TrIMS: Transparent and Isolated Model Sharing for Low Latency Deep Learning Inference in Function-as-a-Service

Abdul Dakkak[1], Cheng Li[1], Simon Garcia de Gonzalo[1], Jinjun Xiong[2], and Wen-mei Hwu[3]

dakkak@illinois.edu, cli99@illinois.edu, grcdgnz2@illinois.edu, jinjun@us.ibm.com, w-hwu@illinois.edu

[1]Department of Computer Science , University of Illinois, Urbana-Champaign

[2]IBM Thomas J. Watson Research Center , Yorktown Heights, NY

[3]Department of Electrical and Computer Engineering , University of Illinois, Urbana-Champaign

***Abstract*—Deep neural networks (DNNs) have become core computation components within low latency Function as a Service (FaaS) prediction pipelines. Cloud computing, as the de-facto backbone of modern computing infrastructure, has to be able to handle user-defined FaaS pipelines containing diverse DNN inference workloads while maintaining isolation and latency guarantees with minimal resource waste. The current solution for guaranteeing isolation and latency within FaaS is inefficient. A major cause of the inefficiency is the need to move large amount of data within and across servers. We propose *TrIMS* as a novel solution to address this issue. *TrIMS* is a generic memory sharing technique that enables constant data to be shared across processes or containers while still maintaining isolation between users. *TrIMS* consists of a persistent model store across the GPU, CPU, local storage, and cloud storage hierarchy, an efficient resource management layer that provides isolation, and a succinct set of abstracts, application APIs, and container technologies for easy and transparent integration with FaaS, Deep Learning (DL) frameworks, and user code. We demonstrate our solution by interfacing *TrIMS* with the Apache MXNet framework and demonstrate up to 24× speedup in latency for image classification models, up to 210× speedup for large models, and up to 8× system throughput improvement.**

I. Introduction

Today, many business-logic and consumer applications rely on Deep Learning(DL) inferences as core components within their application pipelines. These pipelines tend to be deployed to the cloud through serverless computing, since they abstract away low-level details such as system setup and DevOps while providing isolation, decentralization, and scalability, all the while being more cost-effective than dedicated servers. User code which defines the pipeline (acting as glue code) is commonly deployed through Function as a Service (FaaS) [1], [2], [5], [7] onto the cloud and is made available through HTTP endpoints. Since FaaS executes arbitrary user code, the host system **must** execute code in isolation — through virtual machines (VMs) or containers.

While serverless is an emerging and compelling computing paradigm for event-driven cloud applications, use cases of the current FaaS offerings are limited. Currently, serverless functions run as short-lived VMs or containers, and thus are not ideal for long running jobs. FaaS functions are also

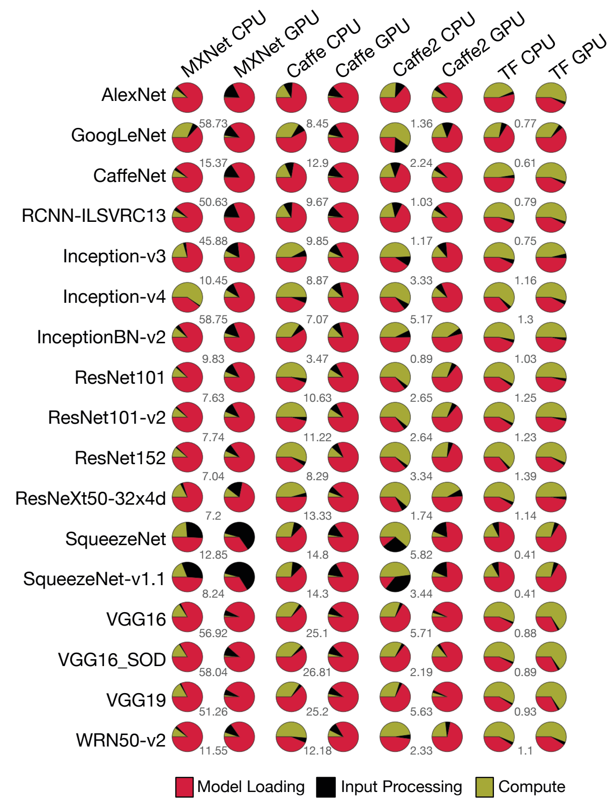

Figure 1: Percentage of time spent in model loading, inference computation, and image preprocessing for online DL inference ($batchsize = 1$) using CPU and GPU for MXNet, Caffe, Caffe2, and TensorFlow on an IBM S822LC with Pascal GPUs. The speedup of using GPU over CPU for inference compute is shown between the pie charts. Inference time for all frameworks is dominated by model loading except for small models. For TensorFlow, GPU initialization overhead impacts the end-to-end time and achieved speedup.

unable to work efficiently with data or distributed computing resources [19], [24], thus are not ideal for functions that require large data.

Recent work has proposed extensions to FaaS infrastructure to expand its usage within DL domains and facilitate

2159-6190/19/$31.00 ©2019 IEEE
DOI 10.1109/CLOUD.2019.00067

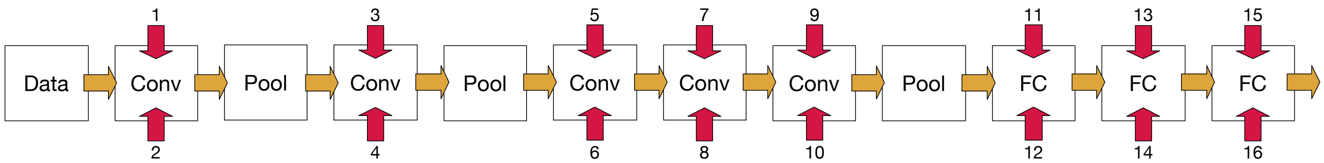

Figure 2: The DL inference graph for AlexNet [23]. The input dimensions and the memory footprint are shown in Table I.

it to leverage heterogeneous hardware. In [19], the authors advocate for code fluidity, where user functions is shipped to the data rather than the data being downloaded by the code. The advantages for this is three-fold. (a) It avoids the overhead of copying data over slow interconnects (such as network). (b) Leveraging heterogeneous hardware becomes attractive if data overhead is reduced and isolation is guaranteed. Finally, (c), since user functions which use the same data are routed to the same system, it exposes an opportunity for sharing constant data across functions.

Both (a) and (b) allow you to minimize data copy overhead and accelerate the computation using heterogeneous hardware. Yet, after removing the inter-node data copy overhead, intra-node data movement becomes a contributing factor to latency. This is even more true for heterogeneous devices, since data must be copied onto the device. This makes heterogeneous devices, such as GPUs, less attractive for accelerating latency-sensitive inference — even though they would offer a significant compute speed advantage, as shown in Figure 1. For (c), we observe that DL models are shared extensively across user pipelines. For example, Google reported that 41 natural translation models can accommodate over 75% of their translation requests in [4]. Because model parameters are constant, we can use data sharing across Faas functions to share DL models within a model catalog, hence eliminating the model loading overhead, decreasing the end-to-end latency, and reducing the memory footprint (since there is only one instance of a model in memory for many users) for DL inferences.

In this paper, we propose a **Tr**ansparent and **I**solated **M**odel **S**haring (*TrIMS*) scheme to leverage the data sharing opportunity introduced by collocating user code with model catalogs within FaaS — it minimizes model loading and data movement overhead while maintaining the isolation constraints and increasing hardware resource utilization. We also introduce the *TrIMS*'s model resource manager (MRM) layer which offers a multi-tiered cache for DL models to be shared across user FaaS functions. By decreasing model loading and data movement overhead, *TrIMS* decreases latency of end-to-end model inference, making inference on GPU a viable FaaS target. *TrIMS* also increases memory efficiency for cloud data centers while maintaining accuracy. In this paper we focus on online prediction within latency sensitive FaaS functions. Specifically, this paper makes the following contributions:

- We characterize the overhead for DL model inference across popular DL frameworks on both CPUs and GPUs and identify model loading as the bottleneck.
- We propose *TrIMS* to mitigate the model loading overhead faced by collocating user code with model catalogs within FaaS, and increase the hardware resource utilization by sharing DL models across all levels of the memory hierarchy in the cloud environment — GPU, CPU, local storage, and remote storage. To our knowledge, this work is the first to propose sharing DL models across isolated FaaS functions.
- We implement *TrIMS* within Apache MXNet [11] and evaluate the impact on GPU inference performance for a representative set of models and systems. We show that *TrIMS* provides $1.12\times$ – $24\times$ speedup on small (less than 600MB) models and $5\times$ – $210\times$ speedup on large (up to 6GB) models and is within 20% of ideal speedup (with ideal being that model loading and data movement taking no time), and gives $8\times$ system throughput improvement.
- *TrIMS* eliminates a substantial part of the non-compute components of the end-to-end latency, making DL model inference on GPU and other novel compute accelerators more viable.
- We architect *TrIMS* so that it can be easily integrated with existing FaaS systems and DL frameworks without user code changes. *TrIMS* is designed to be compatible with existing framework usage patterns, and requires minimal modifications for framework developers.
- While we use DL inference as the motivating application, *TrIMS* is not restricted to DL. *TrIMS* can be generalized to any application where one can share data across FaaS functions, be it a common database, knowledge base, or dataset.

II. Deep Learning Inference Overhead

A single DL inference is much less computationally intensive than training, making it more sensitive to the data loading and deserialization overhead. A DL inference compute graph is a DAG composed of a set of network layers. Each computational layer is parameterized through weights and constants. The model parameters along with the compute topology identify the model [1]. Each layer operator is a function of the incoming edges in the graph and the weights/constants. An inference pass iterates through the layers of a compute graph and applies the layer operators to its input. Figure 2 shows the inference compute graph for

[1]Throughout this paper, sharing a layer means that we are sharing both the weights and constants that parameterize the layer.

Index	Name	Dim	MF (MB)
1	conv1_bias	96	0.001
2	conv1_weight	$96 \times 3 \times 11 \times 11$	0.270
3	conv2_weight	$256 \times 48 \times 5 \times 5$	2.458
4	conv2_bias	256	0.002
5	conv3_weight	$384 \times 256 \times 3 \times 3$	7.078
6	conv3_bias	384	0.003
7	conv4_bias	384	0.003
8	conv4_weight	$384 \times 192 \times 3 \times 3$	5.3086
9	conv5_weight	$256 \times 192 \times 3 \times 3$	3.539
10	conv5_bias	256	0.002
11	fc6_bias	4096	0.033
12	fc6_weight	4096×9216	301.990
13	fc7_weight	4096×4096	134.218
14	fc7_bias	4096	0.033
15	fc8_bias	1000	0.008
16	fc8_weight	1000×4096	32.768

Table I: Memory footprint (MF) for each layer in Figure 2.

AlexNet [23] and Table I lists the dimension and memory footprint for each layer.

For GPUs, the compute graph and associated weights are loaded and copied to GPU memory ahead of the computation. Memory for intermediate layer outputs also need to be allocated. AlexNet, for example, requires 516*MB* of extra GPU memory to store the intermediate results during the inference process. These intermediate outputs are not constant and cannot be shared, since they depend on the user's input. However, layer weights are constant and can be shared across processes. For AlexNet, this results in sharing 238*MB* of constant data.

When compute is optimized, the overhead of model loading is magnified. Figure 1 shows that GPU outperforms the CPU in terms of compute, thus making model loading a bottleneck for end-to-end inference. Without data transfer overhead the NVIDIA Tesla V100 GPU using Tensor Cores can achieve $70\times$ higher throughput on CNNs and $130\times$ higher throughput on RNNs compared to a high-end CPU server [8]. Reducing the data movement overhead makes GPU a more appealing option for DL inference.

To mitigate the model loading overhead, cloud services and previous work [10], [14], [27] persist model catalogs in memory or perform inference in batches. These strategies require knowledge of the model requests, have potential resource waste since the system resources are persisted within processes for models even when they are not used, or increase the latency of requests if batching the inferences.

III. Current Prediction Pipelines in FaaS

Function as a Service (FaaS) is a cost-effective way for users to deploy functions or pipelines that are executed within the cloud. Users define prediction pipelines that use models they deployed or ones found within the model catalog. The pipelines are then mapped to a fabric of containers — used to maintain software stack separation, virtualize system resources, and provide isolation — that run on physical machines. Unlike traditional cloud execution, the functions executed in a FaaS are short lived and are priced on a per-invocation basis (with function execution time and resource utilization being the main cost factors). Because cloud providers use a per-API call and per-resource utilization price model, resource waste affects the cloud user's total cost of ownership.

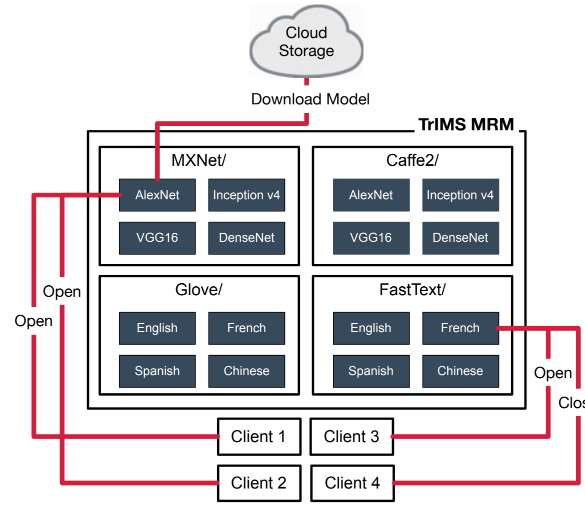

Figure 3: Multiple processes can perform IPC requests to the *TrIMS* Model Resource Manager (MRM) server; for example $Client_1$, $Client_2$, and $Client_3$ are performing an `Open` request, while $Client_4$ is performing a `Close` request. *TrIMS*'s MRM is responsible for loading and managing the placement of the models in GPU memory, CPU memory, or local disk.

In a cloud setting DL models are shared extensively across user functions, for example: between the 4 user functions shown in Figure 2. Based on this observation, we propose *TrIMS* to eliminate such model loading overhead and hardware resource waste, while maintaining resource utilization efficiency and decreasing inference latency in user processes. *TrIMS* achieves this by folding "private copies" of the model into a shared copy under the hood. This is performed by decoupling the model persistence from the user-code execution — enabling model sharing, isolation, and low latency inference.

IV. TrIMS Design

FaaS systems employ a server-worker architecture. The server (or controller) is a central server that orchestrates function execution on remote worker nodes running FaaS agents. Each FaaS agent within the FaaS system accepts user functions from the controller and launches containers on the worker node. A worker node is a multi-tenant system — accepting multiple functions from the controller — it is common for service providers to oversubscribe the worker nodes. *TrIMS* integrates with FaaS systems and consists of two components: ❶ a Model Resource Manager (MRM) that runs on the worker node and ❷ DL framework client that runs within the FaaS container launched by the FaaS agent.

The MRM manages the model resources resident in the system memory and abstracts away the model loading from framework clients. *TrIMS*-enabled frameworks communicate with MRM through inter-process communication (IPC) and maintain their existing APIs. Since *TrIMS* enabled framework follows the original framework's API and semantics — returning the same data structures as the unmodified framework — users can leverage *TrIMS* transparently without code modification.

A. *TrIMS Model Resource Manager (MRM)*

TrIMS's MRM is a model server daemon that performs model management and placement. MRM maintains a database of models, addressing them using namespaces, with framework as well as model name and version being used to distinguish frameworks and models. The MRM placement manager then maps the models into either GPU memory, CPU memory, local storage, or cloud storage. The four levels are analogous to the traditional CPU cache hierarchy. Because of this, we will simply refer to these four levels hierarchies as "cache" in the rest of this paper whenever there is no ambiguity.

For inter-process communication, *TrIMS* uses gRPC [17] to send and receive messages between the MRM and its clients. *TrIMS* leverages the CUDA runtime's *cudaIpc** to share GPU memory across processes. MRM abstracts away the model management, exposing two API functions to be used by the clients: `trims::open` and `trims::close` to load and close a model. MRM maintains a reference count for each model to determine the number of users currently using the shared model and it's up to the MRM to determine the behavior once the reference count is 0. The API is shown in Figure 4.

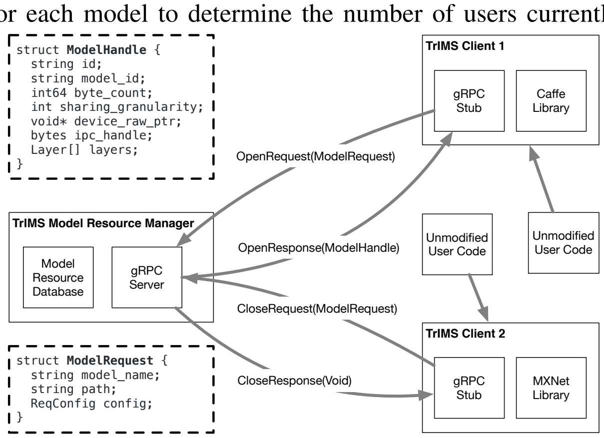

Figure 4: When user code loads a model using the original framework API, instead of loading the model directly from disk, the corresponding *TrIMS* client sends an `Open` request with `ModelRequest` structure to the MRM, and receives a response of type `ModelHandle`, from which it constructs the compute graph with model weights. When user code unloads a model, then instead of destroying the allocated memory, the *TrIMS* client sends out a `Close` request with `ModelHandle` and the MRM does the housekeeping.

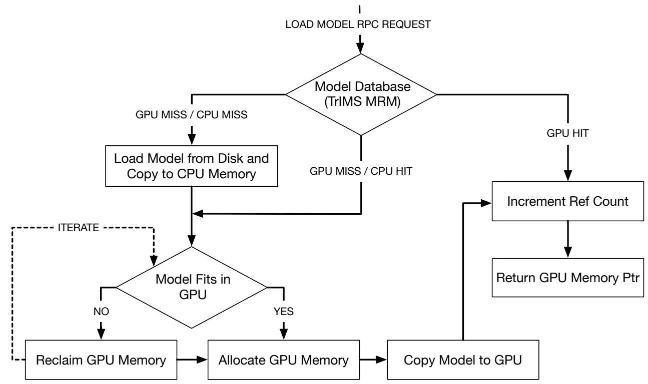

Figure 5: The logic for caching models on both GPU and CPU. The *TrIMS* client initiates the load model call to *TrIMS* MRM and gets back a pointer to GPU memory.

1) Loading Models: When loading a model, MRM performs shape inference on the model to estimate its memory footprint when running on GPU. Shape inference is a simple arithmetic computation performed by any framework to determine the tensor dimensions for a model. After shape inference, MRM follows the state diagram shown in Figure 5 and needs to handle three cases:

GPU cache hit — Model is persistent in GPU memory: MRM increments the model's reference count and creates a shared memory handle from the device memory owned by MRM. The handle is then returned to the framework client. Model eviction is triggered when the intermediate results for a model is greater than the available free memory.

GPU cache miss / CPU cache hit — model is persistent in CPU memory: The server queries the current memory utilization of the GPU to see if the model can be copied to GPU memory. If it can, then GPU memory is allocated and copied; if not, then some memory needs to be reclaimed — entering the memory reclamation procedure.

CPU and GPU cache miss — model is not persistent in memory: If the data is not on local storage, then MRM downloads the model from the cloud. If the data is on disk, then MRM loads the data from disk using the framework's serializer. Pinned memory is allocated on the CPU and the model weights is copied to it. MRM then follows the same logic as when the data is persistent in CPU memory.

2) Reclaiming Memory and Evicting Models: Memory reclamation is performed when the memory space for MRM at a specific cache level is full. Which model to evict to reclaim memory is determined by the eviction policy. *TrIMS* supports a pluggable set of common eviction policies such as least recently used(LRU) and least commonly used (LCU).

For the CPU and GPU level caches, model eviction cannot interfere with running user's code. For example, models within the MRM database are not cannot be reclaimed if they are in use; i.e. the reference count of a model is non-zero. Evicting models that is currently being used (effectively freeing GPU memory that's being used) causes undefined behavior in the user's code.

3) Unloading Models: When a *TrIMS* framework client unloads a model (or the user process exists), a model unload request is sent to MRM. MRM looks up the model in the database and decrements its reference count. By default MRM does not free resources for models that have a zero reference count (not currently used), but MRM can be configured to eagerly reclaim memory for these models.

B. TrIMS Frameworks

MRM can handle requests from multiple *TrIMS*-enabled frameworks, managing their weights (which have different data layouts) in separate namespaces. Shown in Figure 4, when a *TrIMS* framework performs a model load request, the framework's name and version are sent along with the request. The MRM can then perform the model unmarshalling from disk using the format supported by the framework.

To enable *TrIMS* in a framework, the functions to load and unload models need to be modified to perform *gRPC* requests to MRM. Since, each framework may have its own serialization format, support for the model format, to enable unmarshalling the data from disk to memory, needs to be added to MRM. With these changes, any type of network supported by the framework (CNN, RNN, etc.) and any compute pattern is automatically supported by *TrIMS*.

User application rewriting overhead: Since MRM does not modify the framework's API, code that is linked with a *TrIMS*-enabled framework does not require any change and *TrIMS* works with existing Python, Java, R, or code developed in other languages. This is an attractive feature, since the benefits of *TrIMS* can be leveraged by cloud provider transparently from the user.

Sharing Granularity: *TrIMS* supports fixed-size block, layer, and model level sharing granularity. Sub-model level sharing granularity is interesting when considering layers or memory across models. For example, models trained using transfer learning [33] share the frozen layer weights. Block level granularity can also be used to share fixed-size buffers.

Multi-GPU and Multi-Node Support: Multi-GPU is usually used when performing batched inference [9], [10]. *TrIMS* inherently supports the multi-GPUs by leveraging Unified Memory (UM) [3]. Support for Multi-GPU sharing can also be performed without relying on UM by making the *TrIMS* framework client query the device ID of the current GPU context when a model is loaded. The framework client can then send the device ID along with the request. *TrIMS* MRM would then load the model into the GPU with that device ID. When a request loads a model on a GPU and the requested model is persistent on another GPU, MRM will perform GPU peer-to-peer memory copy if supported.

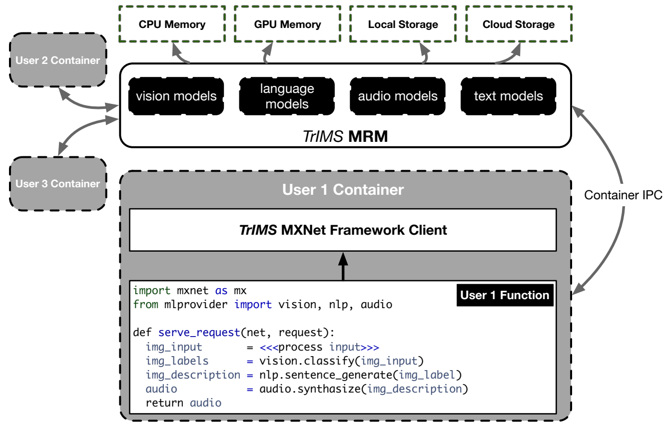

Figure 6: Cloud providers can use *TrIMS* MRM as a container plugin to provision running untrusted user functions while still leveraging model sharing. User code is executed within an isolated containers and can get the benefits of *TrIMS* without code modifications. Sharing occurs when the users utilize the same models as their peers.

Multiple independent instances of *TrIMS* MRM can be loaded for multi-node support and an off-the-shelf task scheduling and load balancing middleware can be used to route and load balance inference requests. *TrIMS* can be setup to advertise the models that have already been loaded by users and the current system load to the load balancer.

C. Inference Isolation and Fairness

To enable seamless container isolation, *TrIMS* provides a Docker [26] volume plugin that allows service providers to provision the container with a communication link to the *TrIMS* MRM. The *TrIMS* MRM process runs in the host system with a link for frameworks to communicate with it across container boundaries. Figure 6 shows how untrusted user code can be run on a multi-tenant system while maintaining isolation. The figure shows how users can use DL models to create an image to audio pipeline. The user uses the cloud provided vision, text, and audio models via a library that is part of a model catalog. All user code executes within a container that communicates with the MRM via the container's IPC mechanism.

V. Implementation

The experiments reported in this paper are based on an implementation of *TrIMS* on top of the Apache MXNet (github.com/rai-project/trims_mxnet) — a popular machine learning framework. The *TrIMS* MRM includes serialization code from MXNet to unmarshal MXNet models from disk. We also modify the MXNet framework to integrate it with *TrIMS* — keeping the MXNet APIs unchanged. Communication between the MXNet framework client and the MRM uses Google's gRPC [17] with the packets encoded using Protocol Buffers [6].

To validate the efficiency and generality of *TrIMS*, we follow a few principles throughout our implementation — even if disregarding some would have given us better speedup:

Backward Compatible: The implementation needs to work with the existing framework's code base and language bindings, i.e. we should be able to run preexisting MXNet code written in Python or Scala with no modifications.

Simple and Minimal: The implementation needs to be simple and minimize the modification to the framework code base. Our modifications adds only 1500 lines of code (less than 0.5% of the MXNet code base) to the framework (800 lines for the server and 700 lines for the client).

Configurable: The implementation has knobs to tweak everything from the eviction strategy of memory sharing, the amount of memory that can be used, whether to enable *TrIMS*, the levels of cache to enable, etc.

Fast, Concurrent and Scalable: We communicate using gRPC and use efficient data structures [20] for the MRM database to make the serving fast and concurrent. The memory sharing strategy in *TrIMS* is scalable and can handle large amount of load.

A. TrIMS Apache MXNet Framework

We implement *TrIMS* on top of the Apache MXNet framework by modifying the `MXPredCreate` and `MXPredFree` in the MXNet C predict API's. When *TrIMS* is enabled, `trims::open` and `trims::close` are invoked during the predictor creation and deletion.

Like most open-source DL Frameworks, MXNet is optimized for training and not inference. We apply a set of optimizations to the original MXNet to improve the inference latency. The optimizations avoid eager initialization of CUDA resources, remove cuDNN algorithm selection for backward propagation, and simplify the random resource generation. With our optimizations, MXNet is 6× faster for inference on average than the vanilla MXNet for the suite of models we use. We use the modified MXNet as our baseline during evaluation.

B. GPU Memory Sharing

We perform GPU memory sharing using the CUDA's `cudaIPC*` runtime functions. For Pre-Volta GPUs, the CUDA IPC mechanism utilizes CUDA MPS — an intermediate user process where the memory allocations are performed. This means that all CUDA operations end up serialized and executed within the same CUDA MPS context — enabling different processes to share the same GPU virtual address space (VAS). For Volta GPUs, NVIDIA introduced a new feature to allow contexts to share page-table mappings. This makes it possible for user processes to run using different contexts while still sharing memory. For CUDA 9.2, CUDA MPS is still invoked to keep shared allocations and communicate across them, but, with the exception of a handful of functions, most CUDA operations are performed without IPC communication.

Because sharing may serialize to use CUDA MPS, one slight disadvantage of CUDA IPC functions is that they have a measurable overhead. This can become a bottleneck. When sharing models at layer granularity, networks with large number of layers, such as *R*esNet269-v2, have high overhead. We remedy this by having enabling one to set the sharing granularity at groups of layer or whole models.

The CUDA IPC overhead is measurable, and we can quantify whether using *TrIMS* is beneficial statically using the empirical formula: $\rho = b \div q - n \times (o + s)$, where n is the number of objects to share (when the sharing granularity is at the model level, this value is 1; when the granularity is at the layer, this value is the number of layers); o is the overhead of sharing CUDA memory via CUDA IPC and s is the overhead of obtaining a CUDA device pointer from a shared CUDA IPC handle; b is the number of bytes the model occupies on disk; and q is the disk I/O bandwidth. These constants can be computed once at system startup and cached to be used by *TrIMS*. If ρ is positive, then its magnitude is correlated to the speedup one gets using *TrIMS*. This equation can be used within the *TrIMS* framework to determine at runtime whether to call *TrIMS* to share a model or not and at what granularity to share the model.

VI. Evaluation

We evaluate *TrIMS* on 3 systems (shown in Table II) using 37 (shown in Table III) pre-trained models and 8 large models (shown in Table IV). The systems selected represent different types of instances that are currently provisioned in the cloud. System 3 uses the NVLink bus [15], [32] which allows up to $35GB/s$ transfer between CPU and GPU. System 3 is used as proxy for understanding our proposed method's behavior on high end cloud instances and next generation interconnects currently being deployed on HPC and cloud systems [34]. Multi-GPU results are similar to the single-GPU results and are omitted for simplicity.

We used image processing models as a representative workload because these are currently the most plentiful in FaaS pipelines. *TrIMS* is agnostic to the compute patterns of a network and the analysis would apply to other types of networks such as: RNNs, word embeddings, or matrix factorization. The selected 37 pre-trained image processing models, shown in Table III, are based on their popularity in both research and usage. Some of the networks have variants. These are used to simulate user trained models — the same compute networks structure can have different weights. Large models are used to show how *TrIMS* performs with increasing model sizes.

Throughout this section we compare our performance within a FaaS setting against ideal (where the model loading and data movement takes no time, and is faster than model persistence) and use the end-to-end inference with model

Name	CPU	GPU	Memory	GPU Memory	Cached Reads	Buffered Disk Reads
System 1	Intel Core i9-7900X	TITAN Xp P110	32 GB	12 GB	8 GB/sec	193.30 MB/sec
System 2	Intel Xeon E5-2698 v4	Tesla V100-PCIE	256 GB	16 GB	10 GB/sec	421.30 MB/sec
System 3	IBM S822LC Power8 w/ NVLink	Tesla P100-SXM2	512 GB	16 GB	27 GB/sec	521.32 MB/sec

Table II: We evaluate *TrIMS* on 3 systems which represent both cloud offerings and consumer desktop system configurations currently used for DL inference. We use the Linux `hdparm` tool to measure the cached disk reads.

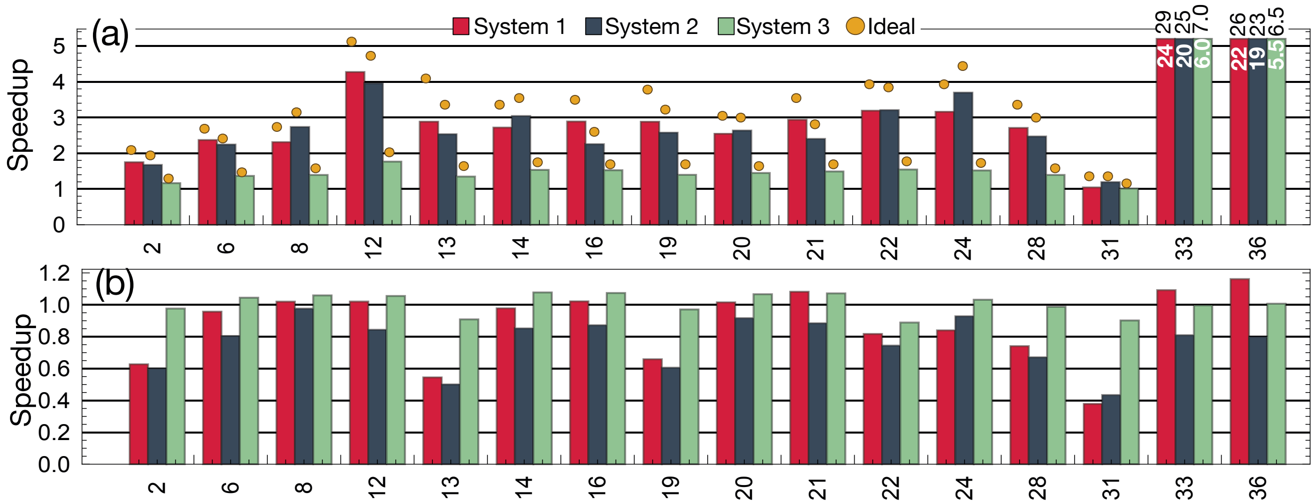

Figure 7: A representative sample of the models shown in Table III are chosen and are run on the systems in Table II to achieve (a) the best case end-to-end time — when the model has been pre-loaded in GPU memory — and (b) the worst case end-to-end time — when the model misses both the CPU and GPU persistence and needs to be loaded from disk. The speedups are normalized to end-to-end running time of the model without *TrIMS*. The yellow dots show the ideal speedup; the speedup achieved by removing any I/O and data-transfer overhead — keeping only the framework initialization and compute. For models 33 and 36, the achieved speedup is shown on the bar (white) and the ideal speedup is shown on top of the bar (black).

loading overhead as the base line, since that's what is currently employed by FaaS.

A. Latency Improvement

We measure the end-to-end inference of MXNet with and without *TrIMS* – omitting input processing time for clarity. Figure 7 shows the achieved speedup on a representative set of the models compared against MXNet that does not utilize *TrIMS*. We show two scenarios: (a) our best case (when there is a GPU cache hit) and (b) our worst case (when the cache misses both the CPU and GPU).

For best case analysis (Figure 7a), the server needs to create the CUDA IPC handles and the framework client needs to embed the GPU device pointers within the framework's container. This introduces overhead, however it is within 20% of the ideal — ideal defined as the time for inference where model loading or deserialization times set to zero. We see that latency speedup increases with model size, and is inversely proportional to the system's data movement bandwidth and model's compute complexity.

For small models, where the I/O overhead is very low, for example SqueezeNet (which has a 5*MB* memory footprint), we observe only marginal speedup (1.04×). These models are designed to have a small footprint — targeting edge devices — and are rarely used within the cloud. For state-of-the-art networks, such as VGG16-SOD, we observe 24× speedup on System 1. Even with fast disk and the NVLink interconnect, which mitigates I/O overhead by offering greater data movement bandwidth, System 3 achieves 6× speedup for VGG16-SOD.

For the worst case analysis (Figure 7b), the MRM needs to load the data from disk, persist the model on the CPU, copy the data to the GPU, and send the GPU memory handles to the client. Although we get a slow down, this case assumes there is no model sharing across pipelines, and therefore uncommon in cloud setting.

B. Speedup Breakdown

To understand where the new bottlenecks are for the inference using *TrIMS*, we look at System 3 where we achieve the lowest speedup and measure the (a) model initialization, (b) time to deserialize and copy the model data to GPU, (c) overhead introduced by *TrIMS* to share a model, and (d) time to perform inference computation. As can be seen in Figure 8, without using *TrIMS* an average of 86% of the time is spent loading and initializing the model while only 7% is spent performing computation. When using *TrIMS* we eliminate the model loading from disk and remove

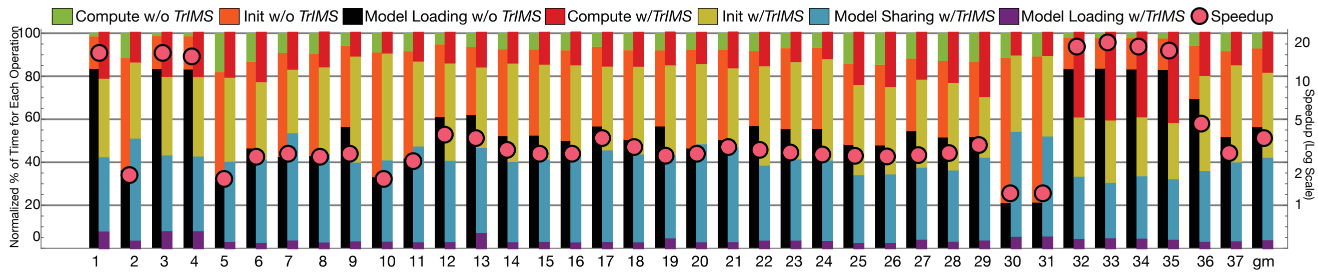

Figure 8: Detailed normalized time of operations with and without *TrIMS* on System 3 using the models in Table III. The duration for *TrIMS* is normalized to the end-to-end time of not using *TrIMS*. Compute is the time spent performing inference computation. Model initialization is the time spent setting up the CUDA contexts for the model, initializing the compute state. Model loading is the time spent copying the weights to GPU memory. Model sharing is the overhead introduced by using *TrIMS* and includes the gRPC communication and sharing GPU data using CUDA IPC. Through *TrIMS* we effectively eliminated model loading and data movement.

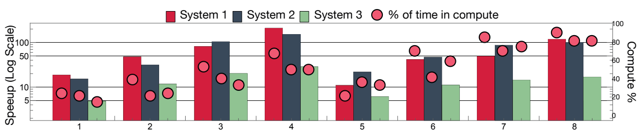

Figure 9: Large models in Table IV are run to achieve the best case end-to-end time — when the model has been pre-loaded in GPU memory. The speedups are normalized to end-to-end running time of the model without *TrIMS*. The red dots show the percentage of time for pure computation.

the need to perform GPU memory copies. Even though we introduce overhead, we still gain a 4.8× geometric mean speedup.

C. Large Model Evaluation

We evaluate our method using large models which are common for medical analysis, NLP, and time series modeling. We generated the large models by starting with the regular AlexNet and VGG16 networks, maintaining their compute graph, and rescaling the input dimensions to generate enlarged model. Table IV shows the 8 models selected for evaluation, their memory footprint, and their input sizes.

Figure 9 shows that by removing model loading overhead, inference on large models becomes compute bound and gives an advantage to faster GPUs. We also observe that *TrIMS* increases the memory efficiency of the GPU. Without *TrIMS*, two inferences using model 8 cannot be run concurrently, since they overrun the GPU memory. *TrIMS* avoids multiple private copies of the 6.4*GB* model on the same machine, enabling concurrent runs of large models.

D. Workload Modeling

Finally, we perform workload modeling to understand the behavior of *TrIMS* when used within a multi-tenant oversubscribed system senario. The workload is selected from the 37 small models shown in Table III following a Pareto distribution. Since the models cannot all be resident on the GPU at the same time — in total having 2× GPU memory footprint — the *TrIMS* MRM needs to exercise the

ID	Name	# Layers	ILS	MWMF
1	AlexNet [23]	16	516	238
2	GoogLeNet [30]	116	111	27
3	CaffeNet [23]	16	512	233
4	RCNN-ILSVRC13 [16]	16	479	221
5	DPN68 [12]	361	122	49
6	DPN92 [12]	481	340	145
7	Inception-v3 [31]	472	257	92
8	Inception-v4 [29]	747	399	164
9	InceptionBN-v2 [22]	416	313	129
10	InceptionBN-v3 [31]	416	142	44
11	Inception-ResNet-v2 [29]	1102	493	214
12	LocationNet [35]	514	666	285
13	NIN [25]	24	131	29
14	ResNet101 [18]	526	423	170
15	ResNet101-v2 [18]	522	428	171
16	ResNet152 [18]	777	548	231
17	ResNet152-11k [18]	769	721	311
18	ResNet152-v2 [18]	761	340	231
19	ResNet18-v2 [18]	99	154	45
20	ResNet200-v2 [18]	1009	589	248
21	ResNet269-v2 [18]	1346	889	391
22	ResNet34-v2 [18]	179	222	84
23	ResNet50 [18]	268	270	98
24	ResNet50-v2 [18]	259	275	98
25	ResNeXt101 [36]	526	375	170
26	ResNeXt101-32x4d [36]	522	378	170
27	ResNeXt26-32x4d [36]	147	147	59
28	ResNeXt50 [36]	271	222	96
29	ResNeXt50-32x4d [36]	267	224	96
30	SqueezeNet-v1.0 [21]	52	34	4.8
31	SqueezeNet-v1.1 [21]	52	28	4.8
32	VGG16 [28]	32	1228	528
33	VGG16-SOD [39]	32	1198	514
34	VGG16-SOS [38]	32	1195	513
35	VGG19 [28]	38	1270	549
36	WRN50-v2 [37]	267	758	264
37	Xception [13]	236	244	88

Table III: The small models are popular models used in literature and is used as proxy models that offer a wide variety of sizes and computational complexity. Image classification models are used since they are the most commonly used. Both internal layer sizes (ILS) and the model weights memory footprint (MWMF) are shown in megabytes. The number of models is chosen to be 2x larger than the available 16 GB memory on Systems 2 and 3.

model reclamation and eviction procedure. Because of limited space, we only present the results for the LRU eviction strategy, but our observations are valid for other eviction strategies. Note that persistent model serving platforms are unable to oversubscribe the system, and this is one of the benefits of *TrIMS*.

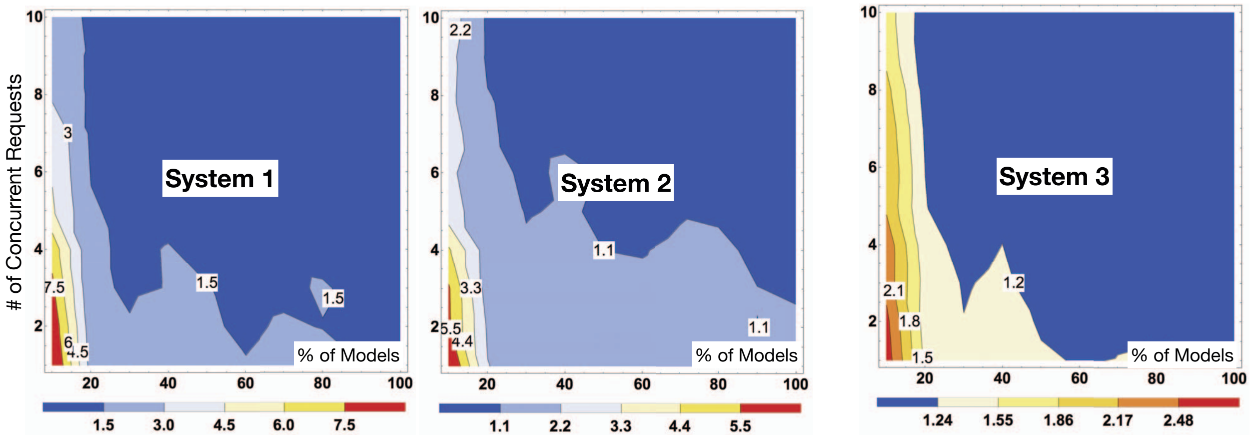

Figure 10: We vary the percentage of models run (from Table III) and we sample them following a Pareto distribution (with $X = 1$ and $l = 1$). We also vary the concurrency level (number of inferences performed concurrently) ranging it from 1 to 10. The iso-curves show the geometric mean of the speedups for Systems 1, 2, and 3.

ID	Name	Input Dims	MWMF
1	AlexNet-S1 [23]	227×227	238
2	AlexNet-S3 [23]	454×454	770
3	AlexNet-S3 [23]	681×681	1694
4	AlexNet-S4 [23]	908×908	3010
5	VGG16-S1 [28]	224×224	528
6	VGG16-S2 [28]	448×448	1704
7	VGG16-S3 [28]	672×672	3664
8	VGG16-S4 [28]	896×896	6408

Table IV: Large models were used to evaluate our method. The models were generated by taking AlexNet and VGG16 and scaling the number of input features. Large models arise in either medical image analysis, NLP, or time series analysis where down-sampling decreases the accuracy or the network requires a large window of features to give accurate results.

Figure 10 shows the level iso-efficiency curves for the geometric mean speedup [2] as we vary the concurrency level and number of models to run. We can see that even in an oversubscribed setting, we can still service 10 clients concurrently, reduce the overall batch execution time (by up to 8×), while incurring only a 20% latency penalty for each request. This slowdown is due to the cost of evicting models to accommodate the larger memory footprint, causing subsequent usage of the model to miss the GPU cache.

For all three systems, we can observe an over-subscription sweet spot, where the percent of models and number of the concurrent request can be increased while the batch execution is preserved to a speedup of 1×. All systems show a sweet spot when 40% of models are actively being requested. For system 1 and 3, the number of concurrent requests can be increased to 4, and system 2 the same number improves to 6. The difference in the over subscription sweet spot can be explained due to the different compute capabilities between the systems. System 1 and 3 are provision with Pascal generation GPUs while system 2 has the latest Volta generation. Essentially, because we are successful in moving the inference bottleneck from I/O to compute, the sweet spot is determined by the available computing resources. In practice, cloud providers can perform sensitivity analysis to determine the number of models hosted on each server and the number of concurrent requests to service based on the service's target requirements.

[2]We measure the speedup value using the geometric mean across the 99% latency speedup of each model.

VII. Conclusion

Collocating compute with model serving within FaaS overcomes the data transfer barrier but suffers from high intra-node data copy latency. We propose *TrIMS* to mitigate the major source of intra-node data copy latency — the model loading overhead — and enable building complex DL pipelines using latency sensitive FaaS functions. We do so by decoupling compute from model persistence and leveraging model sharing across user pipelines. *TrIMS* moves the bottleneck of DL model inference to compute, thus making GPU acceleration more appealing and making specialized novel inference architectures more tractable.

TrIMS was evaluated on three systems that represent current cloud system offerings. We used 45 DL models and show a speedup of up to 24× for small models and up to 210× for large models. When running concurrent inference, we can increase the overall system throughput by up to 8×. Our methodology, when applied to DL frameworks, offers advantages to both cloud providers and users. The isolation along with the significant memory reduction through model sharing enable cloud providers to over-provision hardware resources, thus decreasing the total cost of ownership. The benefits of *TrIMS* to the cloud providers can be passed down to the users in the form of latency or cost reduction.

TrIMS is a generic memory sharing technique that can be used when computation requires large number of constant parameters to be in situ on the CPU or GPU, while still maintaining isolation between users. As such, the proposed method can be easily generalized to any application or algorithm that spans multiple processes and requires large amount of constant data resources. While we motivated our work with deep learning, other types of applications such as image processing, physical simulation, or databases can benefit from our approach.

Acknowledgments

This work is supported by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM Cognitive Horizon Network.

References

[1] Amazon Lambda. http://aws.amazon.com/lambda. Accessed: 2018-8-04.

[2] Azure Functions. https://azure.microsoft.com/en-us/services/functions. Accessed: 2018-8-04.

[3] CUDA Unified Memory. https://devblogs.nvidia.com/tag/unified-memory. Accessed: 2018-8-04.

[4] Google Cloud AI. https://cloud.google.com/products/machine-learning. Accessed: 2018-8-04.

[5] Google Cloud Functions. https://cloud.google.com/functions. Accessed: 2018-8-04.

[6] Google Protocol Buffers. https://developers.google.com/protocol-buffers. Accessed: 2018-8-04.

[7] IBM OpenWhisk. http://www.ibm.com/cloud-computing/bluemix/openwhisk. Accessed: 2018-8-04.

[8] Nvidia Inference Technical Overview. https://images.nvidia.com/content/pdf/inference-technical-overview.pdf. Accessed: 2018-8-04.

[9] TensorFlow Serving. https://www.tensorflow.org/serving. Accessed: 2018-8-04.

[10] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395. ACM, 2017.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[12] Yunpeng Chen, Jianan Li, Huaxin Xiao, Xiaojie Jin, Shuicheng Yan, and Jiashi Feng. Dual path networks. In *Advances in Neural Information Processing Systems*, pages 4470–4478, 2017.

[13] François Chollet. Xception: Deep learning with depthwise separable convolutions. *arXiv preprint*, 2016.

[14] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627, 2017.

[15] Denis Foley and John Danskin. Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[16] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[17] gRPC. https://www.grpc.io. Accessed: 2018-8-04.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[19] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back, 2018.

[20] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *International Symposium on Distributed Computing*, pages 350–364. Springer, 2008.

[21] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[22] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[24] Philipp Leitner, Erik Wittern, Josef Spillner, and Waldemar Hummer. A mixed-method empirical study of function-as-a-service software development in industrial practice. *PeerJ PrePrints*, 6:e27005v1, 2018.

[25] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[26] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[27] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[28] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[29] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12, 2017.

[30] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. Going deeper with convolutions. Cvpr, 2015.

[31] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[32] Nathan R Tallent, Nitin A Gawande, Charles Siegel, Abhinav Vishnu, and Adolfy Hoisie. Evaluating on-node gpu interconnects for deep learning workloads. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 3–21. Springer, 2017.

[33] Lisa Torrey and Jude Shavlik. Transfer learning. *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, 1:242, 2009.

[34] RL Vogt, PR Kotta, and CN Meissner. Science and technology review march 2017. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2017.

[35] Tobias Weyand, Ilya Kostrikov, and James Philbin. Planet-photo geolocation with convolutional neural networks. In *European Conference on Computer Vision*, pages 37–55. Springer, 2016.

[36] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 5987–5995. IEEE, 2017.

[37] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[38] Jianming Zhang, Shugao Ma, Mehrnoosh Sameki, Stan Sclaroff, Margrit Betke, Zhe Lin, Xiaohui Shen, Brian Price, and Radomir Mech. Salient object subitizing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4045–4054, 2015.

[39] Jianming Zhang, Stan Sclaroff, Zhe Lin, Xiaohui Shen, Brian Price, and Radomir Mech. Unconstrained salient object detection via proposal subset optimization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5733–5742, 2016.