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

Abdul Dakkak
Department of Computer Science
University of Illinois, Urbana-Champaign
Champaign, IL 61820
dakkak@illinois.edu

Cheng Li
Department of Computer Science
University of Illinois, Urbana-Champaign
Champaign, IL 61820
cli99@illinois.edu

Simon Garcia de Gonzalo
Department of Computer Science
University of Illinois, Urbana-Champaign
Champaign, IL 61820
grcdgnz2@illinois.edu

Jinjun Xiong
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598
jinjun@us.ibm.com

Wen-mei Hwu
Department of Electrical and Computer Engineering
University of Illinois, Urbana-Champaign
Champaign, IL 61820
w-hwu@illinois.edu

Abstract

Deep neural networks (DNNs) have become pervasive within low latency Function as a Service (FaaS) prediction pipelines, but suffers from two major sources of latency overhead: 1) the round-trip network latency between FaaS container and a remote model serving process; 2) Deep Learning (DL) framework runtime instantiation and model loading from storage to CPU or GPU memory. While models servers process solves the latter, they do so by eternally persisting models within memory — introduces resource waste and increases cost. With FaaS environments, models are frequently shared: image recognition, object detection, NLP, and speech synthesis for example. We propose TrIMS, a multi-tier software caching layer on top of FaaS worker machines to solve this problem. Our solution consists of a managing model within caches that span GPUs, CPUs, local and cloud storage through a resource management service. This enables sharing models across user processes within a system while guaranteeing isolation, a succinct set of APIs and container technologies for easy and transparent integration with FaaS, DL frameworks and user code. Moreover, we show that by adopting this technique, we are able to oversubscribe the system without degrading the baseline latency. We evaluate our solution by interfacing TrIMS with the Apache MXNet framework and demonstrate up to 24× speedup in latency for image classification models.

Preprint. Work in progress.

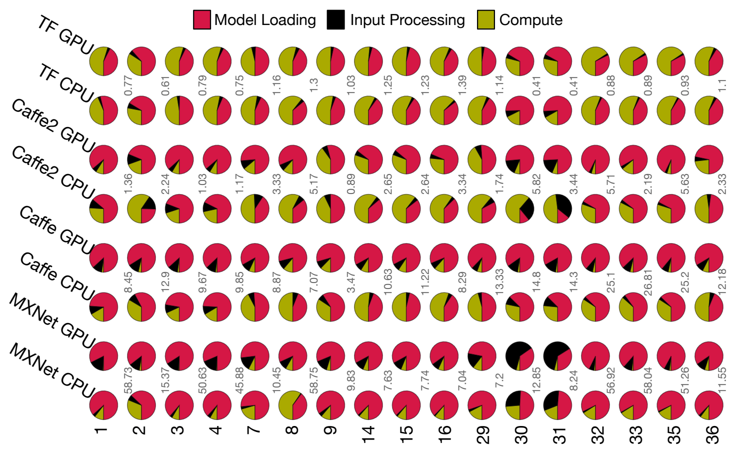


Figure 1: Percentage of time spent in model (with ids corresponding to Table 2) loading, inference computation, and image preprocessing for "cold start" online DL inference ($batchsize = 1$) using CPU and GPU for MXNet, Caffe, Caffe2, and TensorFlow on an IBM S822LC with Pascal GPUs. The speedup of using GPU over CPU for the inference compute alone is shown between the pie charts. Inference time for all frameworks is dominated by model loading except for small models, such as SqueezeNet, where the model size is a few megabytes. For TensorFlow, high GPU initialization overhead impacts the end-to-end time and the achieved speedup.

1 Introduction

Today, many business-logic and consumer applications rely on DL inference as core components within their pipelines. These pipelines tend to be deployed to the cloud through Function as a Service (FaaS) platforms [7, 1, 4, 8], since they abstract away low-level details such as system setup, dev-ops, and monitoring — promising service isolation, decentralization, and scalability, while still being more cost-effective compared to dedicated servers. Since FaaS services execute arbitrary user pipelines, FaaS system must execute code in isolation — through virtual machines (VMs) or containers.

Current off-the-shelf DL inference [10, 2, 6, 3, 9] is performed through HTTP APIs and uses pre-built general models (model catalogs) deployed by the cloud provider Within the FaaS pipelines, users interact with these remote models inference services through network and construct their prediction pipelines by defining glue code that parse the input, perform the model prediction, and process the output. There are two type of model inference services, batch inference and online inference. Batch inference is performed offline on a large set of inputs, while online inference is usually performed in real-time on a one-by-one basis [11, 14]. In this paper we focus on online inferences within a latency sensitive function 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. FaaS can be used to express latency sensitive prediction pipelines that leverage a chain or ensemble of models. However, the current practice of integrating FaaS with model catalogs is inefficient for this usage — the "cold start" model load latency associated with the inference limits how complex or intelligent a pipeline can be — making these pipelines out of reach for most but the cloud giants. DL service providers are aware of the "cold start" cost of inference, and therefore eagerly persist models within their catalog — keeping the models in memory ("warm") to guarantee the promised latency. For example, for premium cost, Amazon ML attempts to respond to most real-time prediction requests within $100ms$ [12]. This introduces waste and increases the cost of inference, yet without model persistence, model loading contributes to a significant portion of the end-to-end inference latency.

We observe that for "cold start" model inference, model loading (I/O, data structure deserialization, GPU data movement) is the main source of "cold start" latency. Figure 1 shows the "cold start" inference time breakdown for popular DL frameworks: Caffe [26], Caffe2 [25], MXNet [17], and TensorFlow [15]. For GPU inference, data movement is another contributing factor making GPU less attractive for accelerating inference — even though GPUs offer a significant compute speed advantage, as shown in Figure 1.

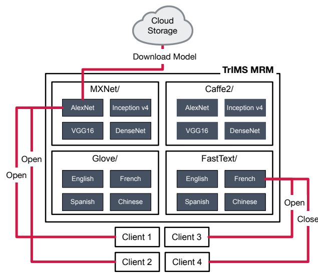
An insight is that within a cloud setting DL models are shared extensively across user functions or use common API to perform ML related tasks. For example, Google reported that 41 natural translation models can accommodate over 75% of their translation requests in [6]. Because model parameters are constant, we can leverage model sharing across pipelines by persisting model parameters in GPU and/or CPU memory, hence eliminating the model loading overhead, decreasing the end-to-end latency, and reducing the memory footprint for DL inferences. Based on this observation, we propose TrIMS to eliminate 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.

In this paper, we propose a Transparent and Isolated Model Sharing (TrIMS) scheme to address the "cold start" latency challenge faced by collocating user code with model catalogs within FaaS — it does so while maintaining the isolation constraints, minimizing model-loading overhead, and increasing hardware resource utilization. We describe TrIMS's model resource manager (MRM) which offers a multi-tiered cache for DL models to be shared across user pipelines. By decreasing model loading and data movement overhead, TrIMS decreases latency of end-to-end model inference, making inference on GPU a viable target. TrIMS also increases memory efficiency for cloud data centers while maintaining accuracy. Specifically, this paper makes the following contributions:

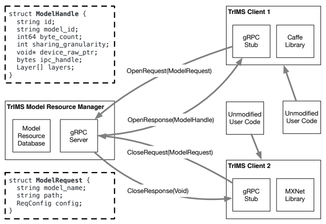
- We characterize the "cold start" overhead for online DL model inference across popular DL frameworks, such as Caffe, Caffe2, MXNet, and TensorFlow, on both CPUs and GPUs and identify model loading as the bottleneck.
- We propose TrIMS to mitigate the model loading overhead faced by collocating user code with model catalogs within FaaS, and increase the model serving efficiency 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 online prediction pipelines while increasing hardware efficiency and decreasing latency.
- We implement TrIMS within Apache MXNet [17] and evaluate the impact on online inference performance for a representative set of models and systems. We show that TrIMS provides $1.12\times - 24\times$ speedup and is within 20% of ideal speedup (with ideal being that model loading and data movement takes no time).
- TrIMS eliminates a substantial part of the non-compute components of the end-to-end latency, making DL model inference on GPU and novel compute accelerator more viable.
- We architect TrIMS so that it can be easily integrated with existing frameworks without user code changes. The method is designed to be compatible with existing framework usage patterns, and requires minimal modifications for framework developers.

2 TrIMS Design

Within TrIMS, models are managed by the Model Resource Manager (MRM) server. TrIMS's MRM is a model server daemon that performs model management and placement — abstracting away the model loading from framework clients. Each framework client communicates with MRM through inter-process communication (IPC). MRM maintains a database of models, addressing them using namespaces, with framework as well as model name and version being used to distinguish models. Figure 2a shows that MRM is managing models for MXNet, Caffe2 DL frameworks as well as word vector embedding models for FastText and Glove. 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



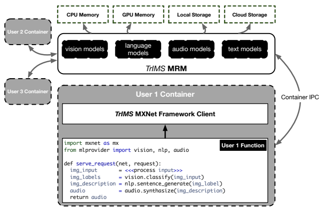
(a) TrIMS' MRM.



```
struct ModelHandle {
  string id;
  string model_id;
  int64 byte_count;
  int sharing_granularity;
  void* device_raw_ptr;
  bytes ipc_handle;
  Layer[] layers;
}
```

```
struct ModelRequest {
  string model_name;
  string path;
  ReqConfig config;
}
```

(b) TrIMS model load cycle.



```
import mxnet as mx
from mlprovider import vision, nlp, audio

def serve_request(net, request):
  img_input       = <<<process input>>>
  img_labels      = vision.classify(img_input)
  img_description = nlp.sentence_generate(img_label)
  audio           = audio.synthasize(img_description)
  return audio
```

(c) TrIMS usage within FaaS.

traditional CPU cache hierarchy. Because of this, we will simply refer to these four different memory hierarchies as "cache" in the rest of this paper whenever there is no ambiguity.

After system cold boot, initial model requests miss the GPU, CPU, and local storage caches, causing the model to be downloaded from the cloud storage and loaded into the "caches" to serve both the current quest and future requests. When one of the caches becomes full, one or more models are evicted from the cache.

TrIMS leverages the CUDA runtime's cudaIpc* to share GPU memory across processes. 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 difference processes to share the same GPU virtual address space (VAS). For Volta GPUs, NVIDIA introduced a new feature to allows contexts to share page-table mappings. This makes it possible for user processes to run using different contexts while still sharing memory.

The 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, respectively. MRM maintains a reference count for each model to determine the number of users currently using the shared model. The API is shown in Figure 2b. On `trims::open`, the MRM needs to handle three cases:

1. 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.
2. 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.
3. 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.

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 these models. Models is reclaimed 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).

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

- 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.

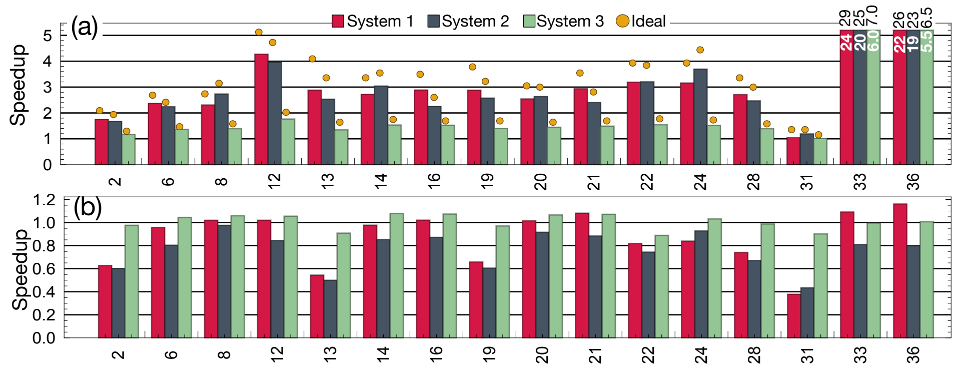


Figure 3: A representative sample of the models shown in Table 2 are chosen and are run on the systems in Table 1 to achieve (a) the best case end-to-end time — when the model has been pre-loaded in GPU memory — and (b) the worst case end-to-end time — when the model misses both the CPU and GPU persistence 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).

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 1: 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.

TrIMS works within Python, Java, or R. 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 [35] could share their frozen layer weights.
- Multi-GPU and Multi-Node Support — Multi-GPU is usually used when performing batched inference [13, 16]. TrIMS inherently supports the multi-GPUs by leveraging Unified Memory (UM) [5].

3 Evaluation

The experiments reported in this paper are based on an implementation of TrIMS on top of Apache MXNet. Since 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$\times$ faster for inference on average than the vanilla MXNet for the suite of models we use. We use the modified MXNet as our baseline for evaluation.

We evaluate TrIMS on 3 systems (shown in Table 1) using 37 (shown in Table 2) pre-trained models. The systems selected represent different types of instances that are currently provisioned in the cloud. System 3 uses the NVLink bus [20, 33] which allows up to $35 GB/s$ 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 [36, 34].

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

ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Network	[27]	[31]	[27]	[21]	[18]	[18]	[32]	[30]	[24]	[32]	[30]	[37]	[28]	[22]	[22]	[22]	[22]	[22]	[22]
#Layers	16	116	16	16	361	481	472	747	416	416	1102	514	24	526	522	777	769	761	99
ILS	516	111	512	479	122	340	257	399	313	142	493	666	131	423	428	548	721	340	154
MWMF	238	27	233	221	49	145	92	164	129	44	214	285	29	170	171	231	311	231	45

ID	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	
Network	[22]	[22]	[22]	[22]	[22]	[38]	[38]	[38]	[38]	[38]	[23]	[23]	[29]	[41]	[40]	[29]	[39]	[19]	
#Layers	1009	1346	179	268	259	526	522	147	271	267	52	52	32	32	32	38	267	236	
ILS	589	889	222	270	275	375	378	147	222	224	34	28	1228	1198	1195	1270	758	244	
MWMF	248	391	84	98	98	170	170	59	96	96	4.8	4.8	528	514	513	549	264	88	

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

and the analysis would apply to other types of networks.We select 37 pre-trained image processing models, shown in Table 2, that are based on their popularity in both research and usage – with some networks having variants. We compare our performance within a FaaS setting against ideal (where the model loading and data movement takes no time — ideal is faster than model persistence) and use end-to-end "cold-start" inference as the base line, since that's what is currently employed by FaaS environments.

We measure the end-to-end "cold-start" inference of MXNet with and without TrIMS – for the sake of clarity we omit the input processing time. Figure 3 shows the achieved speedup on a representative set of the models compared against MXNet that does not utilize TrIMS. We show two cases: (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 3a), 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 a slight 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 improves proportionally to the model size, the system's data movement bandwidth, the system's compute resources, and the model's compute complexity. For models, where the I/O overhead is very low, for example SueezeNet (which has a $5MB$ 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. A simple optimization is to bypass TrIMS for small models (which can be known statically).

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 3b), 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.

4 Conclusion

We propose TrIMS to mitigate the major source of "code start" FaaS latency — the model loading overhead — and make building complex latency sensitive pipelines with modular DL components feasible. 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.

The proposed method is not restricted to DL workloads, but we use DL as a motivating case. 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 reducing latency or cost of inference.

References

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

[2] Amazon Rekognition. https://aws.amazon.com/rekognition. Accessed: 2018-8-04.

[3] Azure Cognitive Services. https://azure.microsoft.com/en-us/services/cognitive-services. Accessed: 2018-8-04.

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

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

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

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

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

[9] IBM Watson. https://www.ibm.com/watson. Accessed: 2018-8-04.

[10] Machine Learning on AWS. https://aws.amazon.com/machine-learning. Accessed: 2018-8-04.

[11] Online versus Batch Prediction. https://cloud.google.com/ml-engine/docs/tensorflow/online-vs-batch-prediction. Accessed: 2018-8-04.

[12] Requesting Real-time Predictions. https://docs.aws.amazon.com/machine-learning/latest/dg/requesting-real-time-predictions.html. Accessed: 2018-8-04.

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

[14] Using the Model to Make Predictions. https://docs.aws.amazon.com/machine-learning/latest/dg/using-the-model-to-make-predictions.html. Accessed: 2018-8-04.

[15] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

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

[25] Yangqing Jia. Caffe2. `https://www.caffe2.ai`, 2017.

[26] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In Proceedings of the 22nd ACM international conference on Multimedia, pages 675–678. ACM, 2014.

[27] 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.

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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] Arnold Tharrington, Wael R Elwasif, and Don Maxwell. Experiences evaluating functionality and performance of ibm power8+ systems. In High Performance Computing: ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^ 3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers, volume 10524, page 254. Springer, 2017.

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

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

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.