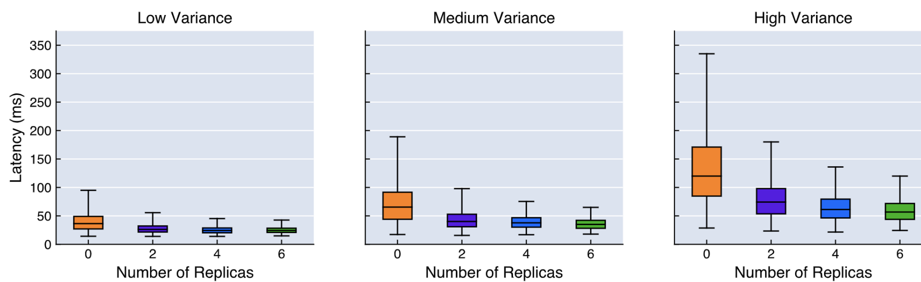

FIGURE 5.2: Measured latency distributions (1st, 25th, 50th, 75th, and 99th percentile) as a function of the number of additional replicas computed of a high-variance function. The runtime of the function is drawn from the Gamma distributions visualized in Figure 5.2. Adding more replicas reduces both median and tail latencies, especially for the high variance function.

amount of time in the sample (in milliseconds) before returning and triggering the next third stage. Our Gamma distributions have a fixed shape parameter (α) of 3.0 and have shape parameters (β) of of 1.0, 2.0, and 4.0; the higher the shape parameter, the higher the variance of the distribution. Thus, the variance in runtime of the whole dataflow is dependent on the value drawn from the corresponding Gamma distribution.

To measure the benefits of competitive execution, we add additional replicas of the high-variance function, as discussed in Section 4.2.2. Figure 5.2 shows our results; the boxes on each graph show the interquartile range, and the whiskers show the 1st and 99th percentile latencies. In all cases, increasing from 1 replica to 3 replicas reduces tail latencies significantly: 71%, 94%, and 86% for low, medium, and high variance respectively. Similarly, median latencies reduced by 39%, 63%, and 62% for each of the settings.

Beyond 3 replicas, improvements vary. For the low variance setting, increasing from 3 to 7 replicas yields a 30% improvement in tail latency and only a 7% improvement at median. However, for the high variance settings, there is a 50% improvement in 99th percentile latency and a 31% improvement in median latency.

***Takeaway**: Increasing the number of replicas reduces both tail and median latencies, with particularly significant improvements for extremely highly-variable dataflows.*

Locality

Next, we look at the benefits of data locality. As we described in Section 4.2, data locality in FLOWSERVE is achieved with two dataflow rewrites: fusing `lookups` with

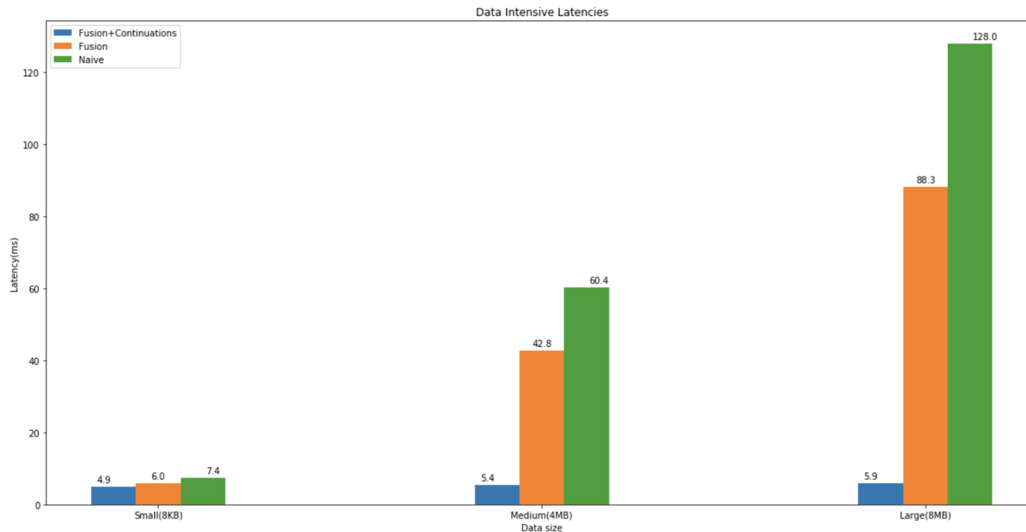

FIGURE 5.3: Median latency for a data-intensive pipeline on FLOWSERVE with the fusion and continuations optimizations enabled, only fusion enabled, and neither enabled. The pipeline retrieves large objects from storage and returns a small result; FLOWSERVE's optimizations reduce data shipping costs by scheduling requests on machines where the data is likely to be cached. For small data, the data shipping cost is only a milliseconds, but for the medium and large inputs, FLOWSERVE's optimizations enable orders of magnitude faster latencies.

downstream operators and inserting continuations to take advantage of Cloudburst's scheduling heuristics. To measure the benefits of locality, we measured the incremental benefits of both these rewrites.

We picked a representative task in which a small set of objects (in our case, 100) was each accessed a few (10) times in a random order. The pipeline consists `map` to dynamically pick which object to access, followed by a `lookup` of the object, followed by a second `map` to compute a result (the sum of elements in an array). We vary the data size from 8KB and 8MB, and Figure 5.3 shows our results. We warm up the caches in Cloudburst by issuing requests for each of the data items once before starting the benchmark.

The Naive bar implements neither optimization—data is simply retrieved from the KVS in the execution of a regular operator and shipped downstream to the next operator. The Fusion Only bar merges the `lookup` operator with the subsequent map operator to avoid data movement costs. The Fusion + Continuations bar inserts a continuation between the first `map` and the `lookup` to leverage locality.

For small data sizes, our optimizations yield minimal wins—the cost of shipping 8KB of data is negligible, and the Naive implementation is only 2.5ms slower than

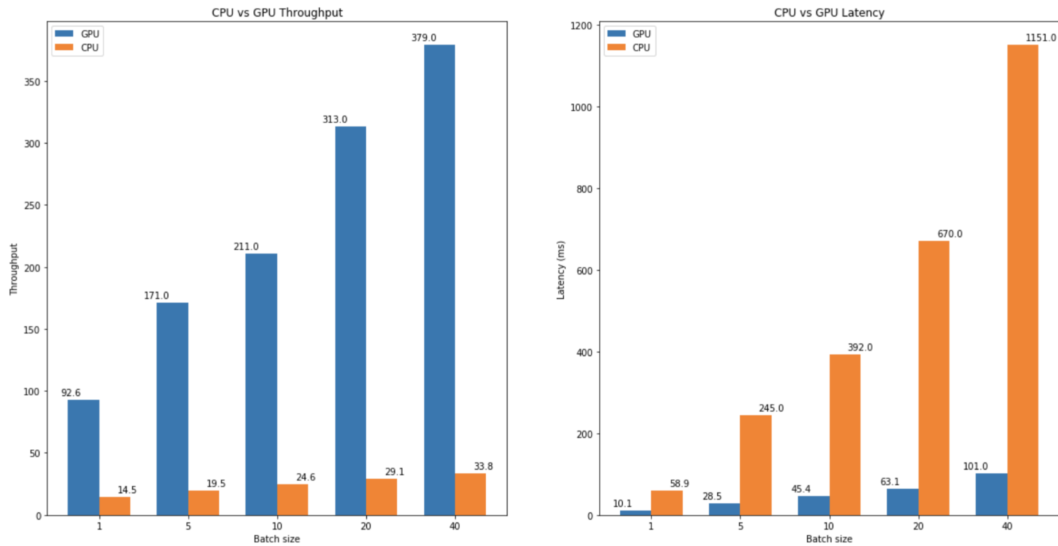

FIGURE 5.4:

having both optimizations implemented. As we increase data size, the Naive performance significantly worsens—for each request, the data is moved once from the KVS to the cache and again from the `lookup` to the second `map`. Similarly, the Fusion Only operator avoids *one* round of data shipping (between operators), but still must retrieve data from the cache. With the continuations optimization implemented, FLOWSERVE is able to take advantage of locality-aware scheduling and for the largest is 15× faster than Fusion Only and 22× faster than the Naive implementation. Tail latencies, however, do increase with data size for the optimized version, as the handful of requests that incur cache misses will still pay data shipping costs.

Takeaway: FLOWSERVE's *continuations and fusion enable it to take advantage of data locality by removing data shipping costs and scheduling for data locality. This leads to an order of magnitude improvement in latencies for non-trivial data accesses.*

Batching

Finally, we look at the benefits of batching. Since batching is most often used with GPUs, we introduce GPUs in this experiment and measure the effects of batching on GPUs vs. CPUs. We picked a pipeline with a single machine learning model (the AlexNet image classification model in PyTorch) and no other operators. Importantly enabling batching only required changing two lines of code—receiving a list of multiple inputs and calling `torch.stack` to combine them into a single tensor.

We varied the batch size from 1 to 40. We asynchronously issued k requests (where k is the batch size) from a single client in order to control the batching and measure

the time until all results are returned. Figure 5.4 shows our results. The top graph in Figure 5.4 reports latencies (on a log-scale graph), and the bottom graph reports throughput.

For a batch size of 1, the GPU has roughly a 6× better latency and throughput than the CPU. As we increase the batch size, we find that the latencies for both implementations increase sublinearly with the GPU generally outperforming the CPU—between batch sizes 1 and 20, GPU latencies increased 6× and CPU latency about 11×. This 6× increase in latency is still well below the threshold for model executions in real-time pipelines [6]. Similarly, GPU throughput increased almost 3.5× while CPU throughput increased only 2×.

Past batch size 20, performance begins to trail off. For the GPU, we see a 67% increase in latency between sizes 20 and 40 and only a 20% increase in throughput. The CPU sees a 70% increase in latency and 16% throughput improvement. This is because the largest batch size is saturating the compute resources available to the model, forcing a linear execution of susbsets of the batches. Note that the optimal batch size likely varies from model to model and would need to be tuned.

Takeaway: FLOWSERVE*'s dataflow model enables simple prediction pipeline to increase performance by over 3× without significantly compromising on latency.*

6 Conclusion

6.1 Future Work

There are several different extensions to FLOWSERVE that could improve performance of the system. First, we can implement the adaptive batching policy that Inferline pioneered. Rather than taking a fixed max batch size, we can use an explore/exploit program to determine the optimal batch size based on example traces fed to the system. Adaptive batch could also be applied on a more granular, per model level, since different models may work best with different batch sizes.
Another optimization to borrow from the Inferline work would be cached inputs for the whole pipeline. By implementing a LRU cache at the input level, FLOWSERVE would be able to handle very bursty but dense request load.
There is also work to be done on sharing GPUs between functions. Currently, each FLOWSERVE function served on a GPU is isolated due to pod constraints. However, this isn't the most efficient use of the GPU, as inference often doesn't use the full capability of the GPU. There is work done on multiplexing GPUs that could be applied in FLOWSERVE as well.

6.2 Closing Summary

The goal of this project was to provide a new abstraction to think about machine learning inference for pipelines in the form of dataflow. We proposed FLOWSERVE as a system that uses this dataflow paradigm to simplify the challenges of doing inference on a pipeline. FLOWSERVE provides familiar dataflow operators as an API to deploy these inference pipelines as a DAG. FLOWSERVE provides optimizations to serve these pipelines on Cloudburst in five different areas: operator fusion, competitive execution, fine-grained autoscaling, data locality, and batching. We show that we achieve improvements on top of current research and industrial inference systems, both in microbenchmarks and end-to-end pipelines of various type.

Acknowledgements

I thank Professor Joseph Gonzalez, Professor Michael Ball, Professor Joe Hellerstein, Vikram Sreekanti, Dan Crankshaw, and Simon Mo for providing guidance and support.

Bibliography

[1] Paris Carbone et al. "Apache flink: Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[2] Daniel Crankshaw et al. "Clipper: A low-latency online prediction serving system". In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 613–627.

[3] Daniel Crankshaw et al. "InferLine: ML Inference Pipeline Composition Framework". In: *CoRR* (2018).

[4] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[5] Jon Gjengset et al. "Noria: dynamic, partially-stateful data-flow for high-performance web applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 213–231. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/gjengset.

[6] K. Hazelwood et al. "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018, pp. 620–629. DOI: 10.1109/HPCA.2018.00059.

[7] Michael Isard et al. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: Association for Computing Machinery, 2007, 59–72. ISBN: 9781595936363. DOI: 10.1145/1272996.1273005. URL: https://doi.org/10.1145/1272996.1273005.

[8] Jack Kosaian, KV Rashmi, and Shivaram Venkataraman. "Parity models: erasure-coded resilience for prediction serving systems". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 30–46.

[9] Yunseong Lee et al. "{PRETZEL}: Opening the Black Box of Machine Learning Prediction Serving Systems". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 611–626.

[10] Derek G. Murray et al. "Naiad: A Timely Dataflow System". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, 439–455. ISBN: 9781450323888. DOI: 10.1145/2517349.2522738. URL: https://doi.org/10.1145/2517349.2522738.

[11] Vikram Sreekanti et al. *Cloudburst: Stateful Functions-as-a-Service*. 2020. arXiv: 2001.04592 [cs.DC].

[12] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. "Autoscaling tiered cloud storage in Anna". In: *Proceedings of the VLDB Endowment* 12.6 (2019), pp. 624–638.

[13] Chenggang Wu et al. "Anna: A kvs for any scale". In: *IEEE Transactions on Knowledge and Data Engineering* (2019).

[14] Matei Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* 59.11 (Oct. 2016), 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664. URL: https://doi.org/10.1145/2934664.