center for
cognitive computing
systems research

IBM | ILLINOIS

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

Abdul Dakkak (UIUC), Cheng Li (UIUC), Simon Garcia de Gonzalo (UIUC), Jinjun Xiong (IBM Research), Wen-mei Hwu (UIUC)

Computer Science Department — University of Illinois at Urbana-Champaign
IBM-ILLINOIS Center for Cognitive Computing Systems Research
c3sr.com

Motivation

Motivation

- Inference is latency sensitive
- A single feed forward pass through the DL graph
- Each layer operator is a function of the incoming edges in the graph and the weights/constants
- In the long run, inference is more compute expensive than training
- Layer weights are constant and can be shared across processes

AlexNet Model

TABLE I
MEMORY FOOTPRINT FOR EACH LAYER IN FIGURE 2.

Index	Name	Dimensions	Memory Footprint (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

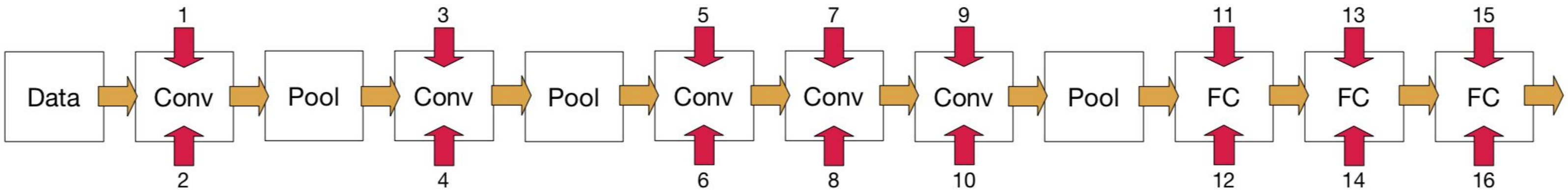
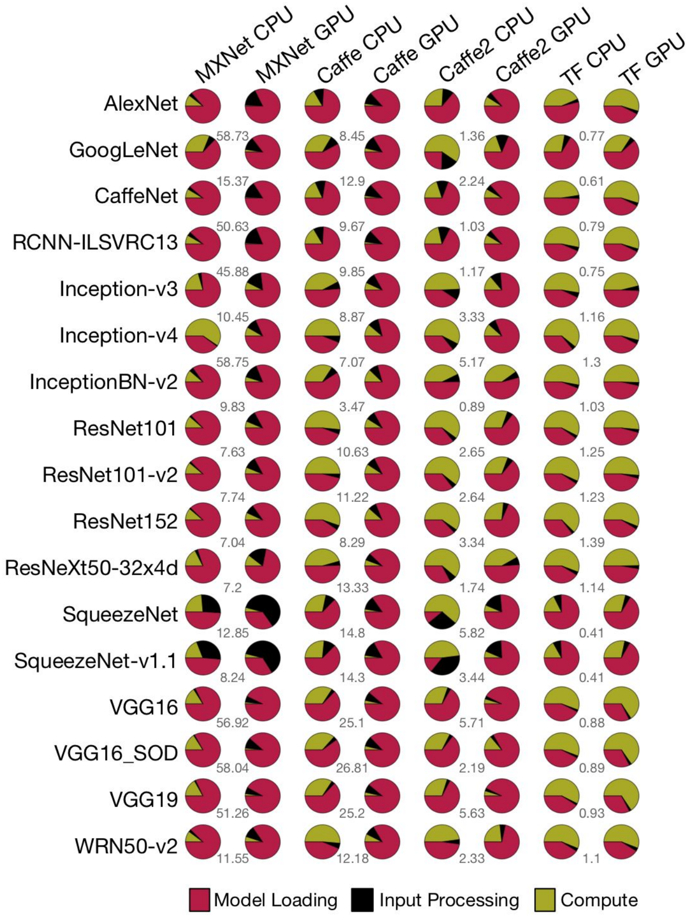


Fig. 2. The DL inference graph for AlexNet [18]. The input dimensions and the number of bytes required by each layer is shown in Table I.

Motivation

- Model loading is the bottleneck in end-to-end Deep Learning (DL) inference
- Current model serving solutions are suboptimal in terms of latency and resource utilization
- DL models are shared extensively across user pipelines
- Want to decouple model parameters persistence from the inference compute



Use Case

Current Model Serving

Model Artifacts are stored in the cloud (e.g. AWS S3)

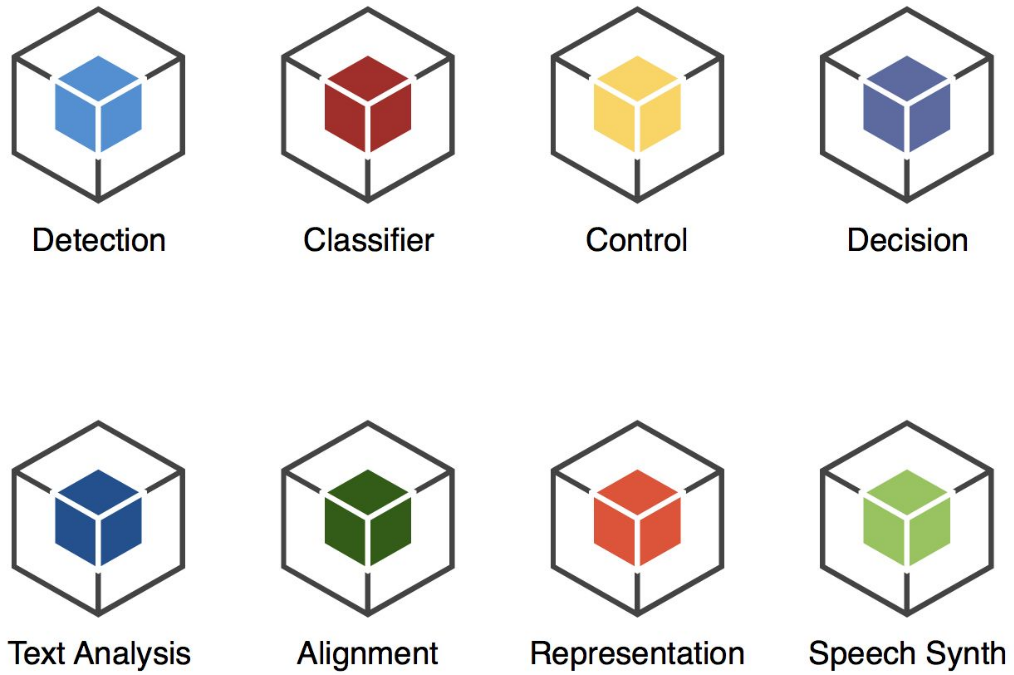
- Users either load models in their code (Suffer from model loading overhead)
- Leverage the APIs exposed by some remote inference server (persists the model inference process, wasting resources if not used)

Current Practice

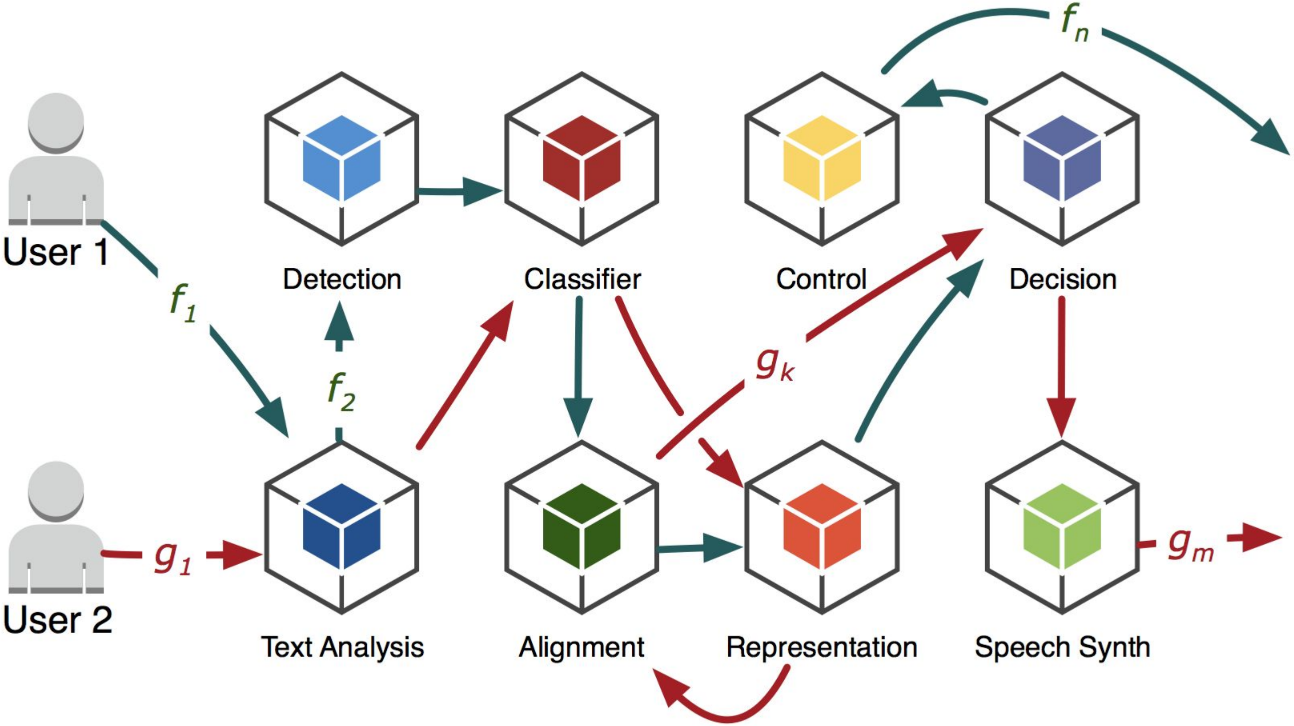
Model Artifacts are stored in the cloud (e.g. AWS S3)

- Users either load models in their code (Suffer from model loading overhead)
- Leverage the APIs exposed by some remote inference server (persists the model inference process, wasting resources if not used)

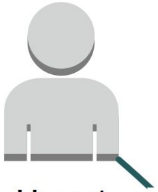
Tasks are Shared Extensively in the Cloud



Tasks are Shared Extensively in the Cloud

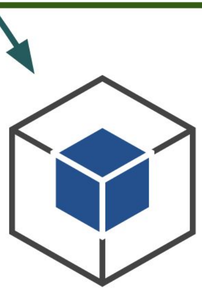


Tasks are Shared Extensively in the Cloud



$f_1 =$

```
stop_words = read('stop_words.txt').split('\n')
def process(input):
   lines = input.split('\n')
   lines = [word for word in lines if !is_stop_word(word)]
   return lines.join('\n')
def is_stop_word(word):
   return word in stop_words
```



TrIMS Design

TrIMS Design

Two components:

- Model Resource Manager (MRM)
- framework clients

Collocate with the user process

Models from different frameworks are managed in separate namespaces

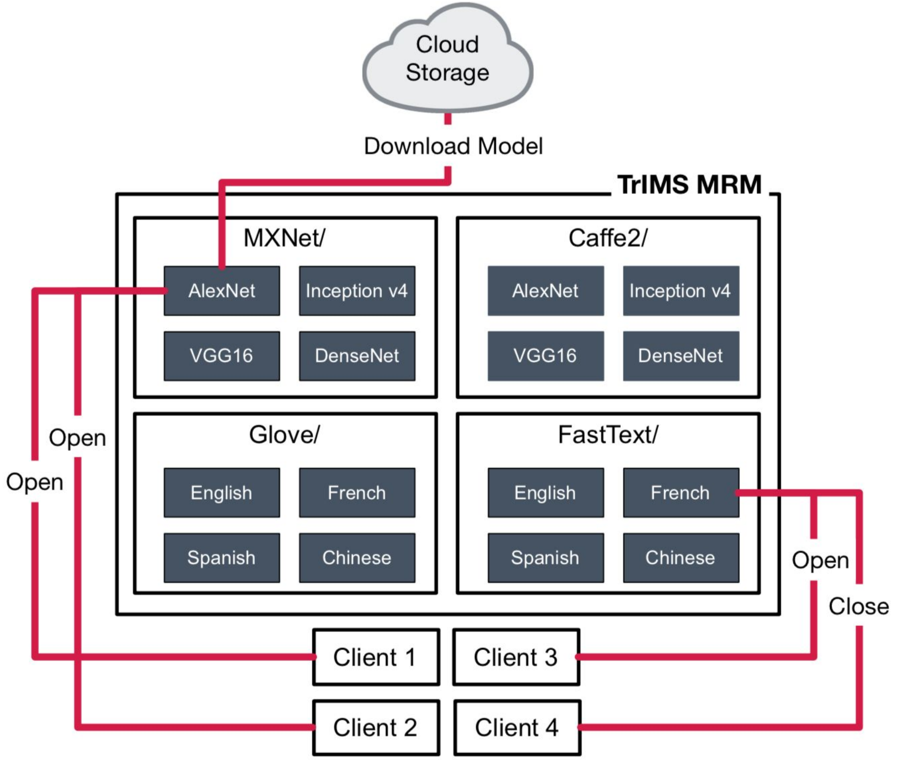


Fig. 4. Multiple processes can perform IPC requests to the *TrIMS* Model Resource Manager (MRM) server; for example *Client*$_1$, *Client*$_2$, and *Client*$_3$ are performing an `Open` request, while *Client*$_4$ is performing a `Close` request. *TrIMS*'s MRM is responsible for loading and managing the placement of the models in GPU memory, CPU memory, or local disk.

TrIMS Model Resource Manager

- gRPC for inter-process communication
- cudaIpc* to share GPU memory across processes

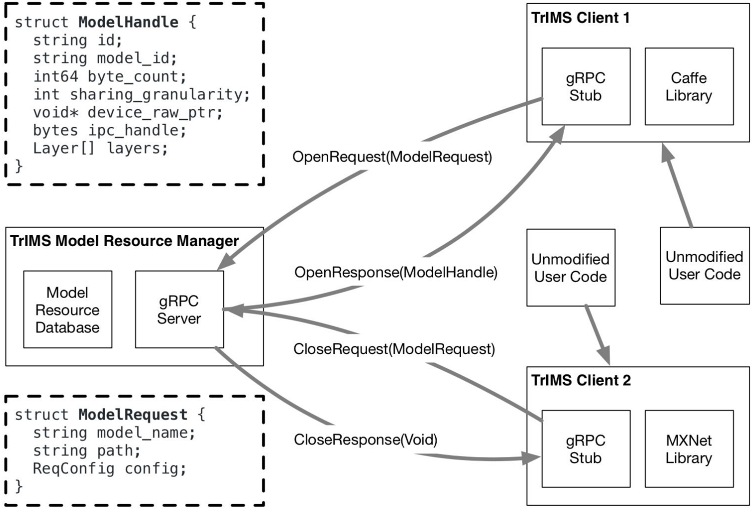


Fig. 5. When user code loads a model using the original framework API, instead of loading the model directly from disk, the corresponding *TrIMS* client sends an `Open` request with `ModelRequest` structure to *TrIMS* MRM, and receives a response of type `ModelHandle`, from which it constructs the compute graph with model weights. When user code unloads a model using the original framework API, instead of directly destroying the allocated memory, the *TrIMS* client sends out a `Close` request with `ModelHandle` and *TrIMS* MRM does the housekeeping.

TrIMS Model Resource Manager

- Maps the models into GPU memory, CPU memory, local storage, cloud storage
- Four-level “cache”
- When a cache level is full, reclaim memory and evict models

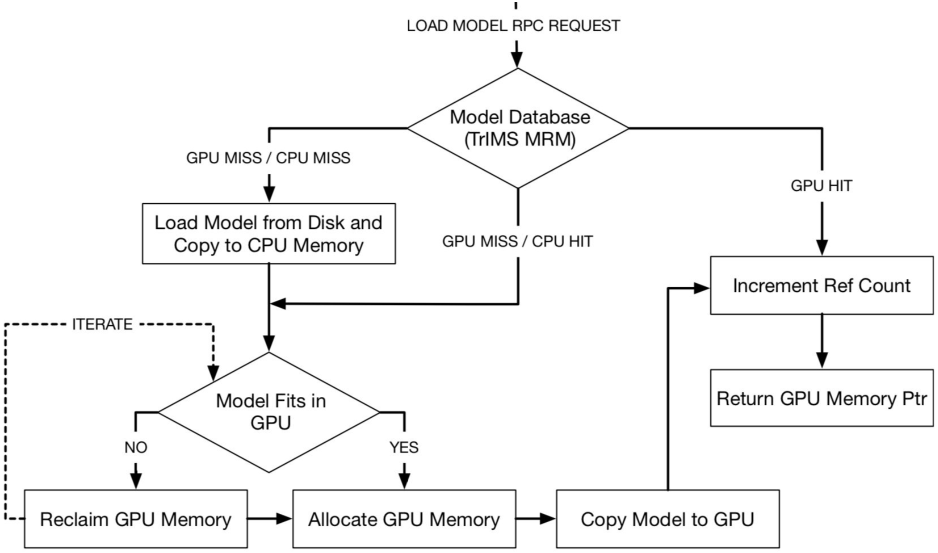


Fig. 6. The logic for caching models on both GPU and CPU. The *TrIMS* client initiates the load model call to *TrIMS* MRM and gets back a pointer to GPU memory.

Other Design Philosophies

- User application rewriting overhead
  - None
- Sharing Granularity
  - Model, layer or block
- Multi-GPU and Multi-Node Support
  - Yes
- Inference Isolation and Fairness
  - Guaranteed

Inference Isolation

- User codes run in isolation (separate processes or containers)
- Self-contained
  - Crash or error
  - No interference
- Cloud providers have fine grained control over each process (or container)

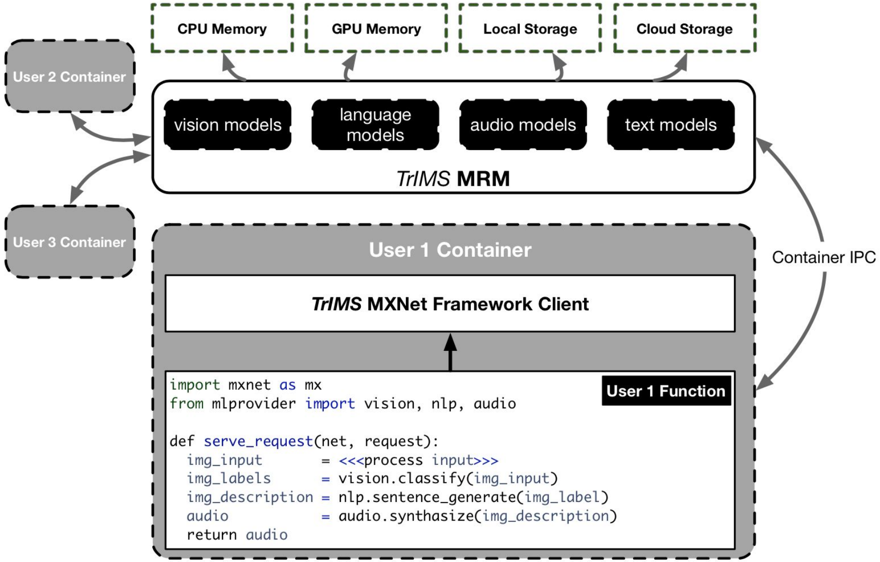
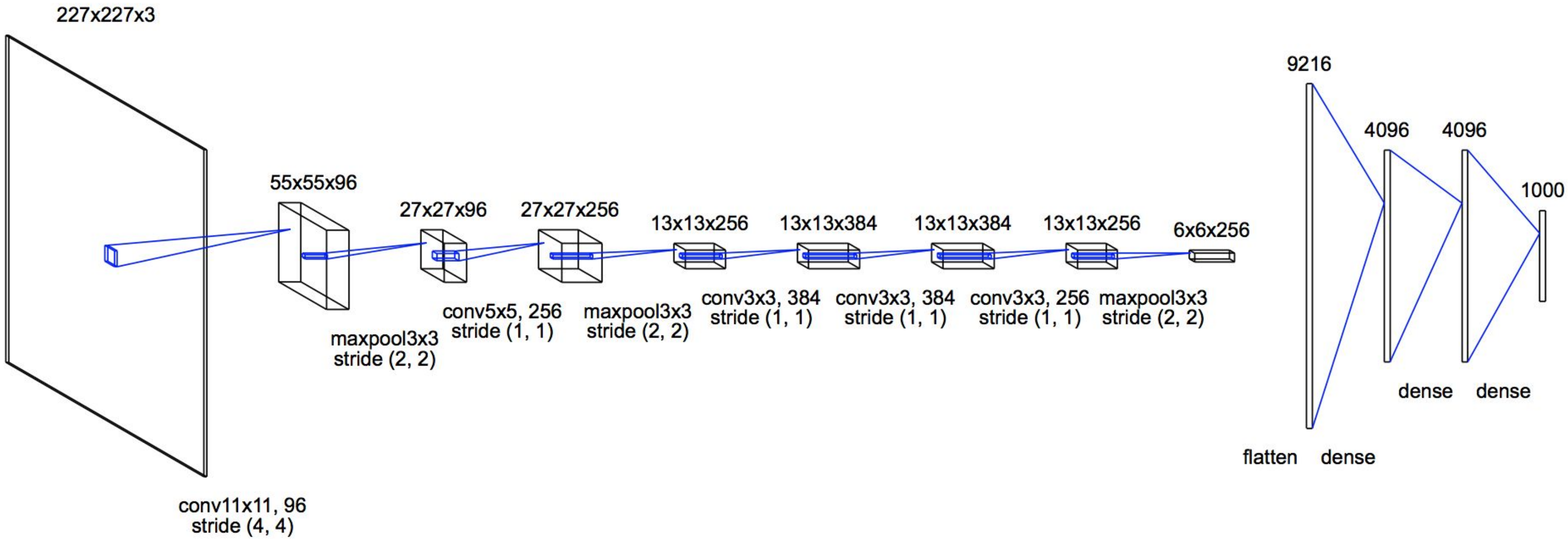


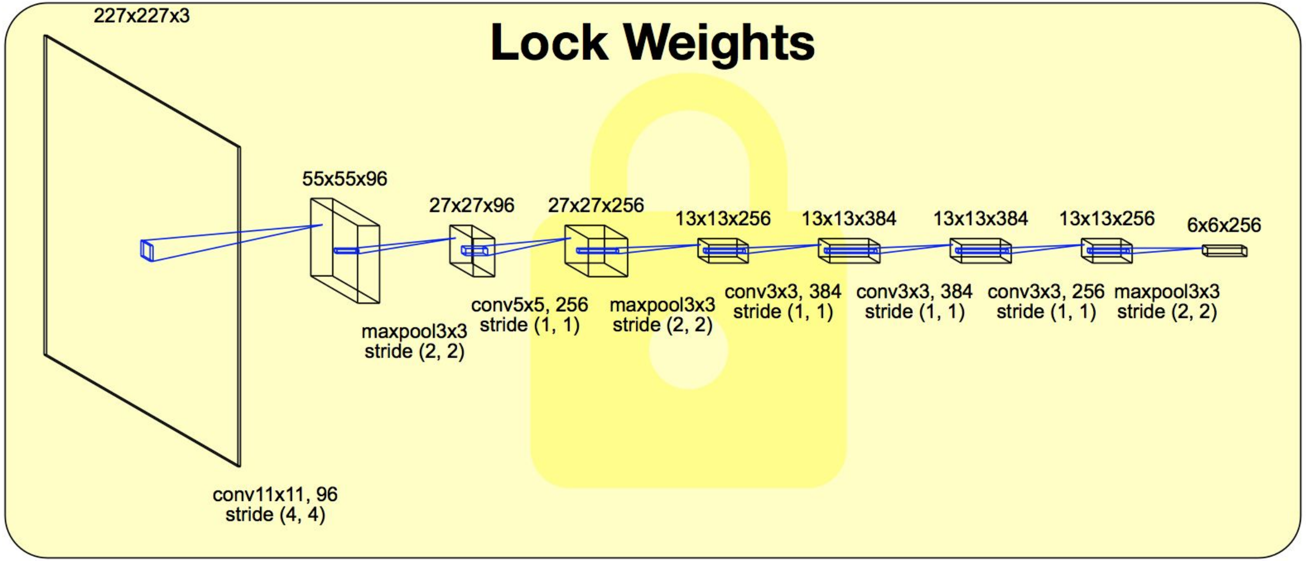
Fig. 7. Cloud providers can use *TrIMS* MRM as a container plugin to provision running untrusted user functions while still leveraging model sharing. User code is executed within an isolated containers and can get the benefits of *TrIMS* without code modifications. Sharing occurs when the users utilize the same models as their peers — which is not uncommon in cloud settings using cloud provided APIs.

Excessive Sharing due to Transfer Learning

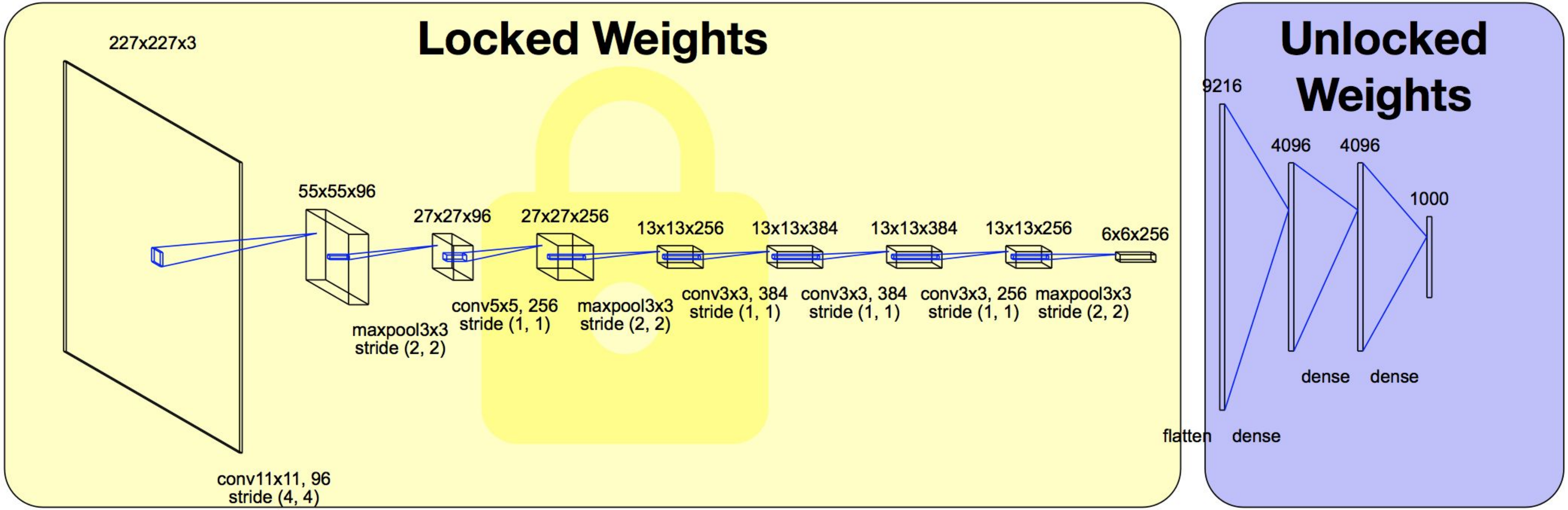
Sharing Granularities



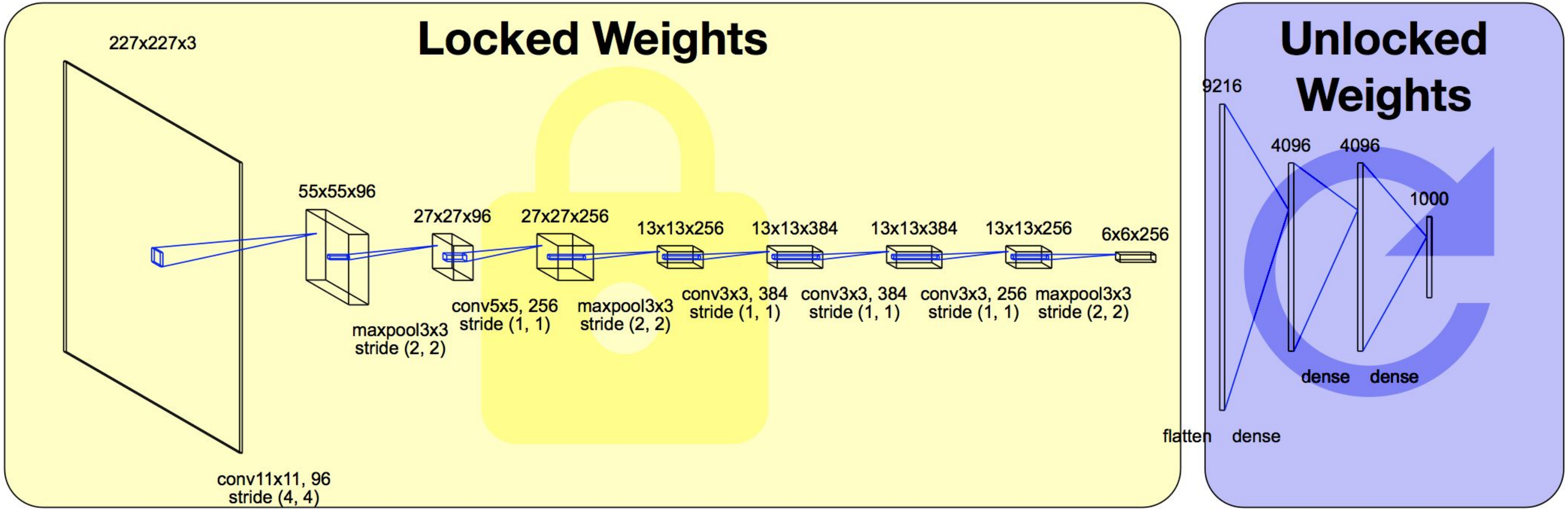
Sharing Granularities



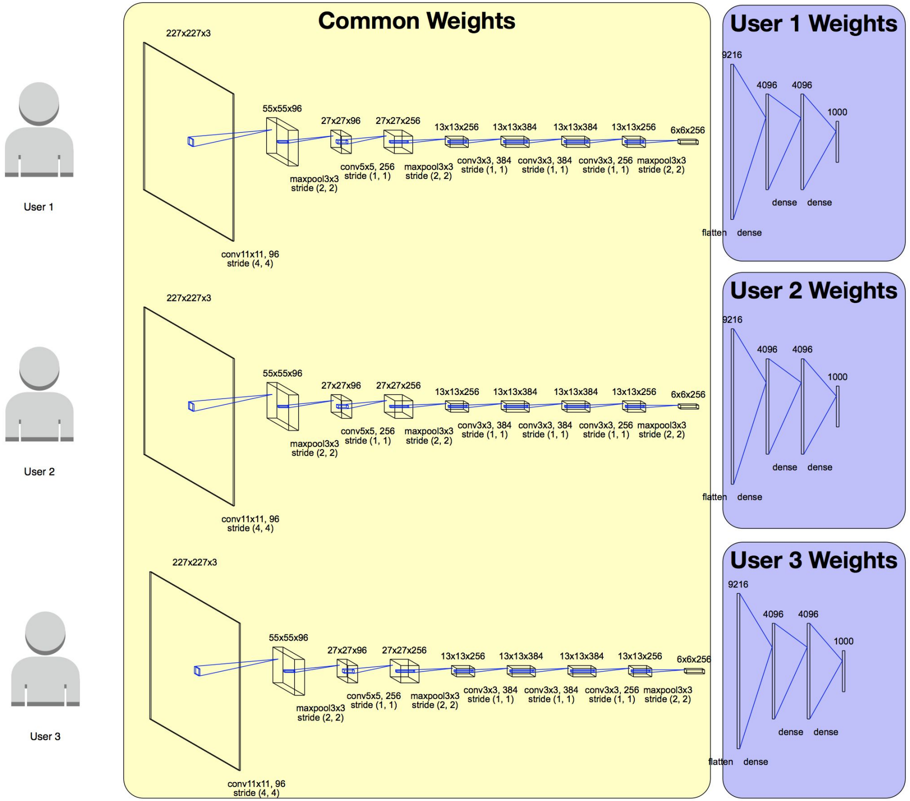
Sharing Granularities



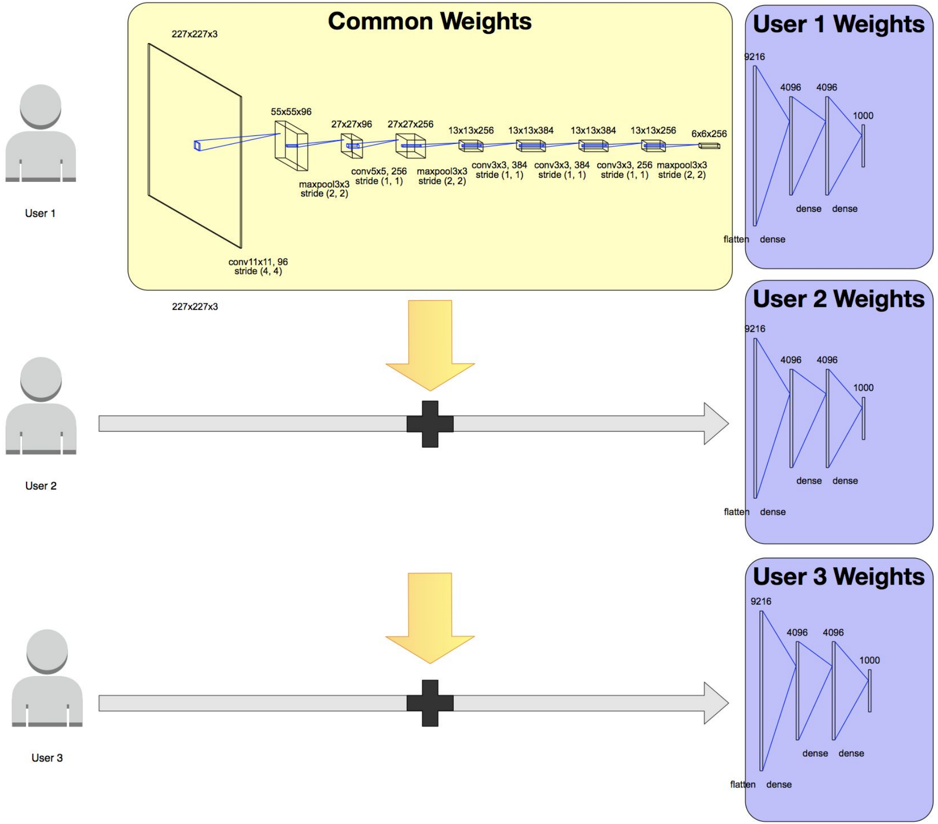
Sharing Granularities



Sharing Granularities



Sharing Granularities



Evaluation

Evaluation Setup

3 systems, 37 pre-trained popular models and 8 large models

Name	System1	System2	System3
CPU	Intel Core i9-7900X	Intel Xeon E5-2698 v4	IBM S822LC Power8 with NVLink
GPU	TITAN Xp P110	Tesla V100-PCIE	Tesla P100-SXM2
Memory	32 GB	256 GB	512 GB
GPU Memory	12 GB	16 GB	16 GB
Cached Reads	8 GB/sec	10 GB/sec	27 GB/sec
Buffered Disk Reads	193.30 MB/sec	421.30 MB/sec	521.32 MB/sec

ID	Name	# Layers	ILS	MWMF
1	AlexNet [18]	16	516	238
2	GoogLeNet [31]	116	111	27
3	CaffeNet [18]	16	512	233
4	RCNN-ILSVRC13 [32]	16	479	221
5	DPN68 [33]	361	122	49
6	DPN92 [33]	481	340	145
7	Inception-v3 [34]	472	257	92
8	Inception-v4 [35]	747	399	164
9	InceptionBN-v2 [36]	416	313	129
10	InceptionBN-v3 [34]	416	142	44
11	Inception-ResNet-v2 [35]	1102	493	214
12	LocationNet [37]	514	666	285
13	NIN [38]	24	131	29
14	ResNet101 [39]	526	423	170
15	ResNet101-v2 [39]	522	428	171
16	ResNet152 [39]	777	548	231
17	ResNet152-11k [39]	769	721	311
18	ResNet152-v2 [39]	761	340	231
19	ResNet18-v2 [39]	99	154	45
20	ResNet200-v2 [39]	1009	589	248
21	ResNet269-v2 [39]	1346	889	391
22	ResNet34-v2 [39]	179	222	84
23	ResNet50 [39]	268	270	98
24	ResNet50-v2 [39]	259	275	98
25	ResNeXt101 [40]	526	375	170
26	ResNeXt101-32x4d [40]	522	378	170
27	ResNeXt26-32x4d [40]	147	147	59
28	ResNeXt50 [40]	271	222	96
29	ResNeXt50-32x4d [40]	267	224	96
30	SqueezeNet-v1.0 [41]	52	34	4.8
31	SqueezeNet-v1.1 [41]	52	28	4.8
32	VGG16 [42]	32	1228	528
33	VGG16-SOD [43]	32	1198	514
34	VGG16-SOS [44]	32	1195	513
35	VGG19 [42]	38	1270	549
36	WRN50-v2 [45]	267	758	264
37	Xception [46]	236	244	88

Results

Latency improvement on the end-to-end inference

- Up to 24X and 4.8X geomean speedup

Timing breakdown

- Model loading is no longer the bottleneck

Inference Latency

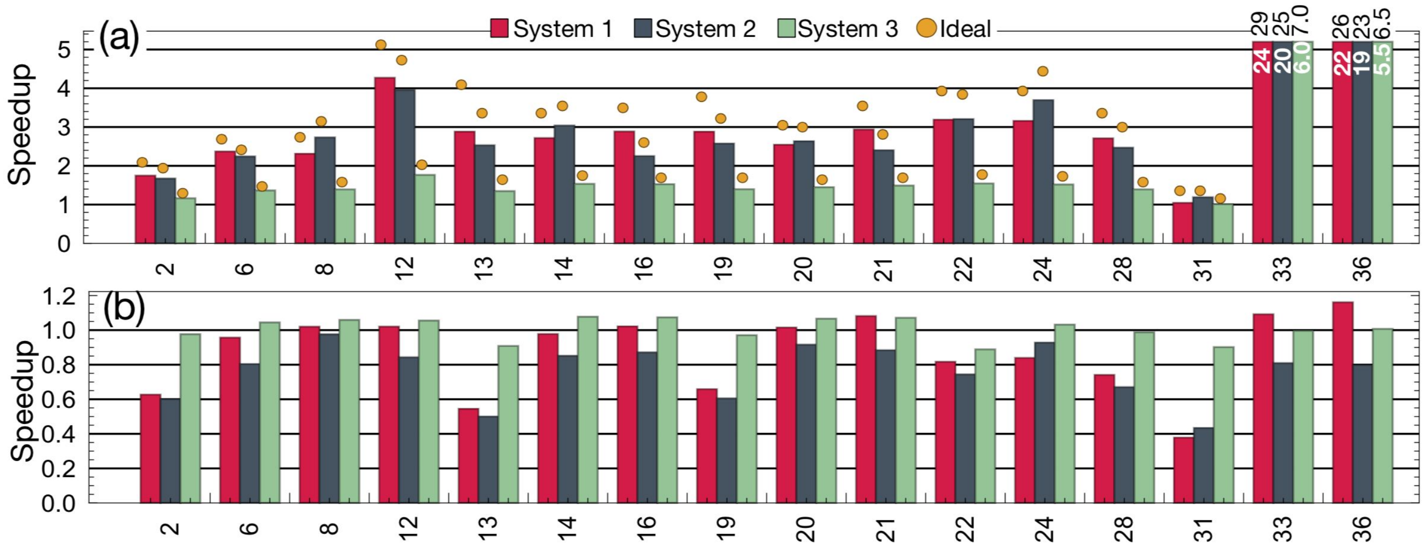


Fig. 8. A representative sample of the models shown in Table III are chosen and are run on the systems in Table II to achieve (a) the best case end-to-end time — when the model has been pre-loaded in GPU memory — and (b) the worst case end-to-end time — when the model misses both the CPU and GPU persistence 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).

Latency Breakdown

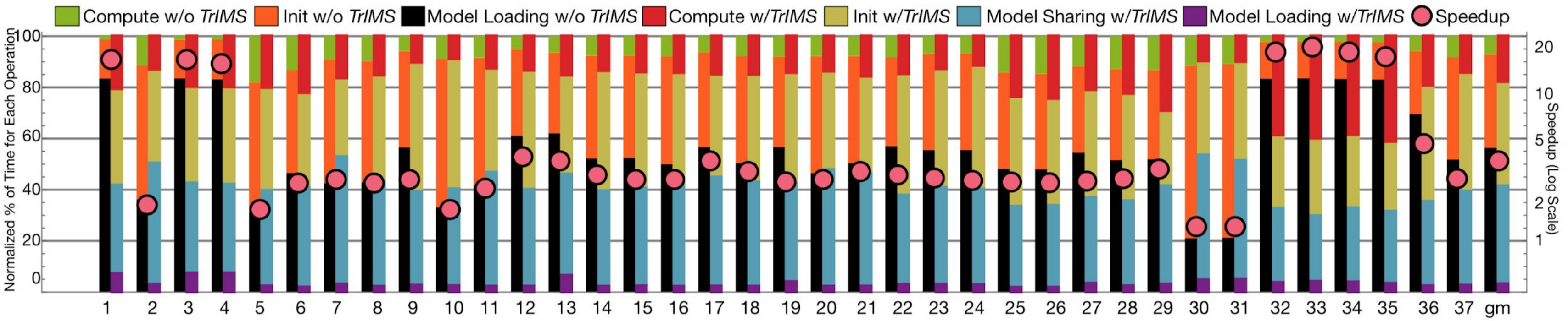


Fig. 9. Detailed normalized times of operations with and without *TrIMS* on System 3 using the models in Table III. The duration for *TrIMS* is normalized to the end-to-end time of not using *TrIMS*. Model initialization is the time spent setting up the CUDA contexts for the model, initializing the the compute state, and (in the case of not using *TrIMS*) copying the weights to GPU memory. Compute is the time spent performing inference computation. 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.

Large Models

Large Models

- Common for medical image analysis, NLP, time series modeling, etc.
- Either input is large, or want a large window of memory

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.

ID	Name	Input Dimensions	MWMF
1	AlexNet-S1 [18]	227×227	238
2	AlexNet-S3 [18]	454×454	770
3	AlexNet-S3 [18]	681×681	1694
4	AlexNet-S4 [18]	908×908	3010
5	VGG16-S1 [42]	224×224	528
6	VGG16-S2 [42]	448×448	1704
7	VGG16-S3 [42]	672×672	3664
8	VGG16-S4 [42]	896×896	6408

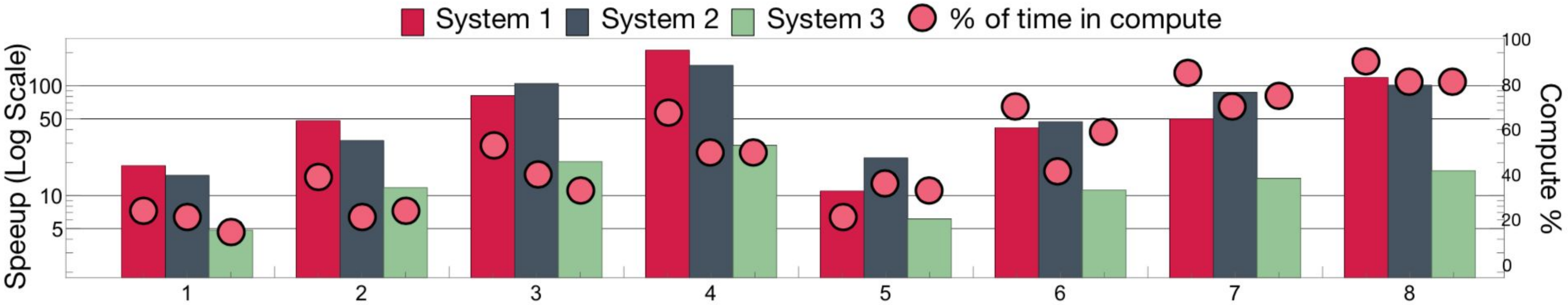


Fig. 10. large models in Table IV are run to achieve the best case end-to-end time — when the model has been pre-loaded in GPU memory. The speedups are normalized to end-to-end running time of the model without *TrIMS*. The red dots show the percentage of time spent performing the compute. We see linear speedup for scaling, until the inference becomes compute bound.

Workload Analysis

Workload Modeling

To understand the behavior of TrIMS on multi-tenant oversubscribed system

Workload is selected from the 37 models following a Pareto distribution

Design space of concurrency level, number of models to run and MRM configurations on a system

Improve the overall batch execution time => throughput

Workload Modeling

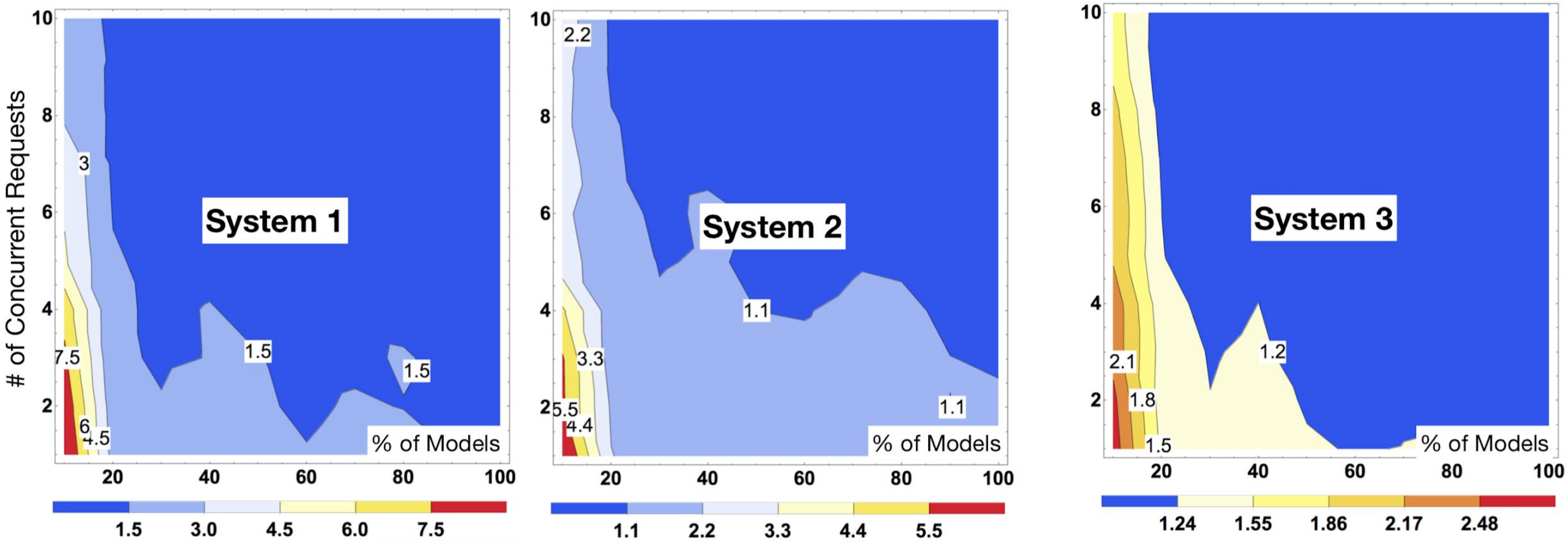


Fig. 11. We vary the percentage of number models (from Table III) run and we select from them following Pareto distribution (with $X = 1$ and $l = 1$). We Also vary the concurrency level ranging from 1 to 10. The geometric mean of the speedups is shown for both System 1 and 2.

Conclusion

Conclusion

- We showed how to remove model loading overhead from DL inference
  - Enabling more novel compute acceleration and optimizations
  - Over Provisioning of resources in cloud setting
- Our technique is
  - Transparent to the user
  - Maintains isolation for security
  - Scalable for the cloud provider
  - Reduces cost by improving latency and throughput while decreasing resource waste

github.com/rai-project/trims_mxnet
github.com/rai-project/trims-tools