# Reducing Serverless Inference Latency of Partitioned Deep Neural Networks Using a Hybrid Inter-Function Tensor Sharing Strategy*



Douglas Issichopoulos, Patrick Sun, Reza Farivar
*Department of Computer Science*
*University of Illinois Urbana-Champaign*
{doi4, ptsun2, farivar2}@illinois.edu

***Abstract*—Cloud-based serverless platforms offer an efficient solution for deploying Deep Neural Networks (DNNs). These platforms handle the complex infrastructure provisioning tasks, scale dynamically, and charge developers only for the actual runtime of the inference requests. Recent research suggests that partitioning a large DNN into smaller, parallelizable submodels deployed in separate serverless functions can minimize end-to-end inference latency. This partitioned design requires sharing intermediate tensors among the coordinated serverless functions. While the current state-of-the-art system achieves this by encoding the tensor information in the payload of a REST API endpoint, our study demonstrates that using a remote object storage system for tensors exceeding a specific size threshold can facilitate faster sharing. Therefore, we propose a hybrid inter-function tensor sharing strategy based on tensor size to further decrease the end-to-end latency in partitioned DNN models. Our experimental results indicate that implementing this hybrid strategy can reduce latency by up to 17.3%.**

***Index Terms*—Serverless computing, Artificial neural networks, Tensors, Object oriented databases**

## I. Introduction

Machine learning models are growing increasingly important in software systems across the globe. Among the many varieties of models in existence, Deep Neural Networks (DNNs) have become especially popular, seeing as they have impressively high prediction accuracy and can be used for a wide variety of applications [16].

The ability to minimize inference latency—the duration of time between querying a trained model and receiving its prediction response—can be supremely valuable for many applications. Given that representatives from Microsoft and Google famously presented in 2009 that artificially introducing latencies in their web systems caused a significant decrease in revenue [3], it is important for any company or organization deploying machine learning models in cloud systems to keep latency as a primary consideration.

Machine learning model inferencing typically requires advanced hardware and a significant amount of memory. It should be no surprise, then, that traditionally the primary means to serve trained machine learning models in a cloud environment has been by using virtual machines (VMs), and there is ongoing research to improve the efficacy of VMs for model serving [7] [5] [17] [21] [25].

Still, an attractive alternative to VMs for deploying machine learning models is by leveraging serverless architectures. They offer numerous benefits for customers, such as handling complex infrastructure provisioning tasks and scaling dynamically to different request volumes, all while only charging for the actual runtime of the inference requests. Research from 2018 showed that even these comparatively larger DNN models can be deployed for inferencing using serverless technology [8], and it has continued to be an active area of research.

The most obvious approach for deploying a trained DNN model with a serverless platform is to use just *one* serverless function. This method minimizes complexity; when an inference request is received from a user by the serverless platform, it invokes the single function and returns the eventual result to the user. Some promising research with this design has involved batching multiple inference requests together (as opposed to processing each request separately) to reduce both latency and cost [2] [1].

Nevertheless, some DNN models are sufficiently large that they do not fit into a single serverless function, even though impressive research has gone into compressing the size of DNNs [6]. For some specific applications, researchers have been able to circumvent the issue by simply removing unneeded components of large packages like TensorFlow [4] or PyTorch [19]. While these approaches worked well for their particular applications, a more general solution is desired, especially as DNNs continue to grow in size.

With this in mind, we are particularly drawn to other research that focuses on partitioning large models into smaller submodels, where each submodel is ensured to be sufficiently small that it can comfortably fit within a single serverless function. While this approach is appealing, it comes with the formidable challenge that tensor information must now be shared between these functions, breaking the typical paradigm of serverless functions being independent.

Per the results of a recent survey from 2023 [22], there are two notable systems that employ a partitioning approach. The first is AMPS-Inf [9], which partitions the trained model into functions that run sequentially. The other is Gillis [24], which uses a master-worker paradigm that allows groups of worker functions to run in parallel. As mentioned above, there is great value in reducing end-to-end inference latency, and the authors of both systems reported latency measurements for several DNN models that had been partitioned and deployed by their respective systems. They only tested one model in common, namely ResNet50, and Gillis showed the lower latency for this model. While this is hardly a comprehensive comparison, it stands to reason that the parallelism found in Gillis is an advantage in processing inference requests with low latency, so we focus our investigation here on the Gillis system.

Gillis shares intermediate tensors between the master and worker functions by encoding the tensor information in the payload of a REST API endpoint. As argued in [12], however, when it comes to transmitting data between serverless functions, there are situations in which using *remote-storage* (e.g., AWS S3) can be more efficient than *direct-passing* (e.g., a REST API endpoint). Prompted by this claim, we set out to explore if using AWS S3 could be used to make Gillis even faster.

In this work, we propose sharing intermediate tensors between the serverless functions of a partitioned DNN using a REST API endpoint when the tensors are *smaller* than a certain size threshold and using AWS S3 when the tensors are *as large or larger* than the threshold. Moreover, we show that this hybrid tensor sharing strategy can further decrease the end-to-end inference latency compared to the original method used in the Gillis system. The primary challenges with this design are: determining the tensor size at which AWS S3 becomes preferable to the REST API approach, ascertaining whether tensors above this threshold are actually shared in real world partitioned DNNs, and implementing a configurable version of the hybrid strategy within the open-source Gillis system in order to test its efficacy.

This hybrid tensor sharing strategy was developed through a series of four steps, where the first two steps motivated the third and fourth. **First**, we performed an experiment to measure the impact of *tensor size* on the latency for both the REST API and AWS S3 sharing strategies. We found that AWS S3 has comparatively lower latencies for *large tensors*. **Second**, we profiled the size of all tensors shared between the AWS Lambda functions for our test set of large, partitioned DNN models. Notably only one model in our test set shared tensors that were sufficiently large to obviously benefit from being shared via AWS S3.

**Third**, bolstered by the results of these first two findings, we implemented the hybrid sharing strategy into the Gillis system with the following logic: if the size of a given tensor to be shared is at or above the user-provided threshold, it is sent using AWS S3; otherwise, it is sent using the original REST API approach. **Fourth** and finally, we performed an experiment in which we compared the end-to-end latencies for the test models using three tensor sharing strategies: exclusively via a REST API endpoint (as in the original Gillis system), exclusively via AWS S3, and via our hybrid strategy such that only the largest tensors for each model would be sent using AWS S3. In line with our hypothesis, the S3-only approach performed worst for all models, and the original Gillis method performed best for most models. However, for the model with the largest shared tensors as mentioned above, we found that our hybrid strategy produced a 17.3% decrease in median latency compared to the original Gillis method.

In summary, we claim any system that shares tensors between serverless functions ought to adjust its sharing strategy based on tensor size, and we show a particular application of our hybrid strategy successfully reduced the end-to-end inference latency of a large partitioned DNN deployed on a serverless platform. While we make no claims as to having discovered the single most optimal method, we believe the findings here are widely applicable and will help foment additional research on the matter, ultimately improving DNN inferencing for all.

## II. Background and Motivating Experiments

As mentioned above, we view Gillis [24] as the state-of-the-art system for partitioning and deploying large deep neural networks to serverless functions in terms of achieving low latency for inferencing requests, and in this work we propose an improvement to its tensor sharing strategy to further reduce latency. In this section specifically, we first conduct a literature survey on related works. Then we provide a high level overview of the Gillis design and explore how Gillis shares tensors between its serverless functions. Finally we describe our initial experiments that motivated our hybrid tensor sharing strategy.

### A. Related Works

Seeing as Gillis [24] was published in 2021 and serves as the foundational paper for our work here, it is especially important to address some additional related works.

Recently, researchers have sought to develop their own serverless systems that are specifically tailored to machine learning tasks. In 2022, researchers developed the *INFless* [23] system which claimed to be the "first ML domain-specific serverless platform." Their focus was different than ours, as their desire was to improve throughput for the serverless system rather than end-to-end latency for the cloud customer. Later in 2023, researchers built *ShmFaas* [11], a Kubernetes-based serverless inferencing system specifically for DNN models that demonstrated reduced memory usage, which again is valuable but does not focus directly on the latency.

Other research has sought to address specific components of inferencing systems for deep learning models using serverless platforms, as opposed to developing entirely new systems. One such work was produced by two authors from Meta [13], but in contrast to our work, their focus was on developing a hybrid scheduler for a serverless platform that could better allocate resources. Similarly, a related work from the recent

SoCC '23 conference called *AsyFunc* [15] also focuses on reducing inefficiencies that arise under bursty workloads with serverless inference for deep learning models rather than latency specifically, as we do.

The creators of the *All-You-Can-Inference* [14] system attempt to aid developers of DNN models who wish to utilize a serverless platform by helping them navigate the many configuration possibilities that exist in today's systems, but in contrast to our work, they do not consider partitioning or parallelization within individual DNN models.

An important area of research that is directly related to partitioning large DNN models is focused on mobile, edge, and IoT environments. These environments can be even more resource constrained than the serverless functions on which we focus, but there is great overlap between the two situations. It is worth mentioning here a work from 2018 called *DeepThings* [26] which focuses on parallelization in the IoT space, and it is referenced by the authors of the Gillis system itself. A more recent work called *SplitPlace* [20] utilizes a partitioning strategy called semantic splitting that allows for even more parallelization, but it comes at the cost of a "considerable drop in inference accuracy," to use their words. Another mobile device related work [18] also develops a novel parallel inference framework, this time for 6G mobile communication systems.

While there is much to be learned from these more recent systems, we are aware of no other works since AMPS-Inf and Gillis that specifically address serving partitioned DNN models using serverless platforms.

### B. Gillis Design Overview

Gillis is an impressive Python system that takes as input an ONNX-formatted file of a trained DNN model, partitions the model using one of its proprietary partitioning algorithms (we focus here on the one that minimizes latency), and then deploys the inferencing system as an interconnected network of serverless functions. Gillis can deploy to AWS Lambda, Google Cloud Functions, and KNIX, but we focus exclusively on AWS Lambda deployments. Also, the Gillis authors focus their investigation on image classification models, so we do as well for the sake of doing an apples-to-apples comparison.

The serverless functions are organized in a multistage master-worker paradigm, an example of which is shown in Figure 1. When an inferencing request is received at the master function, the processing is performed in stages. Generally each stage begins with the master function creating and sending input tensors to four worker functions, which run in parallel. After each worker has finished processing, it sends its output tensor back to the master, where the master aggregates the tensors from each worker function. Some partitioned models may be comprised of only a single stage, while many are made up of several stages. As a caveat, the algorithm at times has all processing for a given stage take place at the master function, and in such situations no tensors are shared between any worker functions.

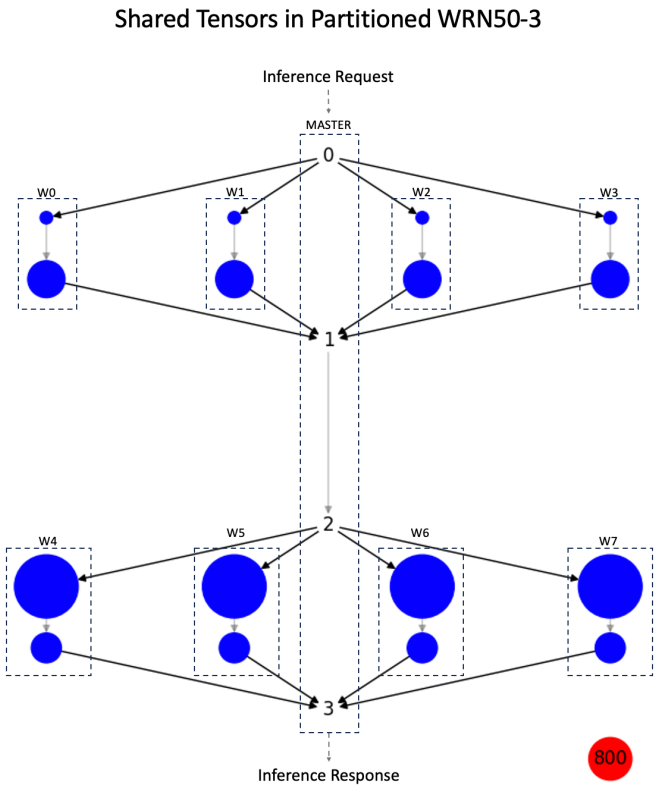


**Fig. 1: Visualization of the shared tensors during an inferencing request by WRN50-3 image classification model that has been partitioned and deployed as AWS Lambda functions using the Gillis system. The dashed boxes represent the master and eight worker functions. The blue circles represent the input and output tensors for workers, all of which require being shared, and each is drawn to scale relative to the red circle in the bottom right representing 800 KB. The numbers 0 through 3 represent the stages that are initiated at the master function. Notice no tensors are shared for stage 1, which indicates that all computation for this stage is done locally by the master.**

Essential to this design is that the master endures for the whole duration of each inference request, coordinating the beginning and end of each stage. By contrast, the worker functions only endure for the duration of a single stage. Once the final stage is complete, the master returns the inference response. We have visualized this logic in Figure 2.

### C. Tensor Sharing in Gillis

Gillis uses a *direct-passing* approach in that the input and output tensors are directly shared between the master and worker functions using a `boto3` REST API endpoint. Again, see Figure 1 for a detailed visualization of how the many intermediate tensors move through each serverless function while processing a single inference request.

Tensors are represented in a format equivalent to a multidimensional `numpy` array, and when a serverless function

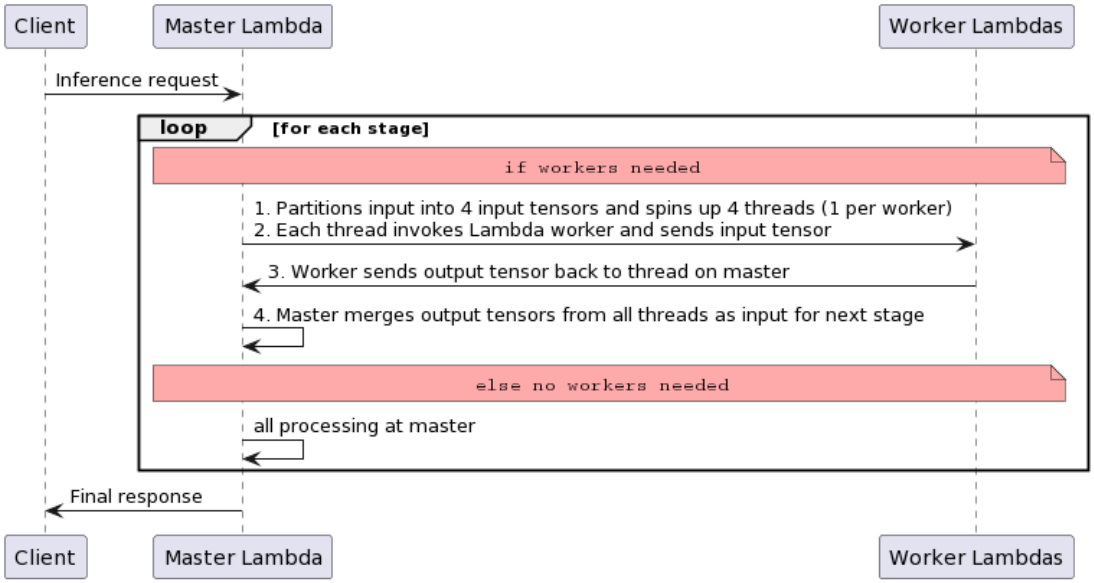


**Fig. 2: Overview of the inferencing request logic of the original Gillis system.**

needs to send a tensor to another function, the function first encodes the tensor into a base 64 bytes string. That encoded string is included in the payload of the API call that is sent to the receiving function. This process of encoding to a base 64 bytes string is beneficial because it makes the string shorter in length compared to an equivalent encoding in base 10 or base 2, and a shorter string allows the payload to be delivered faster to the receiving function. Once the encoded string reaches its destination function, the function can then decode the encoded string and convert it back into a multidimensional array, resulting in a successful transmission of the tensor from one serverless function to another.

### *D. Motivating Experiment: Pairwise Tensor Sharing Latencies by Tensor Size*

The guiding question for our first exploratory experiment was the following: when sharing a tensor between two AWS Lambda functions, for what size of tensor, if any, is it faster to use AWS S3 compared to the REST API endpoint? Conceptually, S3 has an advantage in that it does not need to encode or decode the multidimensional array of tensor data; the array can be saved directly to S3 by the sending function and then read directly by the receiving function (see Section III-D for details on how this is accomplished). That being said, this method has a disadvantage in that there is unavoidable overhead in writing to and reading from S3, even for very small tensors.

To implement our experiment, we created four new AWS Lambda functions. The first two functions were assigned to share tensors using the original REST API endpoint approach, and the remaining two functions were assigned to share tensors using AWS S3. Outlined below is the method used to measure the latency of sharing a tensor between the two functions of each pair.

- At the first function `F1` of the pair:
  - receives a request to share a tensor of a particular size
  - marks the start time
  - invokes the second function `F2`, forwarding the tensor size
- After being invoked, `F2`:
  - creates a tensor of the requested size in the form of a multidimensional `numpy` array of random values
  - calculates a checksum for the tensor
  - if using the S3 sharing method, then saves the tensor to S3; otherwise encodes the tensor to a base 64 bytes string
  - if using the S3 sharing method, then responds back to `F1` with the checksum and the S3 key of the saved tensor; otherwise responds back to `F1` with the checksum and the encoded tensor string
- Back at `F1` after receiving a response from `F2`:
  - if using the S3 sharing method, then reads the tensor from S3; otherwise decodes the encoded tensor data
  - performs the same checksum operation on the local tensor and compares it with the checksum received from `F2`, ensuring (with very high probability) that the received tensor is the same as the one that was sent
  - marks the end time and calculates the roundtrip latency
  - saves the following info: the tensor size, sharing method, latency, and a TRUE/FALSE value for whether the checksums matched

With this framework in place, we first tested the S3 sharing method. This testing involved warming the two functions (to avoid including latencies from cold starts), and then starting

with small tensors of about 8000 bytes, we made 10 repeat requests before moving on to the next tensor size. We increased in 8000 byte increments until stopping at approximately 1600 KB, spanning 200 different tensor sizes. Afterward, we repeated the same process for the REST API endpoint method.

Once all of the data was collected, we first confirmed that the checksums matched for every tensor sharing event. Next, we calculated the median latency across the 10 repeated requests for each tensor size by sharing method, which is summarized in Figure 3. The results were very clear that AWS S3 is the more efficient sharing method for tensors of about 800 KB and larger. To be sure, the latency for the REST API endpoint approach is much lower for *small* tensors, but as tensors increase in size, latency increases at a much higher rate compared to S3.

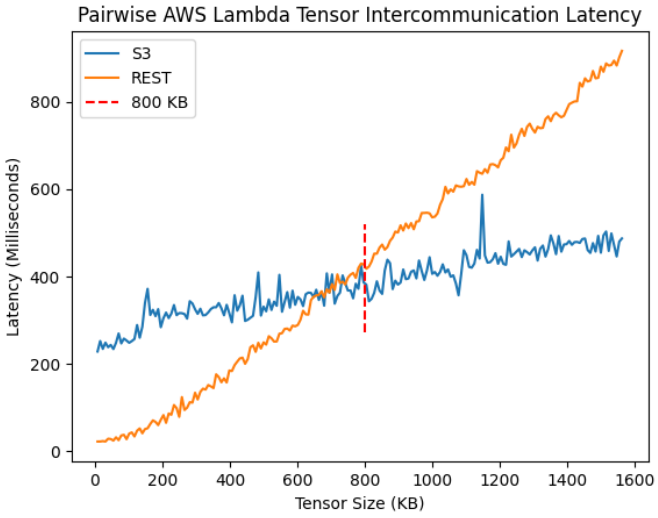


**Fig. 3: Median latencies to send tensors of various sizes between two AWS Lambda functions using either AWS S3 or a REST API endpoint. Tensors of about 800 KB and larger can be shared faster via AWS S3.**

### *E. Profiling Shared Tensors Sizes*

Motivated by the previous experiment, the second task was to measure the size of tensors that are actually exchanged by our test set of large DNN models partitioned using the Gillis system. After all, if all shared tensors are smaller than the theoretical tipping point of 800 KB, then using AWS S3 would likely be of little use in reducing end-to-end latency.

To perform this profiling, we added a custom feature to the Gillis codebase such that each shared tensor would first be saved to S3. While this idea is straightforward, the implementation involves several important details that are more fully refined in our final hybrid strategy, so we leave that discussion for the following section, specifically Section III-E.

The results of this profiling are shown in Figure 4. Most notably, the WRN50-3 (Wide ResNet50 with 3-fold cross validation) model shares tensors of about 1.7 MB, which is well beyond the approximate theoretical tipping point of 800 KB. Also, the WRN34-4 model shares tensors right at the threshold (800.1 KB, to be exact), and the remaining Wide ResNet model WRN34-3 shares tensors just below threshold. The remaining models, however, do not come close to the threshold.

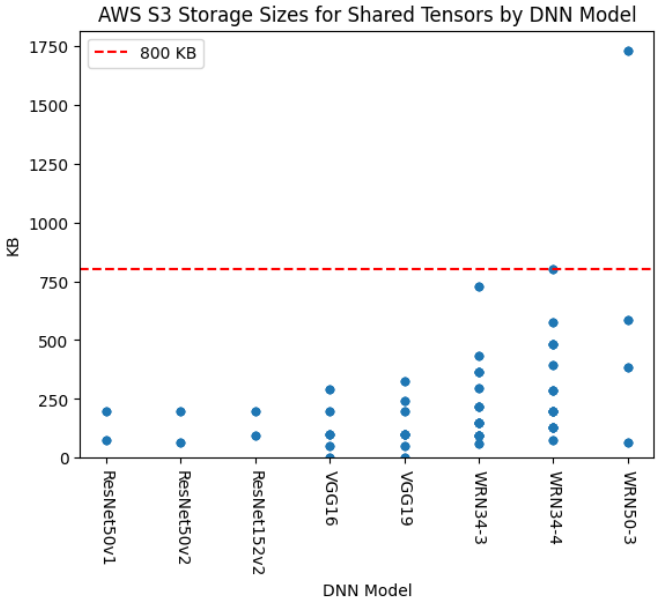


**Fig. 4: Storage sizes in AWS S3 for tensors shared between AWS Lambda functions of partitioned large DNNs deployed with Gillis. Note: due to Gillis' partitioning strategy, at each stage of processing four workers receive tensors of equal size from the master and then afterward share four output tensors of equal size back to the master. See Figure 1 for an example. With this in mind, each datapoint in this figure actually represents *four* tensors shared by a DNN model.**

Consider reviewing Figure 1 which depicts the size of tensors shared by the WRN50-3 model. As shown in the figure, the model sends *four* tensors of about 1.7 MB in size between the master and worker functions, as well as many other smaller tensors.

## III. System Design and Implementation

In this section, we describe the design and implementation of a hybrid inter-function tensor sharing strategy based on tensor size to further decrease the end-to-end latency in partitioned DNN models that have been deployed in AWS Lambda functions using the Gillis system. Conceptually, the strategy is simple: to share a given tensor, a Lambda function uses a REST API endpoint when the tensor is *smaller* than a user-provided size threshold and uses AWS S3 when the tensors are *as large or larger* than the threshold. We outline this design in Figure 5.

While the strategy is simple, implementing a working version within the existing open-source Gillis system has several subtle details, which we summarize here. First, we developed

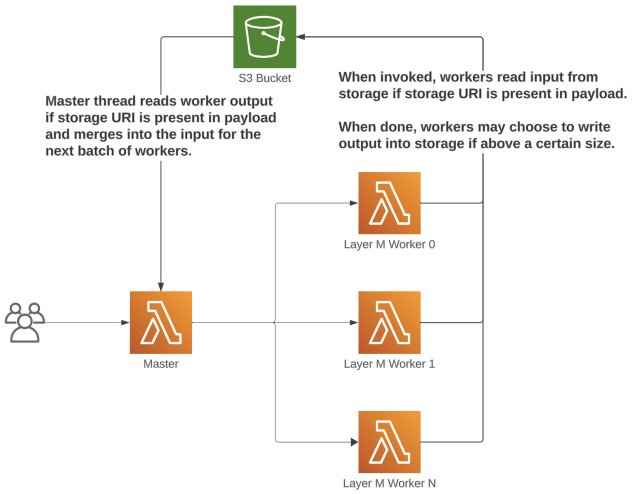


**Fig. 5: General system design of our hybrid tensor sharing strategy that utilizes AWS S3 to share tensors above a certain size threshold between the AWS Lambda functions.**

a method to specify an AWS S3 bucket for sharing tensors during the deployment process. Second, we added a means by which the master and worker functions can be deployed with additional AWS permissions to use that specified S3 bucket. Also, we created an S3 key naming scheme such that each tensor file's key is both unique and intelligible. Next, we identified a highly efficient method for saving tensors to S3 and subsequently reading tensors from S3. Finally, we brought it all together by updating the Gillis codebase with configurable hybrid tensor sharing logic.

### A. Specify AWS S3 Bucket for Sharing Tensors

As a first step, we added a script into the beginning of the deployment process that requires the user to specify the AWS S3 bucket to use for any S3 tensor sharing. The script queries the user's existing S3 buckets, allowing the user to either choose an existing bucket or let the script create a new bucket for the user.

We chose to specify the S3 bucket during deployment rather than allowing it to be configurable afterward for the sake of speed and security. In terms of speed, we wanted all functions to have local knowledge of exactly which bucket to use without having to query that information from elsewhere. In terms of security, we wanted each deployed function to have permission to access only one specific S3 bucket, as we also mention below.

### B. Additional AWS S3 Permissions

When deploying a partitioned DNN to AWS Lambda functions, the original Gillis system supplies the master function with a non-default permissions policy that allows it to invoke each of its worker functions. For our purposes, we needed to provide additional permissions for *all* functions to be able to read and write tensor data in the specified S3 bucket. The master function must be able to write input tensors to S3, and the worker functions must be able to read those input tensors from S3. Afterward, each worker function must be able to write its output function to S3, and the master function must be able to read those outputs from S3.

The advantage of specifying a bucket name in the permission policy of each function is that it provides greater security by ensuring the function cannot access any other S3 bucket, whether accidentally or maliciously during unintended usage.

### C. AWS S3 Key Naming Scheme for Tensors

Each object stored in AWS S3 is accessible via a unique URI that begins with `s3://my-bucket`, where in this case the placeholder bucket name is `my-bucket`. The remainder of the URI is known as the S3 key. As such, we had to develop a naming scheme for the S3 keys we use when reading and writing tensors to our specified S3 bucket. We wanted a scheme that would be unique for each tensor while also encoding valuable information about that tensor.

Without uniqueness, if concurrent image classification requests are made for different images, then the situation could easily occur where a tensor from one request gets overwritten by another request before the first request had time to read it, thereby spoiling the inference result of the first request. That being said, ensuring uniqueness by simply saving each tensor with a key comprised of an unintelligible hash value would be throwing away a lot of information about origin of the tensor.

With these points in mind, we use the following components in the S3 key for each tensor, which taken together ensure uniqueness while also preserving valuable details about the origin of the function.

- `deploymentName`: name of the deployment, which typically includes a description of the DNN model being deployed
- `ts`: human readable timestamp of when the initial inference request was received at the master function (e.g., '2023-11-03_17h05m06s_UTC')
- `id`: AWS-generated request id from when the initial inference request was received at the master function
- `stageNum`: stage number within the inference request processing
- `funcName`: name of the worker function
- `filename`: either 'input.npy' or 'output.npy' depending on whether the tensor is an input to or output for the worker function

### D. Efficiently Sharing Tensors with AWS S3

To share a tensor between two AWS Lambda functions using AWS S3, one function saves the tensor to S3, then the other reads the tensor from S3, thereby completing the transfer. Seeing as our goal is to minimize latency, ideally we would like to use the fastest method for saving a tensor to S3 and similarly use the fastest method for reading a tensor from S3. We leave it as a future work to more rigorously identify these optimal methods.

For our purposes, we are content to lower the bar slightly from using the *optimal* methods to using *highly efficient* methods. To that end, we choose to use the highly optimized Python packages `numpy` and `boto3` to construct our methods.

Tensors are stored internally in a format equivalent to a multidimensional `numpy` array, so to save a tensor to S3, we write the array to a buffer using the `numpy.save` method and then save the buffer to S3 using the `upload_fileobj` method from `boto3`'s `S3.Client` class. The process is analogous for reading a tensor from S3: the saved tensor data is written to a buffer using the `download_fileobj` method from `boto3`'s `S3.Client` class and then converted back into a multidimensional `numpy` array using the `numpy.load` method.

### E. Updating Gillis with Configurable Hybrid Tensor Sharing Logic

With all of the aforementioned components in place, we were ready to bring a configurable version of hybrid tensor sharing logic to the Gillis system. We use the term *configurable* here to indicate that we wanted the user to be able to provide a parameter to the inference request indicating the specific tensor size in KB for which tensors of this size or larger would be sent using AWS S3 and tensors smaller than this size would be sent using the REST API endpoint. Even though we had already found a supposed optimal threshold of about 800 KB, for experimental purposes we wanted this threshold to be configurable by the user.

Unfortunately, without major design changes to the serverless function network structure, it is not feasible for AWS S3 to trigger either a worker function or the master function upon the successful upload of a tensor because the master function endures for the duration of the inference request and inherently must wait by design. For this reason, the implementation we used expands upon the concepts we established in our first experiment described in Section II-D.

One important difference, however, is that in the first experiment each pair of functions only used one sharing method, but the master and worker functions we deploy via Gillis need to be able to share via both methods. We accounted for this by including additional information in the payloads of the REST API communication exchanged between functions. In the case of the master function invoking a worker function, the payload the worker receives contains both the user-specified tensor size threshold and a description indicating whether the worker ought to expect an encoded bytes string in the payload (as in the case of a REST API endpoint sharing method) or whether it should expect to find an S3 key in the payload (as in the case of the AWS S3 sharing method). Depending on how the descriptor dictates, the worker can take the appropriate action to acquire the input tensor.

After the worker function has performed its processing on the input tensor to produce its output tensor, it is now ready to send this output tensor back to the master function. To do so, it first performs a simple calculation to determine if the output tensor is above or below the tensor size threshold it received from the master. Depending on the result, it performs the appropriate operations for the corresponding sharing method and likewise includes the appropriate information in the payload of its response back to the master.

For the sake of clarity, we mention here that the method with which the master function first shares an input tensor with a worker function has no bearing on the method with which the worker function must share the output back to the master; the two sharing operations are completely independent. See Figure 6 for a visualization of our hybrid tensor sharing logic.

## IV. Experiment

After successfully implementing a hybrid tensor sharing strategy within the Gillis system, we were finally ready to perform an experiment to see if this method could improve end-to-end inference latencies for any of our test set of DNN models.

### A. Experiment Design

Our test involved first warming a partitioned and deployed DNN model to avoid cold starts. After the Lambda functions for the model were warmed, we made 30 consecutive inference requests for each of the following tensor sharing strategies:

- All tensors shared via the REST API endpoint (denoted "REST Only"). This is equivalent to the original Gillis method, and it was achieved by not providing any tensor size threshold in the invocation request.
- All tensors shared via AWS S3 (denoted "S3 Only"). This was achieved by providing 0 KB as the tensor size threshold (since all sharable tensors would be at least 0 KB and therefore shared via AWS S3).
- Each model's *largest* tensors shared via S3 but all smaller tensors shared via the REST API endpoint (denoted "Hybrid"). This was achieved by providing a tensor size threshold that was smaller than its largest tensors but larger than its second largest tensors.

After saving these 90 end-to-end latency and prediction measurements for the model, we repeated the process for the next model in our test set until we completed all models.

### B. Results

After the results were collected for all models and all tensor sharing strategies, we first confirmed that the prediction values across all inference requests for a given model were the same. Next, we calculated the median latency across the 30 requests for a given model and tensor sharing strategy, and those median latencies are summarized in Figure 7.

Generally speaking, the S3-only approach performed worst for all models, and the latencies were especially high for the two WRN34 models. Referencing back to a prior figure, Figure 4, this relationship seems to be most closely tied to size and number of tensors shared that are below the 800 KB threshold.

Excluding the WRN34-4 and WRN50-3 models for a moment, comparing the hybrid sharing strategy and REST API

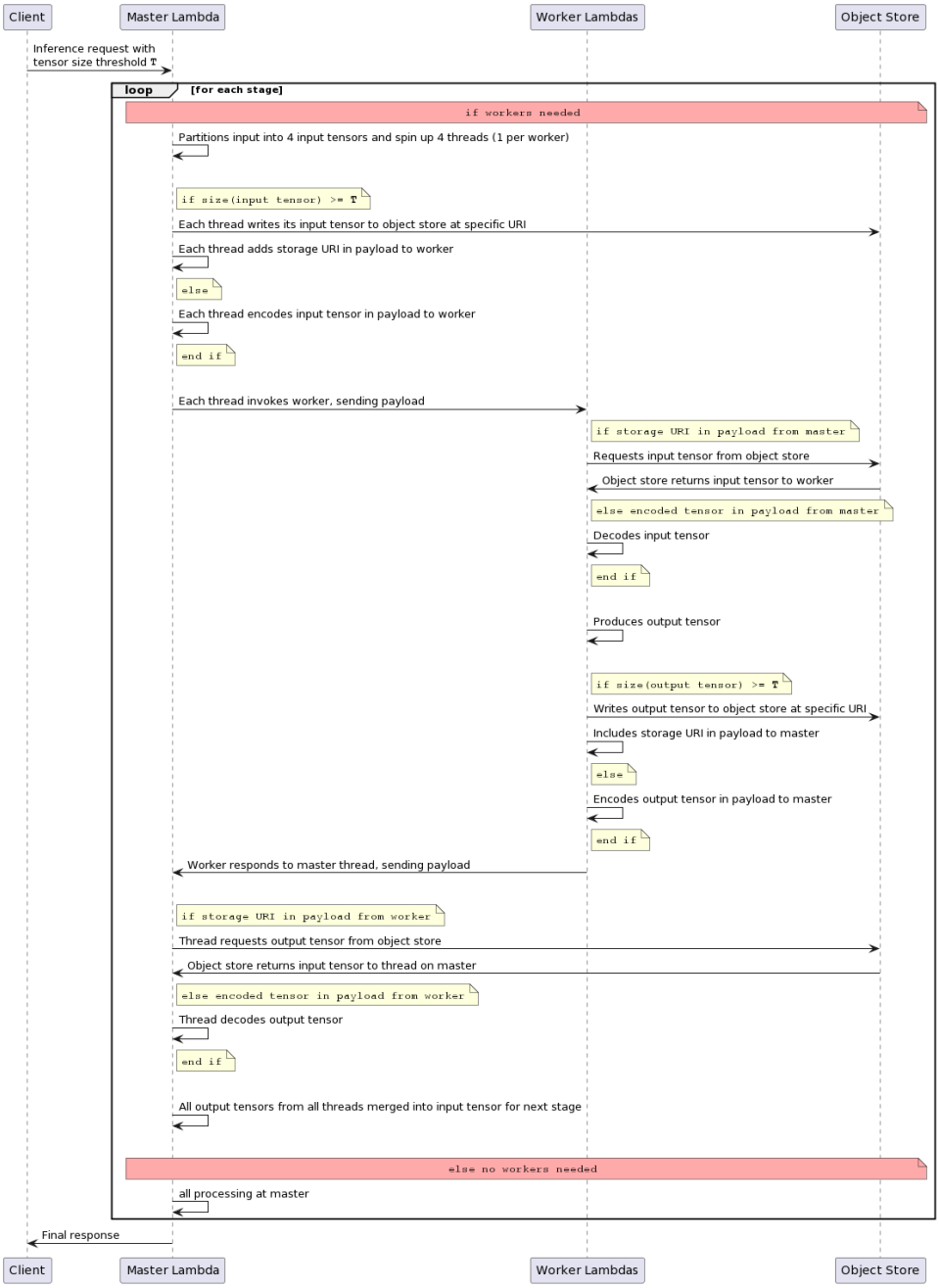


**Fig. 6: Overview of the hybrid tensor sharing strategy logic implemented as an update with the Gillis system. Compare this logic with that of the original Gillis described in Figure 2.**

endpoint approach shows that the REST API endpoint performed better, although the difference appears to be relatively small. We remind the reader that the hybrid strategy used here is such that only the *largest* tensors for each model were shared via S3, while the remaining tensors are shared via a REST API endpoint. So for the VGG and WRN models with many shared tensors, for both strategies the majority of tensors are shared via the REST API approach.

In line with our hypothesis, for the WRN50-3 model with the largest shared tensors, the hybrid strategy produced a 17.3% reduction in median latency compared to the REST API endpoint strategy in our experiment (1267.5 vs 1532.5 milliseconds, respectively), which is a non-trivial improvement. We also performed the same comparison using mean instead of median and found a 15.9% improvement for the hybrid strategy compared to the REST API approach (1292.1 vs 1536.7 milliseconds, respectively). See Figure 8 for a plot of all 90 latency data points for WRN50-3 (30 data points for each of the three tensor sharing strategies).

Returning now to WRN34-4 model, we saw nearly identical

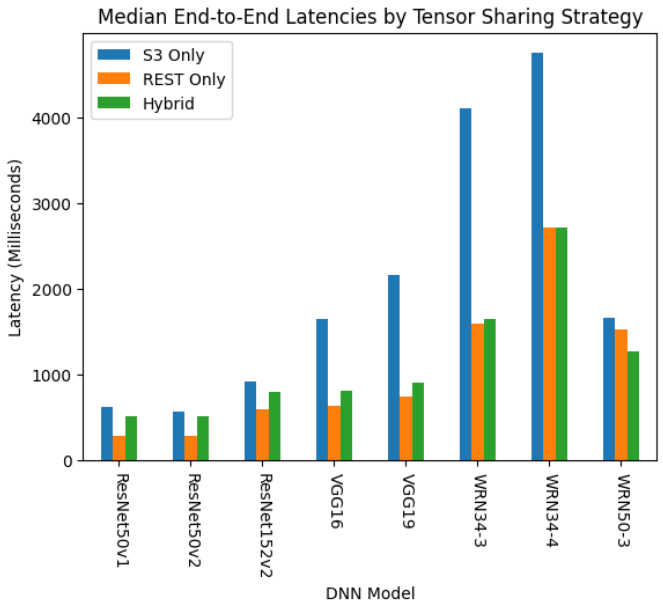


**Fig. 7: Median end-to-end latencies using three tensor sharing strategies across several large partitioned DNN models.**

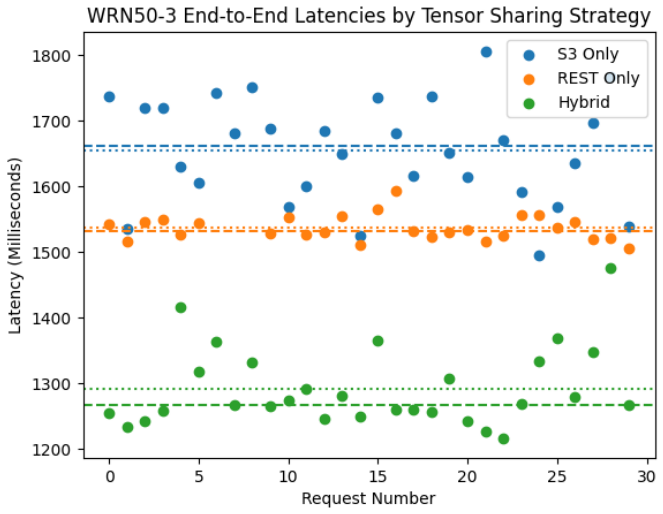


**Fig. 8: End-to-end latencies using three tensor sharing strategies for the partitioned WRN50-3 DNN model. Medians shown with dashed lines; means shown with dotted lines.**

results for the hybrid strategy and the REST API approach. The medians were 2712.5 vs 2713.0 milliseconds, respectively, and the means were 2724.5 vs 2717.4 milliseconds, respectively. Recall that model's largest shared tensors are 800.1 KB in size, so these results provide good evidence that 800 KB is a legitimate threshold.

### C. Implications for Future Work

While the results of this experiment are very promising, they also prompt further questions.

The models in our test set were comprised of the largest publicly available image classification DNN models we could successfully partition and deploy using the Gillis system. However, many more models exist, so we would like to expand our experiment to include even larger models and DNNs from other model families.

As mentioned in Section III-D, we have not rigorously determined whether our method is the optimal way to save or read a tensor from AWS S3. The method we use here is efficient, but if we could determine a faster method, it would improve the end-to-end latencies. Furthermore, the authors of the Astrea [10] system conducted a survey as part of their work, and they found that other systems use intercommunication tools like Redis, AWS DynamoDB, and SQS, each of which may have their own advantages and disadvantages.

There is also the question of *cost* as it pertains to this hybrid strategy. While additional costs are incurred by reading and writing tensors to AWS S3 instead of directly passing the tensors via the REST API endpoint, additional savings are gained by reducing latency in that the billable runtime of the corresponding AWS Lambda functions is also reduced. We leave it as a future work to determine the degree to which the additional savings counteract the additional costs.

Finally, we speculate that in some situations inference latency could be further reduced if DNN models were partitioned into fewer stages with larger shared tensors rather than into more stages with smaller tensors, seeing as we have demonstrated that large tensors can be efficiently shared via a remote object storage system like AWS S3. While the focus of our work here has been on improving serverless function intercommunication, there would be great value in exploring improvements to partitioning algorithms.

## V. Conclusion

In conclusion, we propose a hybrid inter-function tensor sharing strategy to reduce end-to-end latency for partitioned deep neural networks deployed on serverless platforms. The key idea is to include smaller tensors directly in the payload and write larger tensors exceeding a configurable threshold to remote object storage, thereby minimizing the amount of data sent over the wire.

This strategy was motivated by our initial experiments, which demonstrated that using remote object storage as the main data transmission method had lower latencies for larger tensors and that certain DNN models indeed exchange tensors

exceeding 800 KB in size. The strategy was implemented on top of the Gillis system for deploying partitioned models on AWS Lambda. An experiment with image classification models demonstrated the hybrid approach decreases median inference latency by 17.3% for WRN50-3, the model with the largest shared tensors.

We believe this serves as an initial step toward optimizing communication for partitioned models on serverless platforms, particularly for very large DNN models. Exploring adjustments to partitioning algorithms and evaluating cost implications remain as future work. In the end, our findings suggest tensor size should influence the choice of sharing mechanism used by all systems that share tensors between serverless functions.

## REFERENCES


[1] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[2] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Optimizing inference serving on serverless platforms. *Proc. VLDB Endow.*, 15(10):2071–2084, June 2022.

[3] Bob Briscoe, Anna Brunstrom, Andreas Petlund, David Hayes, David Ros, Ing-Jyh Tsang, Stein Gjessing, Gorry Fairhurst, Carsten Griwodz, and Michael Welzl. Reducing internet latency: A survey of techniques and their merits. *IEEE Communications Surveys & Tutorials*, 18(3):2149–2196, 2016.

[4] Dheeraj Chahal, Ravi Ojha, Manju Ramesh, and Rekha Singhal. Migrating large deep learning models to serverless architecture. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 111–116, 2020.

[5] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. InferLine: Latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 477–491, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

[7] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. Scrooge: A cost-effective deep learning inference system. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 624–638, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 257–262, 2018.

[9] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. AMPS-Inf: Automatic model partitioning for serverless inference with cost efficiency. In *Proceedings of the 50th International Conference on Parallel Processing*, ICPP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. Astrea: Auto-serverless analytics towards cost-efficiency and qos-awareness. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3833–3849, 2022.

[11] Myung-Hyun Kim, Jaehak Lee, Heonchang Yu, and Eunyoung Lee. Improving memory utilization by sharing dnn models for serverless inference. In *2023 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–6, 2023.

[12] Yuepeng Li, Deze Zeng, Lin Gu, Kun Wang, and Song Guo. On the joint optimization of function assignment and communication scheduling toward performance efficient serverless edge computing. In *2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS)*, pages 1–9, 2022.

[13] Kunal Mahajan and Rumit Desai. Serving distributed inference deep learning models in serverless computing. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 109–111, 2022.

[14] Subin Park, Jaeghang Choi, and Kyungyong Lee. All-You-Can-Inference: Serverless dnn model inference suite. In *Proceedings of the Eighth International Workshop on Serverless Computing*, WoSC '22, page 1–6, New York, NY, USA, 2022. Association for Computing Machinery.

[15] Qiangyu Pei, Yongjie Yuan, Haichuan Hu, Qiong Chen, and Fangming Liu. AsyFunc: A high-performance and resource-efficient serverless inference system via asymmetric functions. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 324–340, New York, NY, USA, 2023. Association for Computing Machinery.

[16] Babak Rokh, Ali Azarpeyvand, and Alireza Khanteymoori. A comprehensive survey on model quantization for deep neural networks in image classification. *ACM Trans. Intell. Syst. Technol.*, 14(6), November 2023.

[17] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: A model-less and managed inference serving system, 2020.

[18] Hongjian Shi, Weichu Zheng, Zifei Liu, Ruhui Ma, and Haibing Guan. Automatic pipeline parallelism: A parallel inference framework for deep learning applications in 6g mobile communication systems. *IEEE Journal on Selected Areas in Communications*, 41(7):2041–2056, 2023.

[19] Zhucheng Tu, Mengping Li, and Jimmy Lin. Pay-per-request deployment of neural network models using serverless architectures. In Yang Liu, Tim Paek, and Manasi Patwardhan, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 6–10, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[20] Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings. SplitPlace: AI augmented splitting and placement of large-scale neural networks in mobile edge environments. *IEEE Transactions on Mobile Computing*, 22(9):5539–5554, 2023.

[21] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. Morphling: Fast, near-optimal auto-configuration for cloud-native model serving. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 639–653, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. Rise of the planet of serverless computing: A systematic review. *ACM Trans. Softw. Eng. Methodol.*, 32(5), July 2023.

[23] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: A native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 768–781, New York, NY, USA, 2022. Association for Computing Machinery.

[24] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 138–148, 2021.

[25] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 1049–1062, USA, 2019. USENIX Association.

[26] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. DeepThings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.