SerFer: Serverless Inference of Machine Learning Models

Mohammed Danish Shaikh, Mohan Rao Divate Kodandarama, Shreeshrita Patnaik
{mshaikh4, divatekodand, spatnaik2}@wisc.edu
University of Wisconsin, Madison

Abstract

With the advent of serverless functions, we're seeing more and more applications trying to exploit the serverless domain. This can be attributed to three key benefits offered by serverless platforms: 1. Functions can be triggered in response to events, i.e. it adopts an event based programming model. 2. Provisioning and scalability issues are not the concern of application. 3. Serverless platforms adopt the pay-per-use model, hence prevent wastage of application resources that happens either by provisioning too much and hurting budget or provisioning too less and hurting throughput and/or latency. Machine Learning inference is a throughput and latency sensitive domain since it needs to be done in real-time. Furthermore, such inference usually involves expensive computations. We study the viability of doing such inference on serverless platforms. In doing so, we also characterize when it would be suitable to move from expensive GPU machines to serverless platforms.

1 Introduction

Serverless architecture (also known as serverless computing or function as a service, FaaS) is a software design pattern where applications are hosted by a third-party service, eliminating the need for server software and hardware management by the developer. Recently cloud providers (e.g., AWS Lambda, Google Cloud Functions) and open source projects (e.g., OpenLambda, OpenWhisk) have developed infrastructure to run event-driven, stateless functions as micro-services. In this model, a function is deployed once and is invoked repeatedly whenever new inputs arrive and elastically scales with input size.

The advent of serverless functions has spurred an interest across several application domains in trying to adopt a serverless architecture in order to leverage the benefits offered by these functions. Some examples of such adoption include PyWren [7] for distributed computing and ExCamera [6] for low latency video processing. Although distinct in their own respects, what is common to these systems is that they model a time consuming sequential computation as many small parallel computations. These parallel computations are then executed concurrently across several serverless functions, i.e. these systems utilize the serverless platform as a compute engine.

Machine learning inference consists of executing a query over a trained model. It usually involves expensive operations like matrix multiplications, and hence is computationally expensive. Furthermore, inference is usually done on the critical path of applications (e.g., voice recognition), thereby making it latency sensitive. Owing to it's computationally intensive nature, such inference is usually done on expensive GPU machines. However, there are opportunities that can be exploited to speed up inference on CPU, essentially by trying to parallelize the expensive computations. This thereby opens an application domain which might be benefited by using serverless platform as a compute engine.

In this work, we present a way for parallelizing machine learning inference across several serverless functions for deep convolutional neural network models. In doing so, we describe a prototype system, *SerFer*, developed on Python and AWS. In particular, the contributions of our work are as follows:

1. We provide a framework that orchestrates the execution of machine learning inference using serverless functions.

2. We evaluate the performance of the framework against inference using GPU, and characterize the tradeoffs.

3. We provide microbenchmarks for SerFer, pointing out the time consuming components.

4. We list down the limitations of our work, and guide the reader towards possible improvements that will help in realizing the potential of serverless functions in building throughput and latency sensitive inference applications.

2 Related Works

Although Function as a service (FaaS) is a relatively new cloud computing service, there are few work in this domain. PyWren proposed a prototype system developed in python with AWS Lambda. The main idea in the paper was to serialize required python function using cloudpickle and store it in an Amazon S3 bucket. Then, a common lambda function would fetch the data from the S3 bucket, invoke the function, and write back the results to S3. To address more general form of distributed computational model, the paper proposes to use Redis Cluster for data shuffle stages of the computation. Similar to PyWren, we use Redis Cluster for shuffle operations in our computations. However, SerFer's architecture is tailored for inference applications and makes use of several relevant optimizations.

Tensorflow serving [11] is a serving system designed to serve machine learning models in production environments, provide high-performance prediction API to simplify deploying new algorithms and experiment with new models without modifying frontend applications. Although this system supports a

variety of hardware and GPU acceleration, deploying this system in serverless environment restricts the size of the deployable model due to resource limitations on serverless functions (e.g. AWS lambda limits the size of the deployment package to 50MB). Further, since serverless platforms offer only CPU based microvms, the capabilities of GPU acceleration cannot be exploited. We address the former issue by utilizing the lambda layers feature introduced recently by AWS. The latter is addressed by splitting the model across several lambda functions to achieve model parallelism.

Locus [13] presents a performance model case study that brings to light the optimizations that a hybrid storage model can provide. Storing the intermediate results of serverless computations by utilising the hybrid model is shown to produce better performance. Our focus is not just constrained to storage, but also explores the optimizations introduced through better parallelization of intra layer computations across serverless functions.

Clipper [5] presents a general purpose low-latency prediction serving system built on top of existing machine learning frameworks such as Tensorflow [1] and Pytorch [12], [4]. It reduces prediction latency and improves the prediction throughput, accuracy and robustness by using techniques such as caching, batching and adaptive model selection. However, it does not modify the underlying machine learning framework. Unlike Clipper, we propse to build an end-to-end system for serving machine learning inference queries using serverless computing platform.

ExCamera [6] introduces the "mu" framework to run general-purpose massively parallel computations by making lambda workers execute arbitrary linux executables. The mu platform has a coordinator which is basically a long-lived server (e.g., an EC2 VM) that launches jobs and controls their execution. The coordinator describes the logic of a given computation in the form of per-worker finite-state-machine (FSM) descriptions. The coordinator receives status messages from workers and triggers the next lambda by applying state transition logic. Our driver draws parallels to the coordinator, however, our transitions are based on upload of data in storage rather than messages from the workers.

3 Challenges and Motivation

Serverless computing has the advantages of resource transparency and the pay-per-use model. Further, this platform relieves the users from the struggles of complex cluster management and configuration tools for running even simple applications. However, to fully utilize the above features offered by the serverless computing platform, we first need to address the challenges involved in developing an inference system on serverless platform. In this section, we describe the challenges and how SerFer addresses them.

3.1 Serverless Platform Limitations

The microvms currently offered by the serverless platform have several resource limitations. For example, AWS lambda currently limits the size of the deployment package to 50MB and the amount of RAM to 3008MB. Such restrictions hinder deploying modern deep learning models on serverless platform because they typically include millions of floating point operations and several millions of parameters.

We overcome the limit on package size by utilizing the lambda layers feature introduced recently by AWS. A layer is a ZIP archive that contains libraries, a custom runtime, or other dependencies. With layers, one can use libraries in a lambda function without needing to include them in your deployment package. This keeps the deployment package small. SerFer stores the parameters of the model as a dependent library within the layers and imports it into lambda functions when required. This method gives a way to increase the package size to 250MB, which can still be insufficient for many models. For example, the weights of the fully connected layers in AlexNet [9] take about 260MB of storage. Additionally, other dependent packages such as deep learning frameworks (e.g. Pytorch) require about 100MB of storage size. To address this, SerFer quantizes the weights of the fully connected layers and represents them as 16 bit floating point numbers. Another approach that SerFer employs is to use global average pooling layers instead of fully connected layers.

In our experiments with AlexNet, we found that the storage size of fully connected layers decreased from 260MB to 120MB after quantization.

3.2 Latency and Throughput

Inference latency is the time taken to render the result for a given query. Since, machine learning inference systems are often a part of real time applications, inference must be fast and have bounded tail latencies to provide smooth experience to the end user. While traditional machine learning models such as support vector machine and random forest are fast, most modern deep neural networks are computationally intensive and hence have substantial latencies. Further, serverless platform doesn't currently offer GPU based microvms to accelerate such compute intensive workload. SerFer, splits the computation involved in a single layer of a neural network across multiple lambda functions to achieve model level parallelism, thereby decreasing the latency. The results of the model splits are combined by a driver program which orchestrates the entire computation.

Throughput of a system is determined by the computational cost of a single query and the amount of resources available in the system. For example, the throughput of inference systems is determined by the size of the model and also the resources such as number of CPU's and GPU's in the system. Hence, higher throughput demands deploying systems with expensive hardware resources and maintaining them during idle time. To address this, we utilize the auto-scaling feature of the serverless platform to process independent queries in parallel. Further, this approach is cost effective since the user does not have to pay during idle time.

4 Architecture

We envision SerFer to consist of several Drivers under a Load Balancer (Figure 1). Each Driver orchestrates the execution of assigned inference queries by utilizing serverless platform and shared storage.

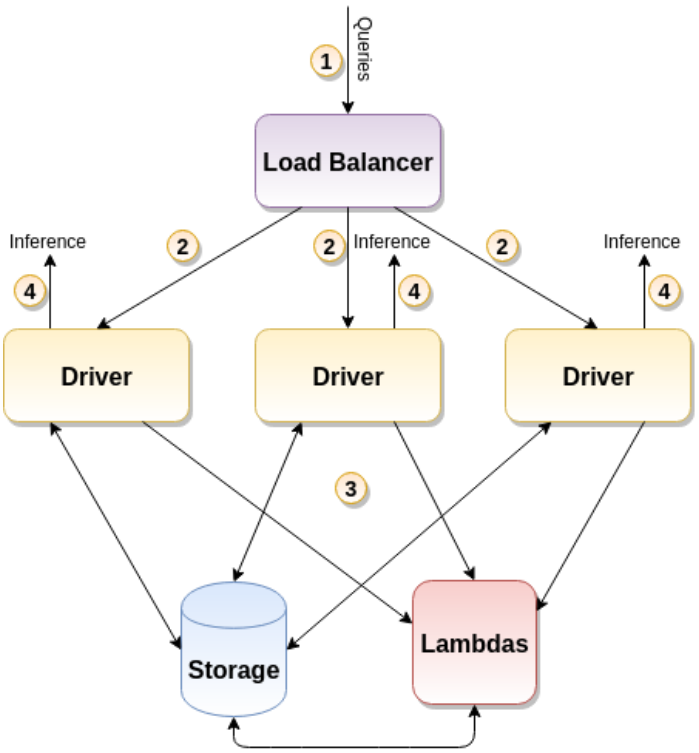

Figure 1: SerFer Architecture

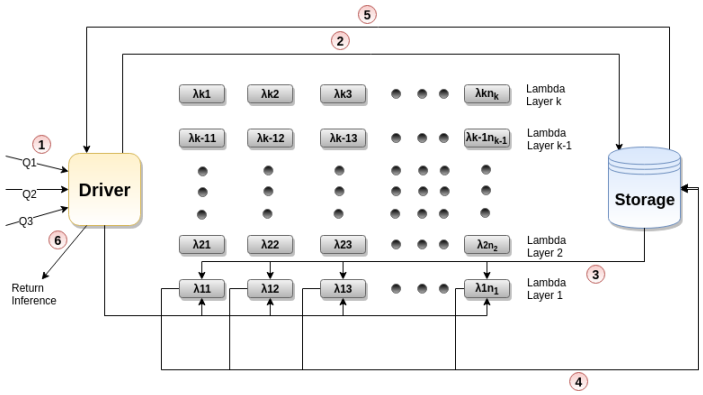

Figure 2: Driver Architecture

SerFer distributes the computations involved in an inference query across multiple lambda functions by partitioning the layers of the neural network and creating a lambda function corresponding to each partition (Figure 2). Then, an input query (image) is passed through the sequence of lambda functions, transforming it into the corresponding output. Each lambda function reads its input from a shared storage and writes its output back to the shared storage.

SerFer Architecture has the following components:

4.1 Model Splitter

Model Splitter maps the computations involved in a given Convolutional Neural Network (CNN) on to several lambda functions. Given a Deep Convolutional Neural Network with L layers, Model Splitter partitions these layers into K partitions $P_1, P_2, ..., P_K$ with $l_1, l_2, ..., l_K$ layers respectively. For each partition P_i, a corresponding lambda function F_i is created, which is responsible for the computations involved in partition P_i. If a lambda function F_i does not contain fully connected layers, then the input activation map to F_i is divided into C (currently $C = 4$) overlapping partitions and fed to C instances of lambda function F_i to achieve model level parallelism. We refer to the C instances of lambda function F_i as lambda layer L_i (Figure 2). In contrast, if a lambda function F_i consists of a fully connected layer, then a single copy of F_i would process the input activation map. In a fully connected layer, each neuron in the input is connected to all the neurons in the output. Hence, there is no simple way to efficiently map the computations involved in a fully connected layer on to multiple lambda functions.

We choose the number of neural network layers l_i in lambda function F_i based on the storage size S_i of the parameters corresponding to the l_i layers in lambda function F_i. Since, AWS limits the size of the deployment package of each lambda function to 250MB, we have the following constraint on $S_i's$

$$S_i + D < M \quad \forall i \in \{1, 2, ..., K\} \tag{1}$$

Currently, $M = 250MB$ and D is the size of the other libraries and dependencies required by the lambda function.

4.1.1 Optimizations

Since, deep convolutional neural networks with fully connected layers (e.g. AlexNet) can easily violate constraint 1, SerFer employs the following two optimizations for CNN's with fully connected layers:

1. **Quantization:** If the storage size S_i is found to violate constraint 1, Model Splitter automatically quantizes the weights corresponding to lambda function F_i to 16 bit floating point values. This reduces the storage requirement approximately by a factor of two. We prefer this technique since neural networks are robust to noise injection [2].

2. **Global Average Pooling layers:** If the storage size violates the constraint 1 even after quantization, then Model Splitter replaces the fully connected layers with Global Average Pooling layers (GAP) [10], which does not have any parameters to be stored. Such optimization decreases the accuracy by less than 2%, e.g. for AlexNet with GAP layers, the test error on ImageNet dataset increases by about 1.3% [14].

4.2 Driver

The Driver orchestrates the inference and will have the following responsibilities. Upon the arrival of a query Q (image), it splits it into C overlapping parts, uploads them to the shared storage and invokes lambda layer L_1. By invoking lambda layer L_1, we mean instantiating C copies of the lambda function F_1 to process C partitions of the input image in parallel. Then, it polls the shared storage to check if the outputs $O_1's$ (activation maps) of C copies of lambda function F_1 are available. Once the C outputs $O_1's$ are available, the driver merges them, splits them into appropriate sizes for the next lambda layer L_2 (Appendix A), uploads them to shared storage and invokes lambda layer L_2. This procedure is repeated for all

other lambda layers and the final output of lambda layer L_K is returned back to the client.

4.3 Worker

Worker is a lambda function which implements a certain number of layers of the given neural network model. Each lambda function F_i stores the weights of the CNN layers that it implements and other dependent libraries (e.g. deep learning framework used) as AWS lambda layers and imports them during runtime. Further, each lambda function reads its input from the shared storage and writes its output to the shared storage.

4.4 Storage

Since serverless functions are stateless, all the intermediate state of the computation is stored in a Storage component. In particular, each lambda layer reads its input from the shared storage and writes its output back to the shared storage. The driver writes the input image to the shared storage before invoking the first lambda layer L_1.

We don't tightly couple the system with any specific storage and aim to have flexibility to use S3, Redis, Pocket [8], Locus or any such storage or performance model beneficial for our use case.

4.5 Workflow

Using the aforementioned components, an end-to-end workflow would be as described by Figure 2.

1. Query (image) arrives at the Driver.
2. Driver splits the image into C partitions, uploads them to shared storage and invokes lambda layer L_1.
3. Lambda Layer L_1 reads its input from the shared storage.
4. Lambda Layer L_1 writes back the results of its computations to shared storage.
5. Driver reads the output of lambda layer L_1, performs merge and split operations and writes back the new splits to the shared storage. This process is repeated for all other lambda layers.
6. The Driver reads the final output of lambda layer L_K and returns it to the client.

5 Implementation

We've implemented SerFer using AWS EC2 instances for the Driver, S3 and Redis for intermediate store and AWS Lambda for serverless functions. We describe our implementation in more detail in the following subsections. Our implementation can be found in the github repository https://github.com/danish778866/serFer.

5.1 Execution Engines

Execution engine (Driver) is the component of SerFer that orchestrates the end-to-end execution of inference. We implemented polling and step functions based execution engines.

5.1.1 Poll Driver

Poll Driver is implemented in an AWS EC2 instance. The flowchart of the execution in Poll Driver is shown in Figure 3. Barrier is essentially where the polling happens, i.e. the Driver polls the Storage to check the presence of all intermediate results required for the next lambda layer. NUM_LAYERS is the total number of lambda layers present in the inference, as returned by the Model Splitter (This is referred to as K in Section 4).

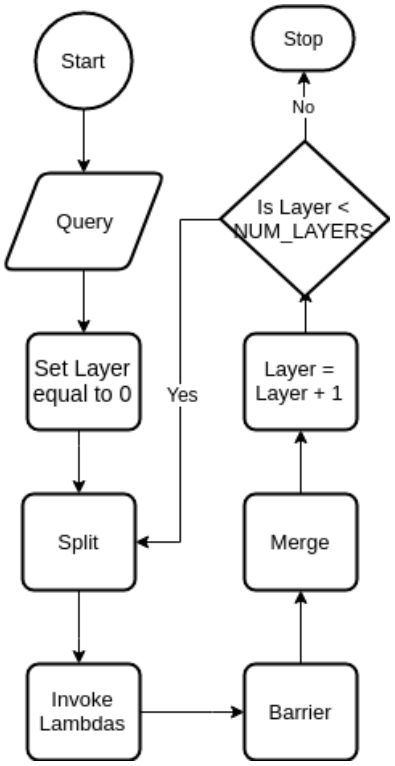

Figure 3: Flowchart for Poll Driver

5.1.2 Step Function Driver

The Poll Driver (Section 5.1.1), as the name suggests was designed to check for the completion of tasks by polling for expected output files in storage. The continuous polling may add overhead to the communication between Driver and Storage, which may cause adverse effects on latency. We, hence explored using step functions as the Driver ecosystem, with the status of tasks driving the execution model. AWS Step Functions helps string together multiple AWS services into serverless workflows. The workflows are made up of tasks, where we can use the output of one task as an input to the next. Step function automatically triggers and tracks each task, and hence removes the need for continuous polling from the Driver. In our implementation of the step function (Figure 4), the state starts with the Split Image lambda, and then

the output of the function is passed onto the next task, which is a parallel task, comprising of lambda layer L_1. Step function ensures the order of execution and allows concurrency with the parallel task Function in case of no dependencies. After the successful completion of all the parallel tasks, the state of the parallel task function transitions to Succeeded, eliminating the need to poll for the successful completion of each task. The result of each task is the key name of the input that has been stored in Redis by the preceding task. The step function is triggered by a Driver running on the EC2 instance, and the final output is returned back to the Driver, which is then available to the user.

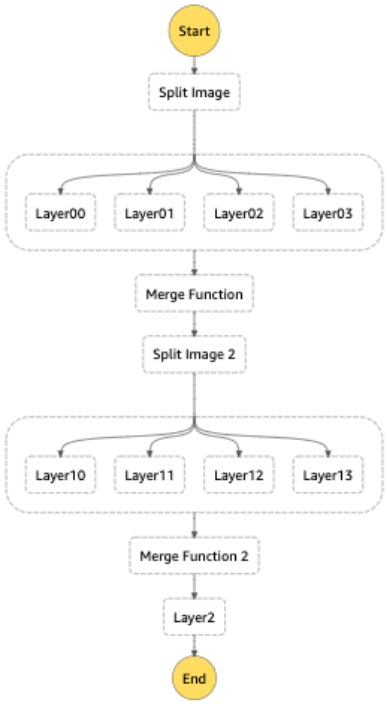

Figure 4: State Machine for Step Function Driver

5.2 Storage

Serverless functions are stateless, hence the state has to be maintained in an intermediate store. There is a lot of research going on in order to optimize the intermediate store for serverless functions (e.g., Pocket). Hence, SerFer storage has been designed to be extensible. Any Storage can be plugged into the Driver by extending the interface defined in Listing 1.

```
class SerferStorage(ABC):
    # Establish connection to the storage
    def prepare_storage(self):
    # Get connection object
    def get_storage_handle(self):
    # Write key containing data to the storage
    def write_to_store(self, key, data):
    # Read data corresponding to key from the
    storage
    def read_from_store(self, key):
    # Check if key exists in the storage
    def check_if_exists(self, key):
```

Listing 1: SerFer Storage Interface

5.2.1 S3

Amazon S3 or Amazon Simple Storage Service is a "simple storage service" offered by Amazon Web Services that provides object storage through a web service interface. Amazon S3 uses the same scalable storage infrastructure that Amazon.com uses to run its global e-commerce network. We used S3 as one of the storage backends for SerFer, implementing the interface given in Listing 1.

5.2.2 Redis

Redis is an in-memory data structure project implementing a distributed, in-memory key-value database with optional durability. Redis supports different kinds of abstract data structures, such as strings, lists, maps, sets, sorted sets, HyperLogLogs, bitmaps, streams, and spatial indexes. We used Redis as one of the storage backends for SerFer, implementing the interface given in Listing 1 (refer Section 4).

5.3 Serverless Functions

We used the serverless platform provided by AWS, i.e. AWS Lambda. Our implementation is currently restricted to AlexNet and its variants (e.g. AlexNet with Global Average Pooling layers). However, extending this to other CNN architectures would be straightforward, since it just involves automating the process of generating the lambda functions $F_i's$.

For our AlexNet implementation, we have three lambda functions, F_1, F_2 and F_3). Lambda functions F_1 and F_2 consists of convolutional layers with relu activation and max pooling layers. Hence, we instantiate four copies of F_1 and F_2 each to process a single query. Lambda function F_3 contains fully connected layers and only a single copy of F_3 is instantiated for each query.

6 Evaluation

Due to time constraints, we didn't evaluate SerFer system as envisioned in Figure 1. Instead, we evaluated the SerFer Driver as depicted in Figure 2. We don't think this should be a problem since adding multiple Drivers under a load balancer to serve given throughput and latency constraints should be straightforward, given that the Driver in itself has been evaluated individually, i.e. we're aware of the throughput/latency that a given Driver can support.

Our evaluations were targetted to answer the following questions:

1. What are the time consuming components in the SerFer Poll Driver? (Section 6.2)
2. How does the tail latency of a burst inference workload on SerFer compare to that on a GPU machine? (Section 6.3.1)
3. What is the relationship between performance of a SerFer Driver with respect to inference burst sizes? (Section 6.3.2)
4. What does the cost model look like for deploying SerFer? (Section 6.4)

6.1 Experimental Setup

AWS t2.xlarge instance with Ubuntu 18.04 image was used for running SerFer Driver. For experiments that use GPU, AWS p2.xlarge (NVIDIA Tesla K80 GPU) instance with Ubuntu 18.04 image was used. Pytorch Framework was used for serving inference queries. SerFer experiments performed are mentioned in Table 1. GPU experiments were performed on burst sizes of $8, 32, 64, 128, 256, 1000$ with a total of 16000 inference queries. These numbers were chosen to help us compare GPU serving system with SerFer.

Traffic	Driver	Number of Images	Sleep Time per burst (s)	Images per burst
Burst	Poll	1000	0	1000
Burst	Poll	500	0	500
Burst	Poll	250	0	250
Burst	Poll	100	0	100
Uniform	Poll	1000	1	1
Uniform	Poll	1000	0.2	1
Uniform	Poll	1000	0.02	1
Uniform	Step Function	1000	0.8	1
Uniform	Step Function	400	0.8	1
Uniform	Step Function	200	0.8	1
Uniform	Step Function	100	0.8	1
Uniform	Step Function	10	0.8	1

Table 1: SerFer experiments

Our experimental data should be interpreted with the following points in mind:

1. In all our experiments, there was a limit of 1000 to the number of concurrent lambda executions across all lambda functions.

2. Microbenchmark evaluations have only been presented for the burst experiments.

3. S3 as an intermediate store performed very bad as compared to Redis, for example, inference of a single query took around 13 seconds as opposed to that of 1.5 seconds using Redis. Hence, we didn't include evaluations using S3 as the intermediate store.

4. The performance of Poll Driver was superior to that of Step Function Driver. Furthermore, the Step Function Driver was experiencing few failures for burst experiments. Hence, we only evaluated Poll Driver extensively.

6.2 Microbenchmarks

The following microbenchmarks were performed for quantifying the time spent across various SerFer components:

6.2.1 Redis Operations Time

Figure 5 shows microbenchmarks for Redis read and write operations. Since polling in Poll Driver checks the existence of a given key by reading it from the store, the read time is expensive. However, 95 percentile for read is still less than a second. As can be seen by referring to Figure 12, write time is neglible as compared to the overall inference time.

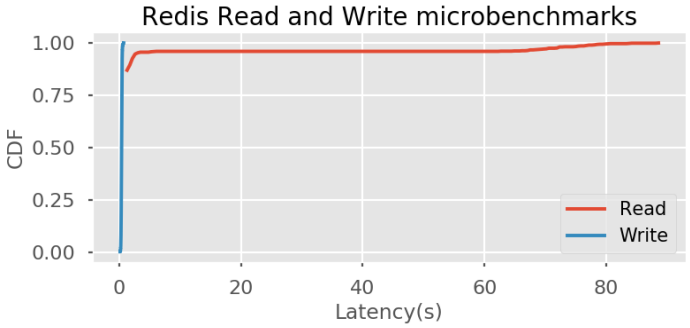

Figure 5: CDF for Redis Operations

6.2.2 Lambda Layers Time

Figure 6 shows the time required for each lambda layer. Lambda layer 3 is consuming more time and has a 95 percentile latency around 2.5 seconds. As can be seen by referring to Figure 12, the fraction of time taken by other two lambda layers is minimal as compared to the overall inference time.

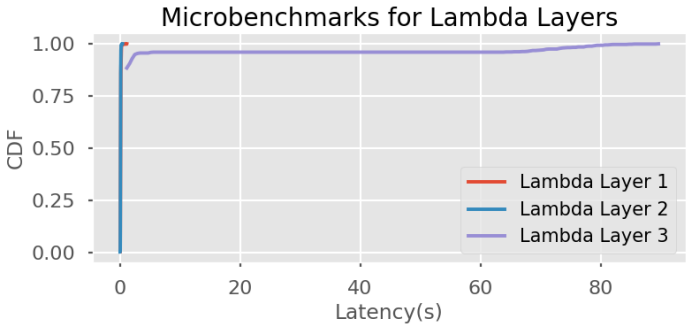

Figure 6: CDF for Lambda Layers

6.2.3 Poll Time

Figure 7 shows the time spent polling the storage for the presence of intermediate results by the Poll Driver. 95 percentile stands pretty decent at around a second.

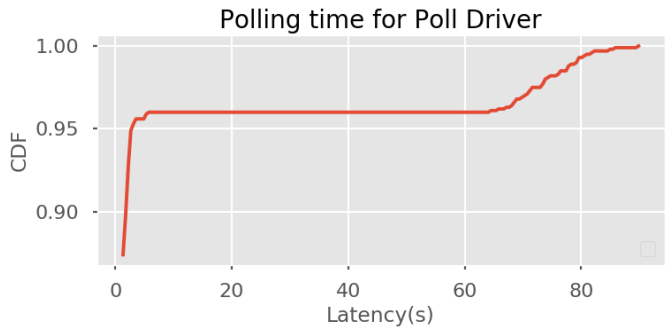

Figure 7: CDF for Polling in Poll Driver

6.2.4 Merge and Split Time

Figure 8 shows the time spent by the Poll Driver for performing merge and split operations as mentioned in the Figure 3. Split takes more time since it involves creating a copy of activation maps. However, as can be seen by referring Figure 12, both merge and split consume negligible fraction of inference time.

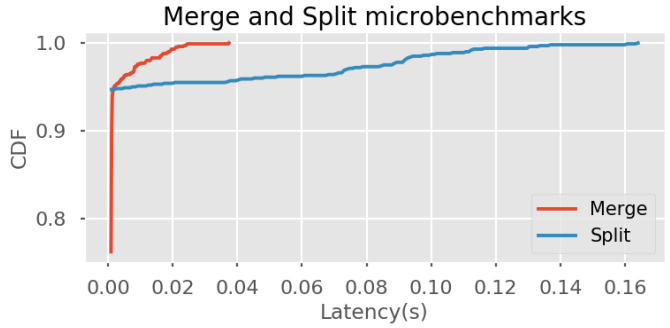

Figure 8: CDF for Merge and Split in Poll Driver

6.3 Performance

In this section, we evaluate the performance of SerFer with respect to traffic type, burst size and against GPU.

6.3.1 SerFer vs GPU

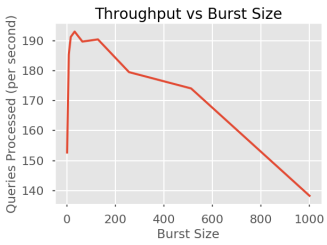

Figure 9: GPU throughput vs Burst Size

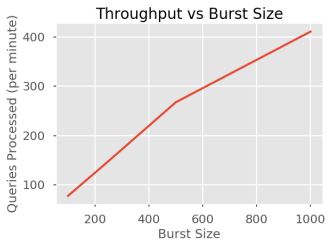

Figure 10: Poll Driver throughput vs Burst Size

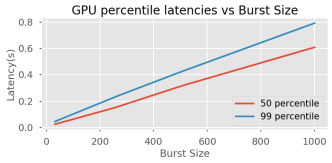

Figure 11: GPU latency vs Burst Size

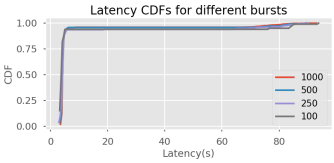

Figure 12: Poll Driver latency vs Burst Size

Figures 9 and 10 show the relationship between throughput and burst size for GPU and SerFer respectively. Even though the throughput on GPU is better, scaling characteristics of SerFer is better than that of GPU.

Figures 11 and 12 show the relationship between latency and burst size for GPU and SerFer respectively. While the latency of GPU based system increases linearly with burst size, this is not the case with SerFer. In fact, latency of SerFer system is relatively unaffected by burst size.

6.3.2 Performance vs Burst Size

Latency increases slightly with increase in the burst size (Figure 12). However, the effect is not much pronounced. 95 percentile for latencies is around 4 seconds. A few queries have extremely bad latencies. A possible reason for this is explained in Section 9.2.

Throughput increases with the increase in burst size (Figure 10), this demonstrates the scaling characteristics of SerFer, and hence that of serverless functions.

6.3.3 Uniform Traffic

Figure 13 shows the CDF for latencies of 1000 inference queries simulated using multiple uniform traffic patterns. It can be observed that as the time interval between consecutive queries decreases, the fraction of queries with higher latencies increases and almost coincides with that of burst workload with 1000 queries.

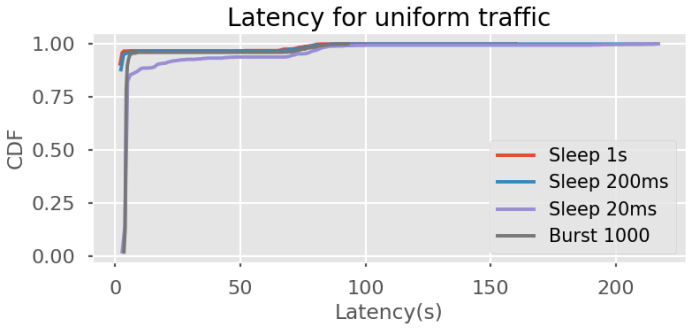

Figure 13: CDF for Latencies for uniform traffic in Poll Driver

Figure 14 shows the CDF of latencies for the Step Function Driver for different batch sizes. Step function has a limit with respect to the number of invocations per second, because of which burst requests have huge failure rates. We provided a uniform rate of incoming requests (inter-request gap of 0.8 seconds).

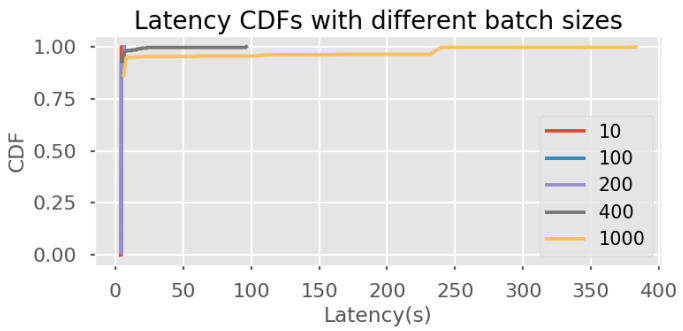

Figure 14: CDF for Latencies in Step Functions Driver

6.4 Cost

As mentioned in Section 5, SerFer divides the AlexNet model into three lambda functions. The cost of each lambda function, based on the resource requirements as given by AWS is listed in Table 2. This table also includes the average time taken by each lambda function, as measured by our microbenchmarks.

F	C	Memory (MB)	CP($)	A(ms)
1	4	512	0.000000834	124
2	4	1216	0.000001980	116
3	1	2048	0.000003334	3881

Table 2: SerFer Cost Evaluation

Where
F - Lambda Function
C - Number of input splits for lambda function
CP - Cost per 100ms
A - Average time for lambda function

The cost of each inference query would then be given by equation 2.

$$Cost = \frac{1}{100} \times \Sigma_{i=1}^{3} l_i \times CP_i \times A_i \quad (2)$$

Hence, each inference query on our setup costs 0.000142716388. Additionally, the Driver and Storage have their own respective costs. However, these costs are amortized over several inference queries. Table 3 summarizes the cost of each component of SerFer.

Component	Cost
Lambda Functions	$0.00014271638/query
Driver	$0.188/hour
Redis Cluster	$0.216/hour

Table 3: Cost of Individual Components

From Figure 10, we see that the throughput of SerFer scales linearly with burst size. Assuming uniform burst size of 1000, the number of queries that would be processed per hour would be about 24000. Therefore, total cost of the SerFer system per hour would be $3.829.

For a similar burst size, the throughput of a GPU based serving system would be 130 queries/second (refer Figure 9). Hence the cost to process 24000 queries would be $0.0508 (Currently, AWS p2.xlarge instance costs $0.9/hour).

Although, the cost of SerFer system is higher than that of GPU based serving system, this calculation only takes into consideration the processing cost on GPU and ignores idle time. If such idle time exists for GPU serving system, it would result in underutilization of expensive hardware resources. We believe that further optimizations can help improve the cost of SerFer system.

SerFer is preferable as an inference system for applications with significant idle time and which only require near real-time inference.

7 Conclusion

The adoption of serverless platforms is growing across different application domains. In this work, we demonstrated that serverless platforms shows promise for machine learning inference at scale. We proposed a system for performing inference of convolutional neural network models and presented evaluations for the performance and cost of the system. We believe that co-designing inference systems on serverless platforms along with the models it is targeted for, would enable such systems to serve the high throughput and low latency requirements of machine learning inference applications.

8 Future Work

Additional optimizations and extensions are possible in the components of SerFer. We believe that these optimizations would bring to light the capabilities of serverless platforms for inference applications.

8.1 Generalize to various model architectures

We've restricted our implementation to serving of AlexNet model. However, generalizing SerFer to work for other CNN model architectures would be straightforward. We plan to automate this process in the future. Further, extending this system to serve other architectures such as Recurrent Neural Networks (RNN) would be an interesting future direction.

8.2 Storage

As mentioned before, optimizing storage for serverless functions is an open area of research. It would be a good idea to evaluate SerFer with such storage systems (e.g., Pocket) as its storage backend. Furthermore, Redis storage used in our work isn't sharded. It might be useful to evaluate SerFer with a sharded Redis cluster as its storage backend. This way, SerFer can leverage the performance benefits offered by distributed storage in order to improve its own performance.

8.3 Caching

Yet another area of research to speed up machine learning inference involves caching the input queries and intermediate results (e.g., freeze inference). Such techniques can be used in order to improve SerFer serving.

9 Discussion

We've studied the viability of serverless platforms (specifically, AWS lambda) for performing inference at scale. While doing so, we made some interesting observations. We discuss a select few of them in the subsequent subsections.

9.1 Warm Start and Cold Start effects

AWS executes lambda functions in a secure and fast microvm called firecracker [3]. Such microvms are instantiated on demand and are kept alive for around 15 minutes after their first use, suspecting reuse of the lambda. This is known as a warm start of lambda function, i.e. a microvm is already available for execution. On the contrary, if no microvms are available when an execution of a lambda function is requested, a new microvm has to be instantiated, known as cold start of lambda function. Irrespective of however fast the instantiation of these microvms is, they still take some time to instantiate and establish dependencies. During this time, the corresponding execution has to wait, thereby hurting latency sensitive applications on startup. This shouldn't be a problem for a considerably busy application, however, such cold starts can still be avoided by using one of the following approaches:

1. The system can be warmed up using some dummy requests before serving real-time traffic.
2. Cloudwatch ping events can be configured to ping the lambda functions every once in a while, thereby keeping one microvm of each such lambda always warm. However, this still doesn't avoid cold start if multiple events arrive simultaneously thereby triggering the execution of a particular lambda function concurrently.
3. The problem with the above approach can be solved by creating a wrapper lambda which basically just invokes the required amount of concurrent lamdas for the lambda functions. The lambda function then should be written in a way that it can recognize such invocations and just do some default actions under such scenarios. However, this approach is a hack and makes it hard to reason about cost.

9.2 Nice-to-Have

AWS lambda has a limit on the number of lambdas that can be executed concurrently across all lambda functions. Once the number of concurrent executions surpass this limit, additional requests are queued at the AWS lambda scheduler. This causes a problem for SerFer in the following scenario.

1. Let's assume that 996 lambdas are executing concurrently at the moment, also lets assume that concurrent lambda execution limit is 1000.
2. Let's say that two queries, i.e. Q1 and Q2 are submitted to SerFer concurrently.
3. The Driver splits the two queries into four pieces each, lets say $q1_1, q1_2, q1_3$ and $q1_4$ for Q1 and $q2_1, q2_2, q2_3$ and $q2_4$ for Q2.
4. The Driver invokes the function F_1 (refer Section 4) for each split of input for Q1 and Q2.
5. Let's say that the functions corresponding to the splits $q1_1, q1_2, q2_1$ and $q2_2$ start executing, thereby making the number of lambda functions concurrently executing equal to 1000. The lambda functions for the splits $q1_3, q1_4, q2_3$ and $q2_4$ are hence queued at AWS lambda scheduler.
6. For each query, the Driver has to wait for the execution of lambda function on all splits before moving on to the next lambda layer. However, we don't really know the queue depth at which lambda functions corresponding to each input split of a given query will get executed.
7. This implies that the latency of such queries is worsened just because the lambda function corresponding to 1 (or possibly few) of it's input splits haven't been processed.

One way to avoid this could be to assign a priority to each lambda invocation. However, AWS Lambda doesn't support this as of now, but it would have been nice-to-have.

10 Acknowledgements

We would like to thank Prof. Akella for the providing the directions towards an interesting project, guiding us throughout the process and suggesting us improvements. We would also like to thank Arjun Balasubramanian for his valuable inputs on designing the initial SerFer architecture.

References

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[2] Sanjeev Arora et al. "Stronger Generalization Bounds for Deep Nets via a Compression Approach". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. 2018, pp. 254–263. URL: http://proceedings.mlr.press/v80/arora18b.html.

[3] Firecracker Contributors. *Firecracker*. https://aws.amazon.com/blogs/opensource/firecracker-open-source-secure-fast-microvm-serverless/.

[4] Pytorch Contributors. *Pytorch*. https://pytorch.org/.

[5] Daniel Crankshaw et al. “Clipper: A Low-latency Online Prediction Serving System”. In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI’17. Boston, MA, USA: USENIX Association, 2017, pp. 613–627. ISBN: 978-1-931971-37-9. URL: http://dl.acm.org/citation.cfm?id=3154630.3154681.

[6] Sadjad Fouladi et al. “Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 363–376. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[7] Eric Jonas et al. “Occupy the Cloud: Distributed Computing for the 99%”. In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC ’17. Santa Clara, California: ACM, 2017, pp. 445–451. ISBN: 978-1-4503-5028-0. DOI: 10.1145/3127479.3128601. URL: http://doi.acm.org/10.1145/3127479.3128601.

[8] Ana Klimovic et al. “Pocket: Elastic Ephemeral Storage for Serverless Analytics”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 427–444. ISBN: 978-1-931971-47-8. URL: https://www.usenix.org/conference/osdi18/presentation/klimovic.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[10] Shuicheng Yan Min Lin Qiang Chen. “Network in Network”. In: (2013). URL: https://arxiv.org/abs/1312.4400.

[11] Christopher Olston et al. “TensorFlow-Serving: Flexible, High-Performance ML Serving”. In: *Workshop on ML Systems at NIPS 2017*. 2017.

[12] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: (2017).

[13] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. “Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure”. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, 2019, pp. 193–206. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/pu.

[14] B. Zhou et al. “Learning Deep Features for Discriminative Localization”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 2921–2929. DOI: 10.1109/CVPR.2016.319.

Appendices

A Merge and Split Logic

In this section, we describe how the Driver merges the activation output of lambda layer L_i and splits them before feeding them to lambda layer L_{i+1}. We assume number of partitions $C = 4$. For simplicity, we further assume that each lambda function contains computation corresponding to one neural network layer only, i.e. $l_i = 1 \quad \forall i \in 1, 2, ...K$.

For convolutional and pooling layers, the size of the input activation volume $(N, C_{in}, H_{in}, W_{in})$ and output activation volume $(N, C_{out}, H_{out}, W_{out})$ are related as give by equations 3 and 4.

$$H_{out} = \frac{H_{in} + 2 \times padding[0] - dilation[0] \times (kernelsize[0] - 1) - 1}{stride[0]} + 1 \tag{3}$$

$$W_{out} = \frac{W_{in} + 2 \times padding[1] - dilation[1] \times (kernelsize[1] - 1) - 1}{stride[1]} + 1 \tag{4}$$

Assume (C_i, H_i, W_i) and $(C_{i+1}, H_{i+1}, W_{i+1})$ to be the size of the input and output activation volumes of lambda layer L_i. Then, the driver splits the input volume into 4 overlapping volumes each of size $(C_i, \frac{H_i}{2} + \Delta_H, \frac{W_i}{2} + \Delta_W)$. Here, Δ_H and Δ_W are chosen such that the size of the output of each copy of lambda function F_i is equal to $(C_{i+1}, \frac{H_{i+1}}{2}, \frac{W_{i+1}}{2})$. The value of Δ_H and Δ_W that satisfy this constraint are given by equations 5 and 6.

$$\Delta_H = \frac{kernelsize[0] - 1}{2} \tag{5}$$

$$\Delta_W = \frac{kernelsize[1] - 1}{2} \tag{6}$$

Since, our actual implementation contains more than one layer in each lambda function, we repeatedly apply equations 5 and 6 to all the layers to get the values of Δ_H and Δ_W. Merging involves concatenating the activation outputs of each of the 4 copies of lambda function F_i.