Gillis: Serving Large Neural Networks in Serverless Functions with Automatic Model Partitioning

Minchen Yu*, Zhifeng Jiang*, Hok Chun Ng*, Wei Wang*, Ruichuan Chen†, Bo Li*
*Hong Kong University of Science and Technology
{myuaj, zjiangaj, hcngac, weiwa, bli}@cse.ust.hk
†Nokia Bell Labs
ruichuan.chen@nokia-bell-labs.com

***Abstract*—The increased use of deep neural networks has stimulated the growing demand for cloud-based model serving platforms. Serverless computing offers a simplified solution: users deploy models as serverless functions and let the platform handle provisioning and scaling. However, serverless functions have constrained resources in CPU and memory, making them *inefficient* or *infeasible* to serve large neural networks—which have become increasingly popular. In this paper, we present Gillis, a serverless-based model serving system that automatically partitions a large model across multiple serverless functions for faster inference and reduced memory footprint per function. Gillis employs two novel model partitioning algorithms that respectively achieves latency-optimal serving and cost-optimal serving with SLO compliance. We have implemented Gillis on three serverless platforms—AWS Lambda, Google Cloud Functions and KNIX—with MXNet as the serving backend. Experimental evaluations against popular models show that Gillis supports serving very large neural networks, reduces the inference latency substantially, and meets various SLOs with a low serving cost.**

***Index Terms*—machine learning inference; serverless computing; reinforcement learning**

I. Introduction

Machine learning models, especially deep neural networks (DNNs), are increasingly trained and published in the cloud to provide inference services for dynamic queries [1]–[4]. As inference is performed in real time with stringent SLOs (Service-Level Objectives), the model serving platforms must be made scalable to the changing workloads. This can be achieved by augmenting the traditional VM-based model serving with *serverless functions*: users simply deploy models as cloud functions and let the serverless platform handle provisioning and scaling; users pay only for the resources used when the functions are running. Serverless functions have hence been exploited, together with the conventional VMs, to build scalable, pay-as-you-use inference serving platforms in the public cloud [3], [5]–[9].

Unlike VMs, serverless functions have *constrained resources* in CPU and memory [10]–[12]. As very large DNNs are increasingly used for improved accuracy [13]–[15], using serverless functions to serve those models is *inefficient* or simply becomes *infeasible* with out-of-memory (OOM) errors.

There are mainly two approaches to addressing this problem. One is to reduce the model size using model compression techniques [16], [17]. However, this approach sacrifices the model accuracy and requires careful tuning to minimize the loss [13]. A better approach is *model partitioning*, which partitions a large model into smaller components for *parallel execution*, without accuracy loss. Model partitioning was originally proposed to train large DNNs on GPUs [13], [18], [19] and has also been used for model inferencing on edge devices [20], [21].

However, existing model partitioning techniques do not apply to serverless functions. First, cloud-based model serving has restrictive latency SLOs and is sensitive to the serving costs—none of these requirements are mandated in GPU training and edge-based serving. Second, unlike GPUs or edge devices, serverless functions are *stateless* and their communications are usually performed through external storage with limited network bandwidth [22], [23]. Yet, existing model partitioning solutions presume addressable devices with high-speed interconnections to facilitate direct data transfer. These solutions cannot be used to coordinate multiple serverless functions and would result in significant communication overhead.

In this paper, we propose Gillis[1], a serverless-based model serving system that *automatically* explores parallel executions of DNN inference across multiple functions for faster inference and reduced per-function memory footprint. Our design follows the *fork-join* computing model: upon receiving an inference request, a *master* function is invoked to initiate multiple *worker* functions, each hosting a partition of the model. The master interacts with workers through stateless connections (function invocations). To reduce communications between master and workers, Gillis performs *coarse-grained* model partitioning. It fuses multiple consecutive layers of a DNN model into a single *layer group*. The model can have multiple layer groups. Gillis partitions each layer group for parallel execution. Such coarse-grained partitioning allows a function to compute all layers in a group *locally*, hence avoiding frequent synchronizations with the other functions.

To determine which layers are grouped and how a layer group is partitioned, we design novel network partitioning algorithms for two common scenarios: (1) minimizing the inference latency (*latency-optimal*), and (2) minimizing the serving cost while complying with the user-specified latency

[1]We name our system after Gillis Lundgren, designer of IKEA who popularized the idea of dissembling a bulky furniture into small parts and "flat-packing" them for storage saving and ease of shipping.

SLOs (*SLO-aware*). For latency-optimal partitioning, we propose a dynamic programming algorithm to efficiently search for the optimal strategy. For SLO-aware partitioning, finding the optimal solution requires searching all feasible network parallelization schemes that meet the latency SLOs. We hence propose to learn the optimal strategy using *reinforcement learning*. We encode the partitioning policy into a neural network and train it with extensive simulated experiments, in which we partition the serving model, observe the resultant cost and inference latency, and iteratively refine the policy.

We have implemented Gillis on AWS Lambda [10], Google Cloud Functions [11] and KNIX [24], [25] with MXNet [26] as the serving framework. Gillis supports models in standard open exchange format ONNX [27]. We evaluate Gillis against various popular DNN models. In the latency-optimal mode, Gillis achieves up to $3\times$ inference speedup over non-parallelization when the models can fit into a single function; for larger models, the speedup is up to $9.2\times$ over the pipelined execution where a single function sequentially loads layers from S3. In the SLO-aware mode, Gillis consistently meets various latency SLOs while minimizing the serving cost with up to $1.8\times$ savings than the Bayesian optimization-based solution. Gillis is open-sourced for public access.[2]

II. Background and Motivation

In this section, we motivate the benefits of serverless-based inference serving (§II-A) and illustrate its inefficiency of serving large DNNs (§II-B). We describe prior efforts in handling large models on resource-constrained devices (§II-C) and discuss the challenges of doing so in serverless functions (§II-D).

A. Serverless-based Inference Serving

Inference serving is conventionally provisioned on VMs with high-performance CPUs and GPU accelerators [1], [2], [4]. VM-based inference serving delivers cost-effective high performance for running *stable* workloads. However, it becomes *inefficient* in handling dynamic queries: as VM instances have long startup latencies (e.g., several minutes), a high margin of over-provisioning is usually needed to accommodate unexpected load spikes, e.g., SageMaker recommends setting an over-provisioning factor of 2 [28].

Compared with VMs, serverless functions have a far shorter startup latency (e.g., 10s ms if warm-started) and can quickly scale out to a large number of instances to accommodate the surging inference requests in a short period of time. On the other hand, serverless functions are not well suited to serve stable workloads, as they have a high price per request [3], [23]. Therefore, one appealing approach is to augment VMs with serverless functions [3], [5], [6], that is, using VMs to handle stable inference requests while using serverless functions to cover transient load bursts. In fact, leading cloud providers like AWS and Google have advocated inference serving as an important use case of their serverless offerings [5], [6]. Many recent works have also exploited serverless functions to deploy neural networks. For example, Fotouhi et al. [8] explored various architectures to deploy NLP (Natural Language Processing) applications on the serverless cloud; MArk [3] uses serverless functions to handle sporadic inference load spikes to avoid over-provisioning VM servers; Barista [29] uses predictive scaling to achieve low-latency inference serving in the serverless cloud.

[2]https://github.com/MincYu/gillis-open-source.

B. Inefficiency of Serving Large Models

The deep learning community is building increasingly larger deep neural networks to achieve higher prediction accuracy for various practical applications such as image classification [14], [30] and language modeling [15]. This trend, driven by the advances of hardware accelerators and the rapid growth of training data, is predicted to continue in the future [13], [31].

Current serverless offerings, on the other hand, only support functions with constrained resources. In Google Cloud Functions, a function instance can access to only 4GB memory with limited CPU cycles [32]. The network bandwidth is also limited, around 300Mbps for running a single function and even lower when running more [22]. Other serverless platforms have the similar constraints [10], [12]. Using resource-constrained functions to serve very large models is problematic—it either results in an exceedingly long inference latency due to limited CPU cores, or simply becomes infeasible when the model is too large to fit into the memory of a function.

Serving Large Models We illustrate these problems in both Google Cloud Functions and AWS Lambda with WResNet [14]. WResNet "widens" the classical ResNet [33] model by increasing the number of both the input and output channels of a weight tensor on each convolution layer. The width of WResNet is determined by a *widening scalar* k, where $k = 1$ means no widening. Increasing the widening scalar improves the model accuracy, but also enlarges the model quadratically. Following the benchmark settings in [13], we experiment WResNet with widening scalar changing from 1 to 5 on ResNet-50. We deploy these models on Google Cloud Functions and AWS Lambda with their respective maximum possible instance memories at the time of experiments. For each model, we run the inference in the hosting function 100 times after warming it up.

Fig. 1 shows that the inference latency increases almost quadratically as the WResNet model grows wider and larger. In particular, the inference requests take over 2000 ms to complete on AWS Lambda and Google Cloud Functions with widening scalar 3 and 4, respectively; further widening the model exceeds the function's memory limit, causing an out-of-memory (OOM) error.

C. Prior Arts in Handling Large Models

Many works have been proposed recently to handle large models in resource-constrained environments [13], [16]–[21], [34]. These works fall into two approaches: model compression and model partitioning.

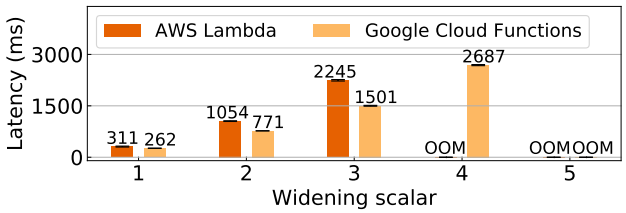
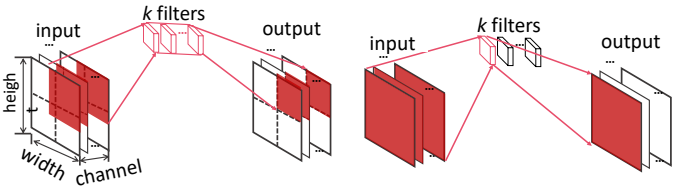


Fig. 1: The inference latencies of WResNet-50 on Google Cloud Functions and AWS Lambda.



(a) Partitioning the output tensor by height and width.

(b) Partitioning the output tensor by channel.

Fig. 2: An illustration of tensor partitioning along different dimensions in a convolution layer.

Model Compression reduces the network parameters of a large DNN and creates a significantly smaller network that can run on a resource-constrained device (e.g., mobile and edge devices). Popular model compression techniques include network pruning [16] and weight quantization [17]. However, this approach usually results in reduced accuracy and requires careful model tuning or even retraining to minimize the accuracy loss, which is too laboring for developers [13].

Model Partitioning Another promising approach is to divide a large neural network into multiple small partitions and run them in parallel [13], [18]–[21], [34], [35]. Compared with compression, model partitioning sacrifices no accuracy while accelerating the computation and reducing the per-partition resource footprint. We therefore choose it over model compression in our design.

The key technique of this approach is *tensor partitioning*. In a DNN model, each layer takes an input tensor and computes an output tensor. An output tensor typically has *multiple dimensions* and can be parallelized in many different ways along those dimensions. Take the convolution layer as an example. A convolution layer with k filters generates output feature maps with k channels. Such feature maps have three dimensions: height, width, and channel (Fig. 2a). Partitioning the output along any of the three dimensions can parallelize a convolution layer. As shown in Fig. 2a, a subset of the output feature maps along height and width depends on a partition of the input and all k filters. We can therefore parallelize the computation of the output by partitioning the input tensor along height and width across multiple workers, each keeping k filters. Alternatively, the partitioning can be along channels. Fig. 2b shows that an output channel can be computed with a single filter applied to the entire input tensor. We can hence parallelize the computation of channels across workers, each keeping a subset of filters.

Previous works use tensor partitioning to parallelize network layers across memory-constrained GPU devices for accelerated model training [13], [18], [19], [34], [35]. Similar approaches have also been used to partition large DNNs across multiple edge devices for distributed model inferencing [20], [21]. However, applying model partitioning to serverless functions poses the following three challenges.

D. Challenges

Coordinating Serverless Functions Model partitioning requires coordinating multiple workers to synchronize their computations. Such coordination is straightforward when the workers are addressable devices with direct interconnections (e.g., GPUs or edge devices). However, serverless functions are usually *unaddressable* and cannot be reached by others via stateful connections. Therefore, how to efficiently coordinate serverless functions in DNN parallelization becomes a challenge.

Reducing Communication Overhead Parallelizing a DNN layer requires transferring the input tensors to multiple workers. As serverless functions have limited network bandwidth [22], [23], parallelizing multiple layers results in a significant communication overhead that may undermine the benefits of tensor partitioning. Therefore, how to reduce the communication between functions poses another challenge.

SLO Compliance and Cost Minimization Unlike model training on GPUs and model serving at edges, serverless-based inference serving needs to comply with the latency SLOs while minimizing the serving cost under the pay-per-use billing. Achieving these two goals complicates the model partitioning algorithms, which is still an open problem.

III. Gillis Overview

In this section, we describe Gillis, a serverless-based inference serving system that automatically parallelizes the execution of large DNNs across multiple serverless functions for faster inference and reduced memory footprint per function. We developed Gillis in three platforms: AWS Lambda [10], Google Cloud Functions [11] and KNIX [24], [25]—an open-source serverless platform with the state-of-the-art performance. Compared with the former two platforms, KNIX improves function interactions with compute-collocated storage (e.g., Redis), allowing Gillis to achieve a substantial latency improvement (§V-B). Gillis uses MXNet [26], [36] as the serving framework, and supports DNN models in standard open exchange format ONNX [27]. To our knowledge, Gillis is the first system that supports very large models in the serverless cloud.

A. Workflow

Fig. 3 gives a workflow overview of Gillis. Gillis starts with the *runtime profiling*. For each type of DNN layer, Gillis profiles its execution time in a single function. Gillis also profiles the function communication latency. Based on the profiling results, Gillis builds a *performance model* and uses it to predict the model execution time under various

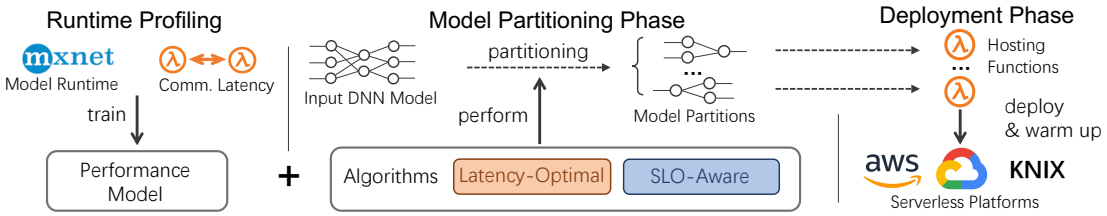


Fig. 3: A workflow overview of Gillis.

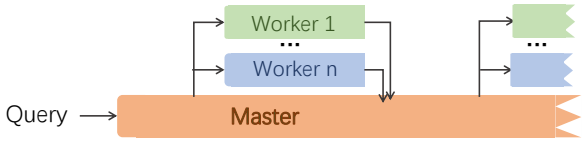


Fig. 4: The fork-join model for function coordination.

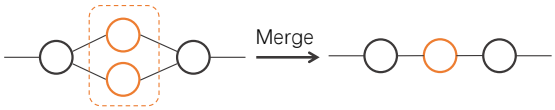


Fig. 5: An illustration of branch merging, where two parallel branch modules are merged into one layer.

parallelization schemes (§IV-A). In the *model partitioning phase*, Gillis accepts a serving model and generates a parallelization scheme to achieve the optimal inference latency (*latency-optimal*, §IV-B) or the minimum serving cost with SLO compliance (*SLO-aware*, §IV-C), using the performance model as a guideline. Gillis automatically partitions the model following that scheme. It then proceeds to the *deployment phase*, where the model partitions are packaged into functions and deployed on serverless platforms. The serving function usually takes a longer time to complete for the first invocation (cold start). To mitigate this issue, Gillis supports periodically warming up functions by sending concurrent pings to the serverless platform [37]. As function instances stay active for a long time [38], the warm-up cost can be amortized by serving numerous inference queries and is hence negligible [3].

Gillis adopts three key designs to address the challenges described in §II-D. First, it uses the *fork-join* computing model to efficiently coordinate multiple functions. Second, Gillis chooses to perform *coarse-grained* model partitioning to reduce the communication cost. Third, Gillis employs two novel partitioning algorithms that respectively achieve latency-optimal serving and cost-optimal serving with SLO compliance in the serverless cloud. We next describe the first two designs and defer the algorithms to the next section.

B. Function Coordination with Fork-Join

Fig. 4 illustrates the fork-join model that Gillis uses to coordinate multiple serverless functions. A *master* function is triggered to run upon receiving an inference query. Following the computed partitioning scheme, the master asynchronously invokes multiple *worker* functions. Each worker computes a partition of the served model, returns the result to the master, and ends its execution. The master can also help to compute a partition if having sufficient memory, which can result in fewer workers and less cost. The master assembles the returned results from all workers into a full tensor, and may initiate more workers to continue parallelizing model execution. The fork-join process may take multiple rounds to complete, with the final inference result given by the master function.

This design provides two benefits. First, in the fork-join model, function interactions are restricted between the master and a worker through REST APIs (i.e., function invocations). In comparison, existing model parallelization schemes [13], [19], [20] require direct data transfer across multiple workers through stateful connections, which is not generally supported in the serverless platforms. Second, unlike many serverless-based parallel systems that use external VMs (rendezvous server) to coordinate function instances [39], Gillis has no serverful component, making it highly scalable to dynamic workload.

C. Coarse-Grained Parallelization

Parallelizing individual layers across multiple devices using different strategies, known as *layer-wise* parallelization, yields the optimal performance, when the network is not a bottleneck [13], [18], [19]. However, this is not the case in our problem as serverless functions have limited network bandwidth. To reduce the communication overhead, Gillis instead performs *coarse-grained* parallelization: it combines multiple consecutive layers into a single *group* and parallelizes each group across serverless functions. All layers in a group are hence computed locally within a function—only the input tensor of the first layer and the output of the last in this group need to be transferred, which significantly reduces the communication overhead.

A similar idea is also used to reduce cross-device communications in edge-based model serving [20], [21], where multiple convolution layers are fused into a single block for parallelization. We extend this idea in two ways.

First, our layer grouping is not limited to convolution layers, but applies to all. This enables more parallelization opportunities, yet significantly increases the search space for optimal grouping as a DNN model can have a large number of layers. To address this problem, we propose to merge consecutive *element-wise layers* (e.g., ReLU) and *branch modules*, if any. The former require less computations and can be flexibly parallelized along any dimensions. We hence merge them into the preceding weight-intensive layers (e.g., convolution). In case that a DNN model has a branch structure

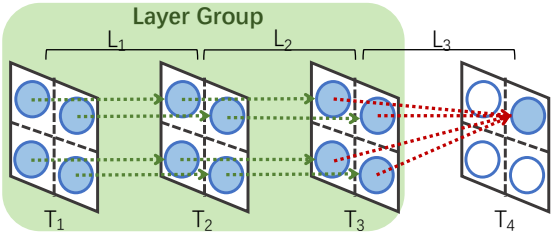


Fig. 6: An example of layer grouping based on tensor dependencies. The first two layers L_1 and L_2 can be group parallelized as they have a local response to the input along width and height; L_3 cannot be grouped with them as its output element depends on the entire input.

(e.g., Inception module [40] and Residual block [33]), we merge parallel branches into a single layer as shown in Fig. 5. Branch merging transforms a complex computation graph of a DNN model into a linear graph, substantially simplifying the partitioning strategy. In Gillis, layer merging is performed before layer grouping and parallelization. We hence do not differentiate between the original layers and the merged ones.

Second, in Gillis, worker functions do not communicate with others. This poses a new requirement to layer grouping and parallelization—group partitions must be computed *independently*. To meet this requirement, we determine if two consecutive layers can be grouped based on the *dependency* of their input and output tensors. Specifically, given two layers, if their output tensors have a *local response* to the input along the *same dimensions*, they can be group-parallelized along those dimensions. Fig. 6 gives an example, where the first two layers L_1 and L_2 can be grouped as each element of their output is locally dependent on a single input element at the same position. The layer group can hence be parallelized across four worker functions.

While layer grouping reduces the communication overhead, grouping too many layers can be inefficient, especially for those with convolution-like operators. As these operators (e.g., convolution and pooling) map multiple input elements to a single output, parallelizing the output tensor results in an *overlap* in the input partitions (Fig. 2a). As more layers are grouped, more overlaps are added, causing more redundant computations in the intermediate layers. Also, as the layer group grows larger, its partition may not fit into the memory of a single function.

Parallelizing a layer group across too many functions can also be inefficient, as it may incur significant synchronization overhead in function communications, undermining the benefits of parallelization. Fig. 7 shows an example (see §V for the experimental settings). As the number of parallel functions increases, the time spent on function communications and synchronizations grows, eventually dominating the end-to-end latency. Specifically in Lambda, increasing the number of parallel functions from 8 to 16 does more harm than good.

Therefore, a desirable grouping and parallelization strategy should strike a good balance between the computation cost and communication and synchronization overheads under the memory constraint of serverless functions.

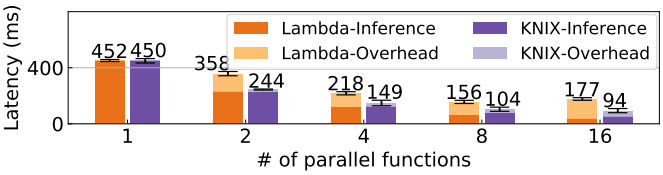


Fig. 7: The latency performance of parallelizing the first four convolution layers of VGG-16 with a varying number of functions in Lambda and KNIX. The end-to-end latency is broken down into the inference time (Inference) and the function communication and synchronization overheads (Overhead).

IV. Model Partitioning

In this section, we describe two layer grouping and parallelization algorithms that respectively achieve (1) the optimal latency and (2) the minimum cost with SLO compliance. Both algorithms use a performance model to guide partitioning, which we introduce first.

A. Performance Model

Searching for the optimal partitioning requires evaluating the latency and cost performance of various DNN parallelization schemes. Gillis builds a performance model to make such predictions based on the profiling results of model runtime and function communication delay.

Model Runtime A DNN model is a stack of convolution, normalization, activation, pooling, and fully connected layers. For each type of layer, we run it with various configurations (e.g., the filters and the shape of the output tensor in a convolution layer) in a single function, profiles the execution time, and build a regression model for prediction. Given a DNN, we infer its runtime by summing up all the predicted layer execution time.

Function Communication Delay The end-to-end latency also includes the function communication delay. We profile it by transferring data of varying sizes through REST APIs. Recall that in the fork-join model, the master function initiates multiple workers, and the communication delay depends on the slowest connection. This is equivalent to predicting the maximum delay of n concurrent communications. Our extensive measurements in AWS Lambda show that function communication delays follow an exponentially modified Gaussian distribution. We hence use the n^{th} order statistics [41] to predict the maximum delay of communicating with n workers.

We will show in §V-E that our performance model can accurately predict the DNN execution time and the function communication delays, which we use to guide the search of optimal partitioning.

B. Latency-Optimal Partitioning

We now present our first model partitioning algorithm. Our goal is to minimize the inference latency. We start with the problem formulation followed by a dynamic programming-based optimization algorithm.

Problem Formulation Consider a model G whose compute graph has n layers $l_1, l_2, \cdots, l_n$. We apply a layer grouping strategy S that fuses the n layers into multiple layer groups $S(G)$. Let $l_{(i,j)} \in S(G)$ be such a group obtained by fusing consecutive layers from l_i to l_j. Following the fork-join model (§III-B), we parallelize each layer group across worker and master functions with memory size M. Unlike a worker that only computes a partition of one group, the master runs for a longer time and could compute the partitions of *multiple groups* along the way (Fig. 4). As the computed partitions are maintained in memory, the master may not have sufficient memory to compute *all groups*. It hence needs to judiciously budget its memory for computing each group. Suppose for each group $l_{(i,j)}$, the master allocates the memory of size $M_{l_{(i,j)}}$ for its computation. Under that allocation, the master *optimally parallelizes* $l_{(i,j)}$ across multiple workers and finishes the computation in time $t(l_{(i,j)}, M_{l_{(i,j)}})$, which includes both the layer execution time and the function communication delay. Adding the computation times of all layer groups, we obtain the total inference latency under layer grouping strategy S:

$$T^S(G) = \sum_{l_{(i,j)} \in S(G)} t(l_{(i,j)}, M_{l_{(i,j)}}), \tag{1}$$

where the memory allocations must not exceed the memory size, i.e.,

$$\sum_{l_{(i,j)} \in S(G)} M_{l_{(i,j)}} \leq M. \tag{2}$$

Our goal is to find the optimal layer grouping and parallelization strategy that minimizes the latency, i.e., $\min_S T^S(G)$.

Dynamic Programming We search for the optimal strategy using a dynamic programming-based algorithm. Let $L(i, j, m)$ be the optimal latency of executing the model's consecutive layers from l_i to l_j, with memory footprint no more than m in the master function. The optimal latency of executing the entire model is given by $L(1, n, M)$, which can be recursively computed using dynamic programming as follows:

$$L(i,j,m) = \begin{cases} 0 & i > j, \\ \min\limits_{\substack{i \leq k \leq j \\ 0 \leq b \leq m}} L(i, k-1, m-b) + t(l_{(k,j)}, b) & i \leq j. \end{cases}$$

That is, we optimally compute a varying number of beginning layers (l_i to l_{k-1}) under various memory budget ($m-b$), while grouping the remainders ($l_{(k,j)}$) for parallel execution. We take the minimum latency as $L(i, j, m)$.

Parallelizing a Layer Group Our dynamic programming algorithm requires to know $t(l_{(k,j)}, b)$, the latency of optimally parallelizing a layer group $l_{(k,j)}$ with memory budget b in the master. As shown in Algorithm 1, we predict the latency using the performance model described in §IV-A. In a nutshell, given a layer group, we search over all feasible parallelization options based on the tensor dependencies (§III-C). For each option, we determine if the parallelization involves the master by checking the size of the computed group partition and the allocated memory budget in master. We predict the execution latencies for all options and return the minimum. As a layer group has only a limited number of parallelization options, the search space is relatively small.

Algorithm 1 Finding the optimal latency of a layer group

```
– l_(k,j): a layer group obtained by fusing layers from l_k to l_j
– b: the memory budget for computing l_(k,j) in the master
– M: the memory size of a function
function FindOptLatency
    O ← all feasible parallelization options of layer group l_(k,j)
    t(l_(k,j), b) ← ∞
    for all option o ∈ O do
        s ← the partition size given by parallelization option o
        τ ← ∞
        if s > M then                          ▷ partition too large to fit into a function
            Continue
        else if s > b then                     ▷ size exceeds the master's budget
            τ ← the latency of worker-only parallel execution with o
        else                                   ▷ size is contained within the master's budget
            τ ← the latency of parallelization across both the master and workers with o
        if τ < t(l_(k,j), b) then
            t(l_(k,j), b) ← τ
    return t(l_(k,j), b)
```

C. Minimizing Cost with SLO Compliance

While minimizing the inference latency is desirable, it is not always necessary. In many cases, cloud users have targeted latency SLOs—as long as the SLOs are met, they are more concerned with *minimizing the inference cost*. We therefore propose a learning-based algorithm to achieve this goal.

1) Problem Definition

Inference Cost Commercial serverless offerings charge users based on *the number of requests* for function invocations and the *duration* for function execution, rounded up to the nearest 100ms [11] or 1ms [10]. In Gillis, the cost for an inference query depends on the number of parallel functions invoked and their durations. As the former incurs *negligibly small* charges in our settings, we simply use the latter to measure the inference cost. More specifically, given a DNN model G and a partitioning strategy S, let $T_w^S(G)$ be the duration of a worker function w and $T^S(G)$ the duration of the master, which is also the inference latency. The *billed function duration* is given by

$$C^S(G) = \left(\lceil T^S(G)/D \rceil + \textstyle\sum_w \lceil T_w^S(G)/D \rceil\right) \times D\text{ms},$$

where D denotes the billing granularity of the concerned serverless platform (i.e., 1 or 100).

Latency SLO We define the SLO for the mean latency with *response-time threshold* T_{max}. That is, the mean inference latency should not exceed T_{max}. We notice that the support of tail-latency requirement would be more meaningful, which we will discuss in §VI and leave for future work.

Problem Formulation Given a DNN model G and the latency SLO T_{max}, our goal is to find the optimal partitioning strategy that minimizes the inference cost with SLO compliance. Specifically, let $L^S(G)$ be the mean inference latency under strategy S. We solve the following optimization problem:

$$\begin{aligned} \min_S \quad & C^S(G), \\ \text{s.t.} \quad & L^S(G) \leq T_{\text{max}}. \end{aligned} \tag{3}$$

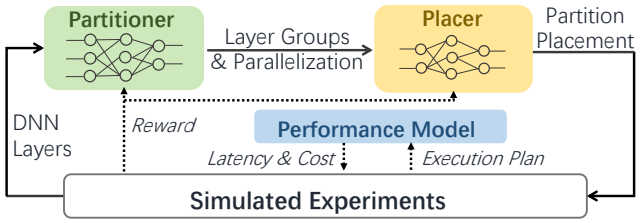


Fig. 8: An overview of RL-based model partitioning.

2) Challenges and Potential Solutions

Minimizing the inference cost with SLO compliance appears more challenging than optimizing the latency (§IV-B). Unlike the latter, the optimization problem (3) cannot be recursively broken down into simpler sub-problems, violating the fundamental structural requirement for dynamic programming-based solutions.

In general, optimally solving problem (3) requires searching over all possible partitioning schemes that meet the latency SLOs, which does not scale to large DNNs. We therefore consider two popular learning-based algorithms: Bayesian optimization (BO) and reinforcement learning (RL).

BO has been widely used in finding the optimal configurations of cloud applications [42]. It is well-suited for optimizing black-box systems that have no clear performance model. This is not the case in our problem, where the objective—the billed function duration—is well defined and can be easily obtained using the performance model (§IV-A).

RL, on the other hand, appears a good fit to our problem, where we encode the partitioning strategy into a neural network and train it with extensive simulated experiments: the RL agent makes a partitioning decision, predicts the resultant cost and latency using the performance model, and iteratively refines the strategy. In fact, RL has been successfully applied to a similar device placement problem for training large DNNs [34], [35], [43]. We hence choose it over BO as our solution. We will justify our choice empirically in §V-C.

3) RL-Based Model Partitioning

Overview We design a hierarchical RL model to learn the optimal partitioning strategy. Fig. 8 gives an overview of our solution. Our RL model has two agents, *partitioner* and *placer*, each of which is a two-layer neural network. The partitioner takes as input the DNN layers and determines how these layers are fused into groups and how each group is parallelized. Given the layer groups, the placer determines how partitions are placed on the master and workers, which works out a detailed function execution plan. In simulated experiments, we can obtain the latency and cost of an execution plan using the performance model (§IV-A), then jointly train the partitioner and the placer with *policy gradients* [44]. We next elaborate the training details.

Reward Function A typical RL training process consists of multiple episodes. In each episode, the agent makes a partitioning attempt, evaluates its performance using a *reward function*, and iteratively refines the strategy in the following episodes. We define the reward function that accounts for both the inference cost and latency. Intuitively, only when the latency SLO is satisfied can a *positive* reward be made, where a shorter billed duration leads to a higher reward. We therefore evaluate strategy S using the following reward function:

$$R^S(G) = \begin{cases} B - C^S(G) & L^S(G) \le T_{\max}, \\ T_{\max} - L^S(G) & \text{otherwise.} \end{cases} \quad (4)$$

Here, B is the budget of cost, which is set large enough to ensure a positive reward $B - C^S(G)$ when the SLO is met. All variables in Eq. (4) are measured in units of milliseconds. In practice, some attempted strategies lead to OOM errors. We impose a large negative reward for those strategies.

RL Training We jointly train the partitioner and the placer to maximize the reward. Both agents are two-layer neural networks and use stochastic policies for decision making. In particular, the partitioner decides layer grouping and parallelization by drawing a sample u from partitioning policy π_a; the placer then decides the host functions for the group partitions by drawing a sample v from placing policy π_b. As sample v is conditional on u, we rewrite the reward $R^S(G)$ as $R(v|u)$. We encode π_a and π_b into two policy networks, parameterized by θ_a and θ_b, respectively, We jointly train the two policies to maximize the expected reward, defined as

$$J(\theta_a, \theta_b) = \mathbf{E}_{u\sim\pi_a, v\sim\pi_b}[R(v|u)]. \quad (5)$$

We compute the policy gradients for the partitioner and the placer using REINFORCE [44], respectively:

$$\nabla_{\theta_a} J(\theta_a, \theta_b) = \mathbf{E}_{u\sim\pi_a}[\nabla_{\theta_a} \log p(u;\theta_a) \cdot \mathbf{E}_{v\sim\pi_b}[R(v|u)]],$$
$$\nabla_{\theta_b} J(\theta_a, \theta_b) = \mathbf{E}_{u\sim\pi_a, v\sim\pi_b}[\nabla_{\theta_b} \log p(v|u;\theta_b) \cdot R(v|u)],$$

where $p(\cdot)$ is the probability density function. With these two gradients, we update θ_a and θ_b using Adam optimizer [45], and iteratively learn the optimal policies.

V. Evaluation

We have implemented Gillis along with the two partitioning algorithms in 3K lines of Python code. In Gillis, a user simply provides a DNN model file in ONNX format and chooses from the two serving modes Gillis provides: latency-optimal and SLO-aware. A user also specifies the latency requirement in the SLO-aware mode—Gillis will notify the user if the SLO requirement is too restrictive that cannot be met. Gillis figures out the parallel execution plan *offline* in the chosen mode using the corresponding partitioning algorithm, and serves the inference queries *online* following that plan. We evaluate Gillis on AWS Lambda [10], Google Cloud Functions [11] and KNIX [24] with popular convolutional and recurrent neural networks. Our evaluations are set to answer the following questions:

- How does Gillis perform when used to reduce the inference latency (§V-B)?
- Can Gillis meet the latency SLOs while minimizing the serving cost (§V-C)?
- How do the algorithms behave in microbenchmarks (§V-D)?

A. Methodology

Benchmarking Models We use four families of popular DNNs as the benchmarking models: VGG [30], ResNet [33], Wide ResNet [14], and Multi-layer recurrent neural networks (RNNs) [15]. Each family has multiple model variants. For Wide ResNet (§II-B), we test models with 34 and 50 layers with widening scalar ranging from 3 to 5. We use notations like WRN-34-3 to denote the ResNet-34 model widened by $3\times$. For RNN models, we can improve the accuracy by adding more RNN layers and using a larger hidden size [13], [15]. Both approaches lead to the increased network parameters. We test RNN models of various numbers of layers with 2K hidden size, and use notations like RNN-6 to denote a 6-layer RNN model. All RNN model variants use LSTM cell.

Serverless Platforms We evaluate Gillis on AWS Lambda functions and Google Cloud Functions with their respective maximum possible instance memories (i.e., 3GB and 4GB, respectively) at the time of experiments[3]. We also evaluate Gillis on KNIX, an open-source serverless platform enabling faster function communications than Lambda (Fig. 7). We deploy KNIX on an EC2 `c5.12xlarge` instance, and configure the resources of each KNIX function to match the performance of a Lambda instance.

Memory Footprint The memory footprint of model serving in a Lambda function is usually much larger than the model weight size, as the function also needs to load the OS kernel, maintain a software stack, and cache the runtime data. We empirically characterize the maximum memory to host the model weights, and set the maximum available function memory M to 1.4 GB in our partitioning algorithms (see Eq. (2)).

Metrics We use the inference latency and the serving cost as the two primary performance metrics. In particular, when comparing Gillis with the baseline, we measure the *inference speedup* as $L_{\text{Baseline}}/L_{\text{Gillis}}$, where L_{Baseline} and L_{Gillis} respectively denote the inference latencies of the baseline and Gillis.

We measure the inference cost as the billed function duration under the pricing scheme [32], [46]. We ignore the function invocation charges, which are two orders of magnitude smaller than the duration charges.

B. Gillis in Latency-Optimal Mode

Baselines We start with the latency-optimal (LO) mode, where we evaluate Gillis against two baselines:

(1) Default serving (**Default**) uses a *single function* to serve a model. Due to the memory constraint, it only applies to a model of a moderate size.

(2) Pipelined execution (**Pipeline**) divides layers of a large model into small partitions and stores them in S3 [47]. It then launches a single function to *sequentially* execute these partitions in a pipeline, one partition at a time.

Serving CNN Models We first evaluate the latency performance of serving popular CNN models on Lambda and Google Cloud Functions (GCF). Fig. 9 compares the latencies of various VGG and WResNet models over 100 queries using Gillis (LO) and Default. The error bars measure the latency variances, which are almost indistinguishable, an indication of the strong performance isolation between function instances. We observe that parallelizing CNN models across multiple functions significantly speeds up the inference than serving them using a single function (Default) for both platforms. Compared with Google Cloud Functions, Lambda can enable more latency improvements for Gillis due to its less resources per instance. For example, Gillis on Lambda achieves $1.4\times$ speedup for WRN-50-3 while it reduces to $1.2\times$ on Google Cloud Functions. In addition, as the model grows deeper and wider with more widened convolution layers, Gillis identifies more parallelization opportunities, leading to more salient latency improvements. In particular, on Lambda we observe $1.6\times$, $1.9\times$, and $2\times$ speedup for VGG-11, VGG-16, and VGG-19, respectively; widening ResNet from WRN-34-3 to WRN-34-4 also improves the speedup from $1.2\times$ to $1.26\times$.

We also evaluate Gillis on KNIX. Fig. 10 shows the latencies of various CNN models using Gillis (LO) and Default KNIX serving. Owing to the low-latency communication in KNIX, Gillis can reap more benefits from parallelization compared with Lambda. Gillis hence achieves greater latency improvement (Fig. 10, left), leading to $3\times$, $2.9\times$, and $1.8\times$ speedup over Default KNIX for VGG-16, VGG-19, and WRN-50-3, respectively. Moreover, the improved communication enables Gillis to speed up more "thin" models (e.g., classical ResNet), which fail to be accelerated on Lambda (Fig. 10 right). In particular, Gillis is $1.4\times$, $1.6\times$, and $1.3\times$ faster than Default KNIX for ResNet-34, ResNet-50, and ResNet-101, respectively.

We next evaluate large CNN models that cannot be handled by Default. Fig. 11 compares the latencies of those models using Gillis and Pipeline. We break down the end-to-end latency of Pipeline into the function computation time and the network transfer time for loading model partitions. Due to the limited network bandwidth, its communication overhead becomes a severe bottleneck. In comparison, Gillis transfers no weight tensors from remote storage. Moreover, its parallel execution is $2\times$ faster than the sequential execution of Pipeline (see Pipeline-comp.). Owing to these two advantages, Gillis significantly outperforms Pipeline, speeding up the end-to-end latency by $9.1\times$, $9.2\times$, and $8.3\times$ for WRN-34-5, WRN-50-4, and WRN-50-5, respectively.

Serving RNN Models We next turn to RNN models. RNN layers need to be unrolled in multiple time steps, and they cannot be parallelized spatially like the convolution layers. Fig. 12 depicts the mean inference latencies of various RNN models on Lambda using Gillis (LO) and Default serving. As RNN layers cannot be parallelized, Gillis shows no advantage over Default for small models that can fit into a single function. However, a single function can only support RNN models with up to 9 layers. Gillis has no such limitation on the model size, but *linearly scales* to large RNNs. The inference latency grows linearly as the number of RNN layers increases, indicating that

[3]Our experiments were conducted in September and October 2020.

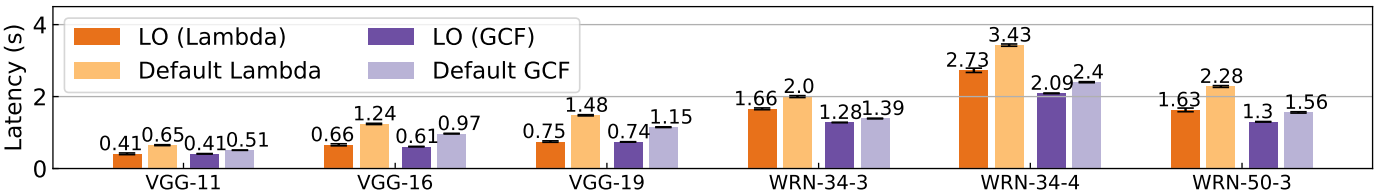


Fig. 9: The inference latencies of various CNN models on Lambda and Google Cloud Functions (GCF) using Default and Gillis's latency-optimal partitioning (LO).

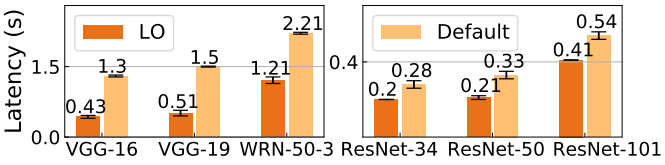


Fig. 10: The inference latencies of various CNN models on KNIX using Gillis (LO) and Default.

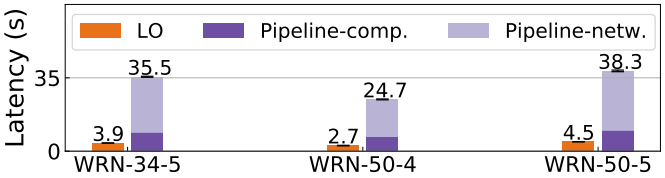


Fig. 11: The inference latencies of serving large CNN models on Lambda using Gillis (LO) and Pipeline. We break down the latency of Pipeline into the computation and network time.

the function communication overhead is minimized in Gillis. This suggests that the function communication overhead is minimized in Gillis, thanks to its layer grouping strategies and efficient function coordination mechanism.

C. Gillis in SLO-Aware Mode

Baselines We now switch to the SLO-aware (SA) mode and evaluate the cost performance of Gillis on Lambda under SLO constraint against two baselines:

(1) Brute-force search (**BF**) enumerates all possible parallelization strategies with SLO compliance and finds the one with the minimum cost. In light of its computational intractability, we only apply it to VGG-11, the smallest model in our benchmarks, which still takes over 24 hours to complete.

(2) Bayesian Optimization (**BO**) models the inference cost as a Gaussian Process, and iteratively refines the model by sampling to search for the optimal strategy. Our BO algorithm follows the same design as Cherrypick [42], where we use the *expected improvement* (EI) [48] to evaluate sampled strategies.

SLO and Models We use four CNN models in our experiments, VGG-11, VGG-16, WRN-50-4, and WRN-50-5, each evaluated with two latency SLOs, restrictive and loose. We do not evaluate Gillis (SA) against RNN models, because without parallelization, minimizing the serving cost is equivalent to optimizing the latency (§V-B).

Cost Performance and SLO Compliance We evaluate the SLO-aware partitioning (SA) against BO and BF. As both SA and BO are randomized algorithms, we conduct three

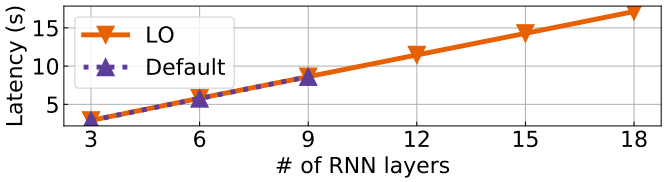


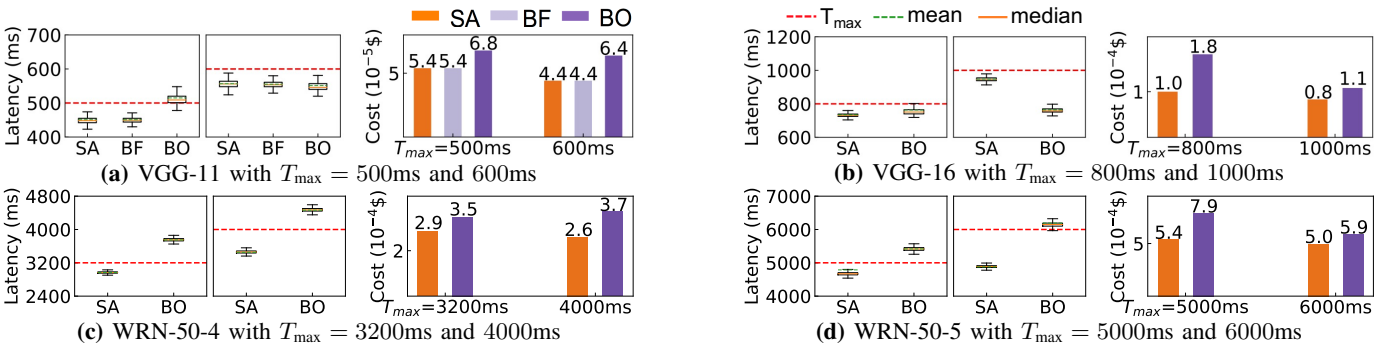
Fig. 12: The mean inference latencies of various RNN models on Lambda using Gillis (LO) and Default.

experiments and report the best result for each algorithm. In each experiment, we launch 100 clients that concurrently query the inference service 1000 times.

Fig. 13 compares the latencies and costs of the three solutions applied to various models with different latency requirements $T_{\max}$. Compared with BF that optimally parallelizes VGG-11 for cost minimization, Gillis (SA) learns the same partitioning strategy (Fig. 13a), achieving the similar latency and cost with much smaller computational overhead. Compared with BO that applies black-box optimizations, Gillis's RL algorithm uses the performance model to guide the search of optimal partitioning in simulated experiments, leading to much improved results. In fact, BO fails to meet the latency SLOs in several cases, especially for complex models like WRN-50-4, or restrictive latency requirements like VGG-11 with $T_{\max} = 500$ms. In contrast, Gillis (SA) can always meet the SLOs while achieving significant cost savings than BO, e.g., up to $1.8\times$ for VGG models and $1.5\times$ for WResNet.

D. Microbenchmark

To understand how Gillis makes partitioning decisions to reduce the inference latency, we turn to Fig. 14, in which we illustrate the layer grouping and parallelization results of WRN-34-5 given by the latency-optimal algorithm. WRN-34-5 has a total of 36 (merged) layers, where low convolution layers (closer to the input) have larger feature maps but smaller weight tensors than the high layers. We make three observations. First, compared with the top-level convolution layers (Groups 6 and 7), the optimal strategy tends to fuse more layers at the bottom (Groups 1 and 2) that have smaller sizes. Second, low convolution layers are usually parallelized across more functions (16 for Group 1) as they have large feature maps. Third, the master tends to compute group partitions of low convolution layers (Groups 2, 3, 4, and 5) with small weight tensors. This not only leads to reduced memory footprint, but also incurs less communication overhead between the master and workers.



(a) VGG-11 with $T_{\max} = 500$ms and 600ms

(b) VGG-16 with $T_{\max} = 800$ms and 1000ms

(c) WRN-50-4 with $T_{\max} = 3200$ms and 4000ms

(d) WRN-50-5 with $T_{\max} = 5000$ms and 6000ms

Fig. 13: Comparison of inference latencies and costs of the SLO-aware partitioning (SA), brute force (BF) and Bayesian optimization (BO) applied to various models with two latency thresholds (T_{max}). The latency boxes depict the 25^{th}, 50^{th}, and 75^{th} percentiles, and whiskers depict the 5^{th} and 95^{th} percentiles. Bars measure the average costs of serving an inference request.

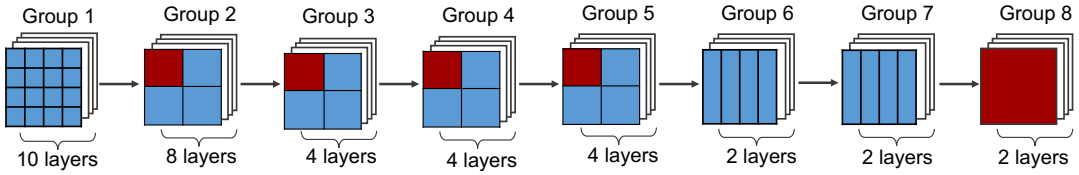


Fig. 14: An illustration of the grouping and parallelization results of WRN-34-5. There are eight layer groups in total, where the first seven contain convolution layers while the last includes a fully-connected layer. We draw the output tensor of each group, and show how it is partitioned along height (horizontal) and width (vertical) for parallel execution. The tensor partitions in red are executed on the master while those in blue run on workers.

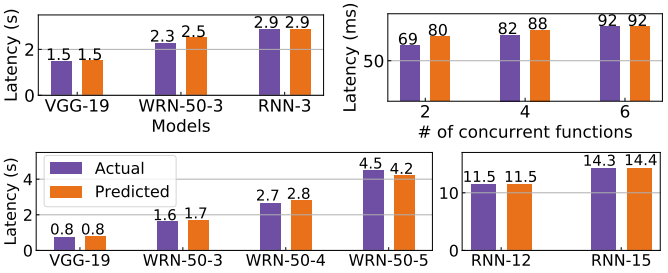


Fig. 15: Comparison of the actual and predicted model runtimes (top left), function communication delays (top right), and end-to-end latencies in latency-optimal mode (bottom).

E. Performance Model

Recall that we use the performance model to predict the latency performance of a parallelization strategy(§IV-A). We next evaluate its prediction accuracy by comparing the predicted model runtime, function communication delay, and end-to-end latencies with the actual measured ones in Lambda functions.

Fig. 15 (top left) depicts the actual and predicted model runtimes on a Lambda instance. Our performance model makes accurate predictions, deviating from actual ones within 3%, 9%, and 1% for VGG-19, WRN-50-3, and RNN-3, respectively. To measure the communication delay, we transfer 1MB data between the master and workers, as they usually exchange tensors of 100s of KBs. Fig. 15 (top right) shows the actual and predicted delays of various numbers of concurrent workers, where the average prediction error is 6.3%.

As our performance model can closely estimate both the model runtime and function communication delays, it makes a high-fidelity prediction to the end-to-end latency. Fig. 15 (bottom) compares the actual and the predicted inference latencies of serving various CNN and RNN models using the latency-optimal partitioning algorithm, where the prediction errors are within 6% across all models.

VI. Discussion and Future Work

Increasing Function Size Since mainstream serverless platforms generally impose stringent constraints on function resources (e.g., Google Cloud Functions [11] and IBM Cloud Functions [49]), Gillis can be widely applicable to improving serverless-based model inference. While we notice that some platforms recently increase their function sizes[4], we believe that parallel model executions across multiple functions is still necessary in the serverless cloud due to two reasons. First, the ever-increasing model parameters, which have doubled every 2.4 years [31], is expected to outpace the growth of function size in the future. Second, we observe that next-generation serverless platforms (e.g., KNIX [24]) enable increasingly faster function communications, making Gillis's parallelization more efficient (See Fig. 10).

Tail Latency SLOs In Gillis, the SLO requirements are specified by the mean latency. Yet, a more common SLO definition concerns the tail latency [3], e.g., at least 99% of queries should be served in 1s. Our RL-based optimization

[4]For example, AWS Lambda increases its memory limit per instance from 3GB to 10GB from December 2020 [50].

(§IV-C) can still be applied to meet these SLOs, as long as the tail latency can be accurately predicted. However, predicting the tail is more challenging and requires extensive profiling to cover more corner cases, which we leave as a future work.

VII. Conclusion

In this paper, we have described Gillis, a serverless-based model serving system that automatically parallelizes a large DNN model across multiple functions for faster inference and reduced per-function memory footprint. We have proposed two partitioning algorithms to achieve latency-optimal serving and cost-optimal serving with SLO compliance, respectively. Evaluations show that Gillis accelerates model inference, supports very large DNNs, and enables significant cost savings while meeting the specified latency SLOs.

Acknowledgment

This research was supported in part by RGC RIF grant R6021-20 and RGC GRF grants 16213120, 16207818, and 16209120. Minchen Yu was supported in part by the Huawei PhD Fellowship Scheme.

References

[1] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *Proc. USENIX NSDI*, 2017.

[2] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: Distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency," in *Proc. ACM Middleware*, 2017.

[3] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving," in *Proc. USENIX ATC*, 2019.

[4] "Amazon SageMaker," https://aws.amazon.com/sagemaker/.

[5] "Seamlessly scale predictions with AWS lambda and MXNet," https://go.aws/2SJyER0.

[6] "Simplifying ML predictions with google cloud functions," https://bit.ly/2YIVFHs.

[7] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *Proc. IEEE Intl. Conf. Cloud Engineering (IC2E)*, 2018.

[8] M. Fotouhi, D. Chen, and W. J. Lloyd, "Function-as-a-service application service composition: Implications for a natural language processing application," in *Proceedings of the 5th International Workshop on Serverless Computing (WOSC)*, 2019.

[9] E. Jonas, A. Khandelwal, K. Krauth, J. Schleier-Smith, Q. Pu, N. Yadwadkar, I. Stoica, V. Sreekanti, V. Shankar, J. E. Gonzalez, D. A. Patterson, C.-C. Tsai, J. Carreira, and R. A. Popa, "Cloud programming simplified: A Berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[10] "AWS Lambda," https://aws.amazon.com/lambda/.

[11] "Google Cloud Functions," https://cloud.google.com/functions.

[12] "Microsoft Azure Functions," https://azure.microsoft.com/en-us/services/functions/.

[13] M. Wang, C.-c. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proc. ACM EuroSys*, 2019.

[14] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.

[15] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, "Exploring the limits of language modeling," *arXiv preprint arXiv:1602.02410*, 2016.

[16] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[17] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[18] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," *arXiv preprint arXiv:1807.05358*, 2018.

[19] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in parallelizing convolutional neural networks," in *Proceedings of the 35 th International Conference on Machine Learning*, 2018.

[20] L. Zhou, H. Wen, R. Teodorescu, and D. H. C. Du, "Distributing deep neural networks with containerized partitions at the edge," in *Proc. USENIX HotEdge*, 2019.

[21] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[22] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX ATC*, 2018.

[23] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in *Proc. CIDR*, 2019.

[24] "KNIX," https://knix.io.

[25] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards high-performance serverless computing," in *Proc. USENIX ATC*, 2018.

[26] "MXNet," https://mxnet.apache.org.

[27] "Open Neural Network Exchange (ONNX)," https://onnx.ai.

[28] "Autoscaling configuration of AWS SageMaker," https://docs.aws.amazon.com/sagemaker/latest/dg/endpoint-scaling-loadtest.html.

[29] A. Bhattacharjee, A. D. Chhokra, Z. Kang, H. Sun, A. Gokhale, and G. Karsai, "BARISTA: Efficient and scalable serverless serving system for deep learning prediction services," *arXiv preprint arXiv:1904.01576*, 2019.

[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[31] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[32] "Google Cloud Functions Pricing," https://cloud.google.com/functions/pricing/.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE CVPR*, 2016.

[34] Y. Gao, L. Chen, and B. Li, "Spotlight: Optimizing device placement for training deep neural networks," in *Proceedings of the 35th International Conference on Machine Learning*, 2018.

[35] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *Proc. ACM ICML*, 2017.

[36] "MXNet Model Server," https://pypi.org/project/mxnet-model-server/.

[37] "Keeping functions warm - how to fix AWS lambda cold start issues," https://serverless.com/blog/keep-your-lambdas-warm/.

[38] "How long does AWS Lambda keep your idle functions around before a cold start?" https://bit.ly/2AVfxgL.

[39] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *Proc. USENIX NSDI*, 2017.

[40] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[41] J. E. Gentle, *Computational statistics*. Springer, 2009.

[42] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *Proc. USENIX NSDI*, 2017.

[43] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A hierarchical model for device placement," 2018.

[44] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press, 2011.

[45] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[46] "AWS Lambda Pricing," https://aws.amazon.com/lambda/pricing/.

[47] "AWS S3," https://aws.amazon.com/s3/.

[48] E. Brochu, V. M. Cora, and N. de Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *arXiv preprint arXiv:1012.2599*, 2010.

[49] "IBM Cloud Functions," https://cloud.ibm.com/functions/.

[50] "AWS Lambda increases function size," https://amzn.to/2WA7kWS.