Exploring Novel Architectures For Serving Machine Learning Models

Aditya Chopra

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-73
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-73.html

May 18, 2018

Exploring Novel Architectures For Serving Machine Learning Models

by Aditya Chopra

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Randy Katz
Research Advisor

(Date)

⋆ ⋆ ⋆ ⋆ ⋆ ⋆ ⋆

Professor Krste Asanović
Second Reader

(Date)

Exploring Novel Architectures For Serving Machine Learning Models

Abstract

Exploring Novel Architectures For Serving Machine Learning Models

by

Aditya Chopra
Master of Science, Plan II in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley
Professor Randy Katz, Research Advisor

In this work, we present two serving-time machine learning service architectures. Our goals are to improve resource utilization, inference latency, and throughput.

The first system, Shockwave, leverages hardware accelerators to explore pipelining a machine learning model such that we can achieve low batch size, low latency inference while improving hardware utilization. Shockwave's model is particularly compelling given the increasingly prevalent disaggregated datacenters.

The second system, ModelFuse, attempts to manipulate inference graphs to reduce overall latency when serving models. Specifically, we optimize model graphs of a single model, or multiple models being served together. ModelFuse offers a compelling way to combine those inferences to reduce overall latency.

Contents

List of Figures

List of Tables

Acknowledgments

This work would not have been possible without the help and collaboration of many of my colleagues at the ADEPT and RISE labs. First, I want to thank Martin Maas for introducing me to the world of research and guiding me through my undergraduate years into grad school. I would also like to thank Sagar Karandikar, Nathan Pemberton, Colin Schmidt, Dayeol Lee, Howard Mao, and the rest of my colleagues at the ADEPT lab as well as Corey Zumar, Ankit Mathur, Chia-Che Tsai, and Jeongsok Son from the RISE lab.

I would like to thank my advisers Professors Randy Katz and Krste Asanović for providing me the opportunity to do research throughout my undergraduate degree and for providing guidance and support during my Master's degree. Thank you also to Professor Raluca Ada Popa for giving me the opportunity to both research and teach security.

Thank you to my friends for all the adventures, memories, and support. Last but not least, I would like to thank my parents, Meenu and Rajesh Chopra for their unwavering support, positive influence, and inspiration. Without them, none of this would have been possible.

Chapter 1

Introduction

Machine learning has become ubiquitous across consumer applications. Machine learning algorithms power everything from natural language processing to computer vision. These algorithms can be broken down into two classes: *training* and *inference*. Training workloads are characterized by large amounts of data being repeatedly loaded into memory to train a model. The inference workload, by comparison, needs to process far less data in order to classify a single piece of input data. While a large body of research has been dedicated to improve training performance, particularly because of this distinction in workload type, inference has not been given as much focus. However, while a model can be allowed to train for long amounts of time on the order of hours to days, inference has strict performance requirements because of its consumer-facing nature. Given these constraints, systems engineers face an interesting problem of being able to optimize a relatively small computation to achieve ambitious performance goals.

Because of the enormous data, computational, and engineering requirements for developing cutting edge machine learning models, companies such as Google devote resources to developing machine learning models which they offer as a service to businesses and consumers. For example, Google offers Cloud Vision API [19] as a machine learning service that classifies images. By leveraging Google's API, businesses can abstract away the complexities of developing and training such models and just focus on deployment in their own product. Because fast and consistent performance from these models can be essential for a cloud provider's customers, it is important for these companies to enforce strict service-level agreements. Two key metrics for measuring this performance are *latency* and *throughput*.

For a machine learning service, latency refers to the time it takes from the server receiving the request to outputting the classification. This is an important metric for companies to optimize for real-time applications because a large latency may directly manifest itself in performance degradation in customer applications. Not only is it important to optimize the average latency but also the *tail latency*. Tail latency refers to the tail part of the latency distribution across a large number of requests. Even if a

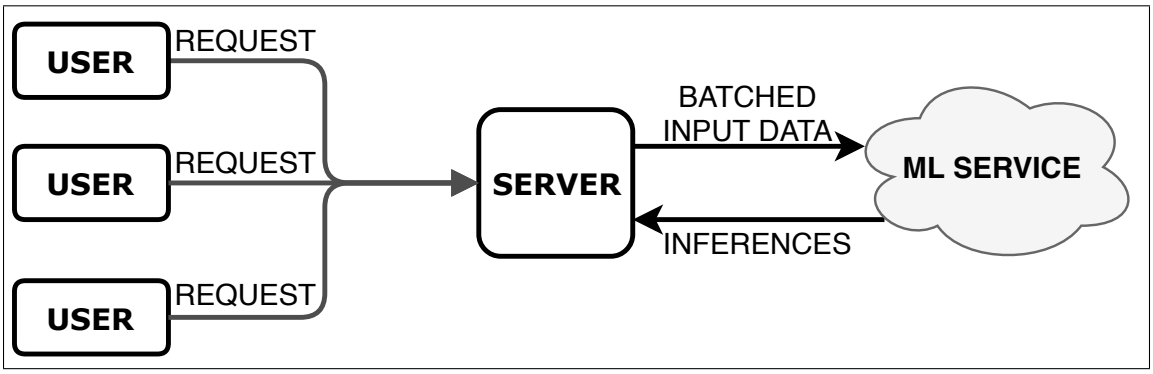


Figure 1.1: Batching in a serving system.

service has a desirable average latency, it is important that said latency is consistent.

Throughput refers to the number of classifications made per time quanta. This is an important metric to optimize for applications that require a large number of inferences. Traditionally, in serving systems there is a user-level tradeoff between throughput and latency. Requests can be batched together in order to improve throughput. However, with large batch sizes, come larger average latencies because requests need to be accumulated before being batch processed.

A metric that service providers look to optimize is *utilization*. A company offering a service is incentivized to completely utilize all their resources to maximize profit. In practice, maximizing utilization is hard because it is impossible to predict exactly how many requests will be incoming to the service at a given time, meaning that a fully-utilized server would make incoming requests wait for the resources needed to process them. This would hurt the service metrics such as latency and throughput, which service providers are further incentivized to keep performant.

The crux of the problem that service providers face is that of maximizing hardware utilization while also ensuring low latencies and high throughput. Hardware accelerators take us one step towards solving this problem by allowing for faster computation, thereby reducing latency while maintaining batch sizes.

In the past, companies were disincentivized from using specialized hardware. There was no reason to spend more money on specialized hardware that will outperform a general-purpose CPU until the next generation CPU comes out with far greater performance. Now, as we see the end of Moore's law, CPUs are no longer able to compete with hardware that has been built specifically for certain workloads. Companies are now economically incentivized to build specialized hardware accelerators such as Google's TPU [10], which is designed to offer performance improvements when running TensorFlow [1] applications. Accelerators are particularly useful for machine learning workloads because these workloads repeatedly use the same mathematical operators such as matrix multiplication. Building said operators into hardware can immediately yield performance upgrades.

Cloud providers have also begun exploring novel datacenter architectures such as the disaggregated [5] model. A disaggregated datacenter is built around splitting resources such as CPUs, storage, and specialized compute. A disaggregated datacenter is desirable because it allows for easy hardware upgrades (because of the decoupling of resources) and statistical multiplexing, which allows for higher resource utilization. Given pools of specialized compute, or hardware accelerators, we are faced with the systems problem of how best to design our software to leverage this compute to serve machine learning models.

The contributions of this paper are as follows:

1. In the first section we explore sharding a single model's inference graph onto several accelerators. In this exploration, we build and evaluate Shockwave, a machine learning model serving pipeline which leverages a pool of custom hardware accelerators.

2. Then in the next section we explore ModelFuse, a system built to transform model graphs in order to improve single and multi-model inference performance.

The structure of the paper is as follows. Chapter II describes Shockwave. Chapter III describes ModelFuse. Chapter IV concludes the paper with a high-level analysis of all the results.

Chapter 2

Shockwave

2.1 Motivation

Neural networks can take on the order of days to train. Given the significant amount of time spent training and tuning these networks, a single version of the network can be served for extended periods of time. Because these neural networks don't change often, we can bake the weights into the hardware serving the models. This enables us to avoid the overhead involved with loading weights into memory by persisting them in on-chip registers. This is particularly useful in the case of larger neural networks such as VGG [20] that can be on the order of 100s of MB large.

Unfortunately, because these networks can be so large, we cannot persist them in a single accelerator's on-chip memory. This is where the idea of a pool of hardware accelerators is useful. We can shard the large neural network across a series of accelerators, which communicate over a datacenter network. Furthermore, given the trend towards disaggregated datacenters, we need to consider that the link latencies for accessing memory and CPU can be much higher than that required to access other accelerators.

We explore this idea with Shockwave, a bare-metal machine learning model serving pipeline. The goal is to leverage a pool of hardware accelerators in order to improve hardware resource utilization while maintaining inference latency, despite the additional communication overhead between accelerator nodes.

2.2 Related Work

2.2.1 Brainwave

Microsoft recently announced their real-time AI system, Brainwave [3]. Project Brainwave leverages Microsoft's cloud FPGAs to synthesize soft deep neural network (DNN) processing units (DPUs). More specifically, they leverage pools of FPGAs

to serve pretrained machine learning models. When multiple DPUs serve a single model, each contains a piece of the DNN. They claim that this design beats out other architectures because:

1. FPGAs are flexible and allow for hardware innovations to be realized quickly.
2. They avoid batching by optimizing their soft DPUs to leverage all of the on-chip resources to perform a single inference. This reduces latency while maintaining high utilization.

Brainwave leverages the scalability of cloud FPGAs to deliver hardware microservices for serving machine learning models independent of CPU.

2.2.2 FireSim

To experiment on a disaggregated datacenter, we use FireSim [11], a cycle-exact, FPGA-accelerated datacenter simulator developed at Berkeley. FireSim allows users to use RTL to specify custom datacenter blades, use C++ to customize switches, and configure network topologies and parameters at runtime. This is particularly valuable for Shockwave, as it allows us quickly iterate on datacenter parameters to identify bottlenecks when doing performance analysis.

2.2.3 Hwacha

Hwacha [14] is a non-standard RISC-V extension, also built at Berkeley, that uses a vector-fetch paradigm. It is built to optimize for better performance, better energy efficiency, and lower complexity. Specifically, we utilize the vector intrinsics to accelerate our neural network computations.

2.3 System Design

We choose to evaluate a one-layer neural network for identifying handwritten digits. This neural network was trained separately on the MNIST [13] dataset. This dataset is often used as a proof of concept network due to its small size and simple design.

We use FireSim to simulate a six node pipeline and measure cycle-accurate timing information. Our simulated server blades are RISC-V rocket cores running at 3.2 GHz, with 16 KiB Instruction and Data Caches, a 256 KiB L2 Cache, and a 200 Gbps Ethernet NIC. We connect the server blades to a top-of-rack switch with 200 Gbps links of varying latencies.

We use the Hwacha vector accelerator to speed up our computations. Hwacha lends itself to our use case since it intrinsicily supports vector and matrix computations, which are commonly found in neural network inference computations.

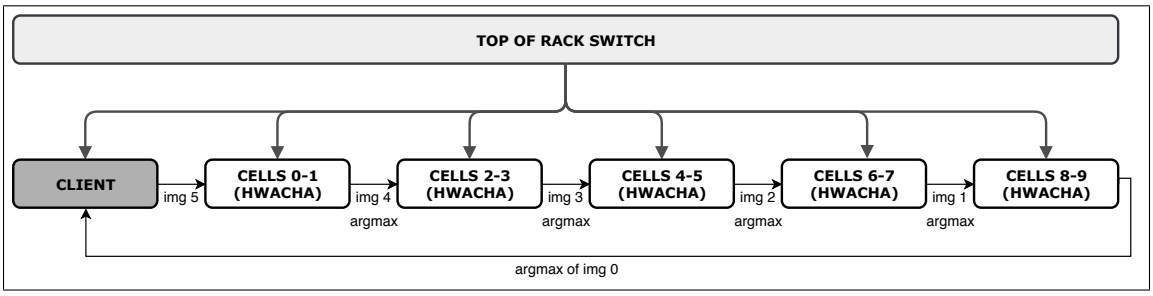


Figure 2.1: The Shockwave pipeline design.

2.3.1 Pipeline Design

We architect our pipeline such that each node is processing one image at a time. Before we discuss what each node is doing, we look into how the neural network computation produces classifications.

In this one-layer neural network, we have 10 cells. Each cell computes the dot product between the weight associated with it, and the image (represented as a vector). Each dot product represents the probability of the given cell being the image's classification. For example, if the cell at index 3 ends up having the largest dot product, the classification is 3 (with probability equal to the normalized value of the dot product).

Now, consider a single node in the pipeline. It begins by taking in an image vector and the argmax of the previous nodes' computations. Then, the node is responsible for performing the computations for two cells. It will compute two dot products, compare the values with the input argmax value, and pass the argmax of those values on to the next node. Then, at the end of the pipeline, the resulting argmax is the classification.

All of the nodes are interconnected using a top-of-rack switch and communicate over an Ethernet link.

2.3.2 Alternate Designs

We also considered other designs for leveraging a pool of hardware accelerators. For example, instead of having requests directly enter the pipeline, one may also design a system where client requests are first handled by a load balancing node that forwards requests on to different nodes based on their utilization. The main difference between this design (Figure 2.2) and Shockwave is that each accelerator node processes the entire neural network. This means that for large neural networks, the weights take up a significant part of memory and need to be loaded for every inference. Furthermore, the design suffers from software overhead of utilization monitoring and load balancing. However, this design is similar to Shockwave in that it benefits from using the entire processing capability of the accelerator to process each image.

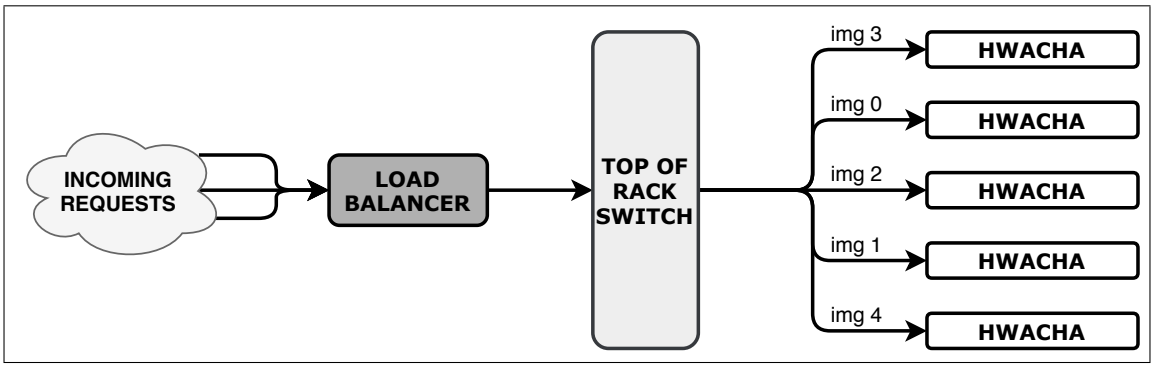


Figure 2.2: An alternative design.

	Cycles	Cycles (2 imgs)[1]
2 cells without Hwacha	460857	158925
2 cells with Hwacha	439805	76397
2 cell cycle savings	21052	82528
Aggregated savings[2]	105260	206320[3]

Table 2.1: Cycle counts from running a VCS simulation of one node with and without pipelining[4].

2.4 Evaluation

We begin by evaluating a single node running two cells on a VCS simulation of a Hwacha where the node does not utilize the Hwacha's vector instructions. Then, we perform the same simulation, only using the vector instructions. We found that we saved 21,052 cycles by using the accelerator. Furthermore, since this was a simulation of one node, we can extrapolate the per-node cycle savings to our entire pipeline, resulting in a speed up by $\sim$ 105,000 cycles.

Next, we simulated a node classifying two images. This was to measure the effect of baking the neural network weights directly into the hardware. We expected to see increased savings, since the model weights would not need to be loaded in redundantly for each image. The results indicate that the per image cycle savings are increased by a factor of two.

Next, we scaled the per-image performance improvements from using Hwacha and subtracted it from the cycle counts from a FireSim simulation of the pipeline. In Figure 2.3 we can see that as the number of images scales up, the performance

[1]The cycle counts for this column differ because of binary version.
[2]Cycle savings per image across all nodes.
[3]Normalizes the cycle savings for a single image.
[4]Note that in order to measure pipelined performance, we run the node on two images.

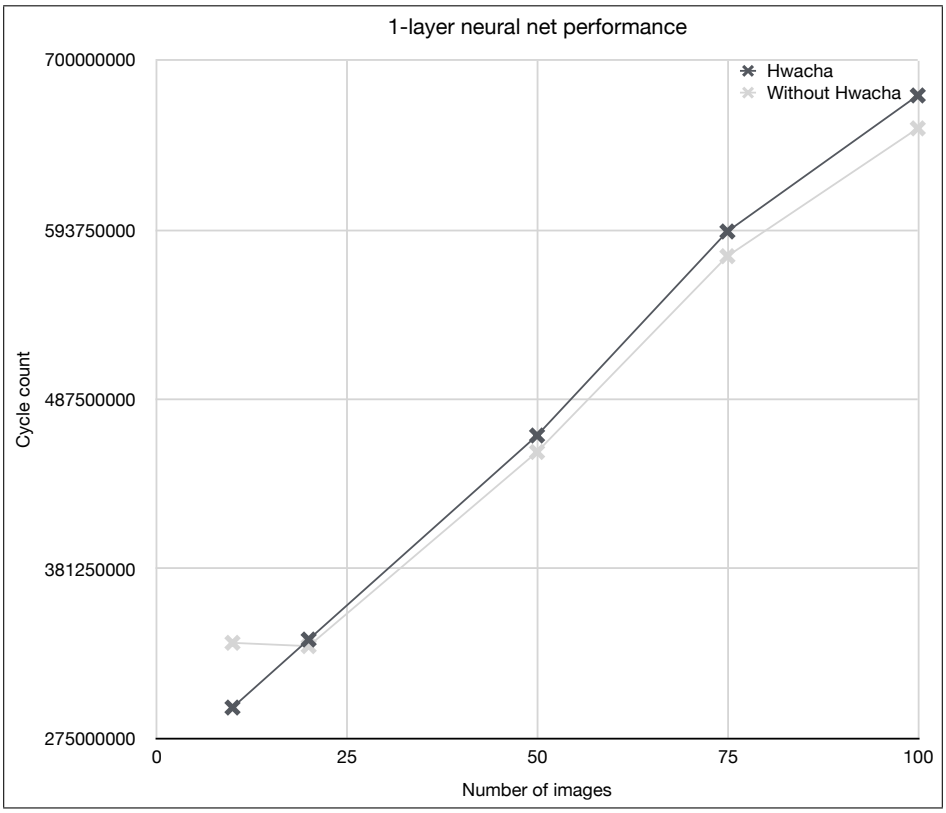


Figure 2.3: Pipeline performance with and without Hwacha.

improvements become significant.

After computing linear regressions on the data collected, we found that the non-Hwacha pipeline cycle count scales in the number of images with a factor of 4.354×10^6 while the Hwacha pipeline scales with a factor of 3.837×10^6. Dividing the two, we determine that Hwacha provides a speedup of around $\sim 1.13\times$ for this use case.

Our next evaluation was of the entire pipeline in FireSim. We ran the pipeline with three network link latencies ranging from 1 μs to 11 μs. As seen in Figure 2.4, the latencies did not offer much performance improvement. Despite setting link latencies to extremely low values, we found that there was no major improvement.

Network	**Link latency** (μs)
0	1.001
1	5.469
2	10.938

Table 2.2: Network configurations.

We investigated the issue and found that it lay in the fact that nodes would

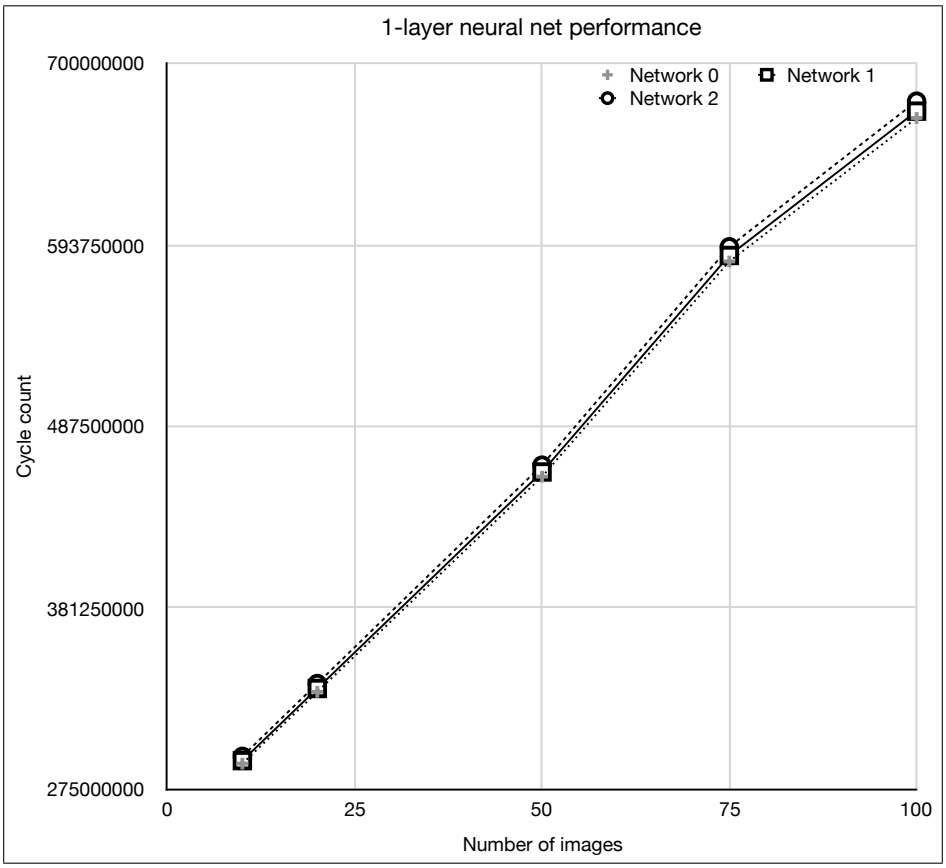


Figure 2.4: Pipeline performance different with varying network parameters.

receive the data from packets, perform computations, and forward them on. These nodes were not signaling to the previous nodes that they were done with computations, meaning that the previous nodes would either wait too long before sending more packets and waste time, or drop packets all together because the receiving nodes had not finished their computations. By introducing a feedback mechanism, we believe we could resolve this bottleneck and improve throughput.

2.5 Conclusion

In this work, we built a neural network serving pipeline and evaluated it against different network capabilities. We leveraged FireSim to quickly prototype and accurately measure the performance of our model. We also modeled performance improvements from utilizing Hwachas to accelerate our model's classification. We learned that the bottleneck in pipelines such as the one we built remains in the computation. While improving the network link latencies offered some speedup, after a certain point, the neural network computations were preventing further performance improvements.

In terms of future improvements for Shockwave, we plan to:

1. Re-evaluate the performance when Hwacha support is added to FireSim.
2. Explore more complicated machine learning workloads such as deep neural networks.
3. Introduce a load generator to measure tail latency performance.

In the next chapter, we move from exploring resource-based optimization to model graph optimizations.

Chapter 3

ModelFuse

3.1 Motivation

As machine learning is becoming increasingly productionized, model serving is emerging as a central problem. Many larger companies have well-developed, yet proprietary solutions for serving models, while companies with less infrastructure at their disposal often leverage ad hoc solutions or traditional query serving systems. The recent introduction of Clipper [4] and Tensorflow Serving [16], two dedicated model serving platforms, is helping to improve the accessibility of serving infrastructure to a broader set of organizations.

These model serving systems rely on an abstraction that treats each model as a black box entity. Furthermore, these systems inhibit fine-grained hardware allocation decisions, stipulating that only a single model be executed on a given accelerator at any one time. We refer to this allocation policy as the one-to-one serving paradigm.

While few companies are able to allocate a dedicated compute resource for each served model per quantum, this is not generally feasible. Rather, models often need to share a constrained set of hardware resources. Accordingly, optimal utilization of this limited set of resources is critical. However, the one-to-one paradigm falls short of this goal: evaluating a single model rarely utilizes the entirety of available compute afforded by a hardware accelerator. This underutilization is particularly prevalent in the case of serving ensembles of lightweight models.

Ensembles are frequently used for a wide variety of production use cases, including object identification, A/B testing, and data analysis. In particular, model bootstrap aggregation (bagging) depends on the evaluation of large collections of lightweight models with homogeneous architectures [2].

Additionally, model evaluation often includes a pre-processing step that converts inference inputs into a model-compatible format. This compute-intensive preprocessing is often performed for each model evaluation. However, a wide variety of models process inputs of the same size and data type; therefore, aggregated preprocessing is likely to reduce these compute overheads and improve inference perfor-

mance when serving multiple models.

Accordingly, this exploration is motivated by the following question: Can we improve hardware utilization by evaluating multiple models on the same hardware resource? Our solution leverages the fact that models can be represented as declaratively-specified computation graphs comprised of layer operators; in practice, graphs reuse a small set of popular layers. We propose merging these layer operators across model graphs to increase utilization by more effectively saturating GPU memory and leveraging unused computational capacity in GPUs. Furthermore, we propose dynamically aggregating compute-heavy pre-processing steps, parallelizing their evaluation on GPUs, and broadcasting results to multiple models for continued evaluation.

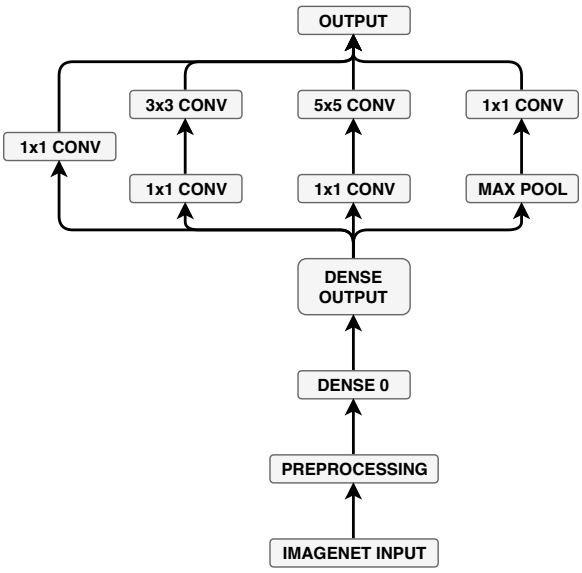


Figure 3.1: A simplified version of the Inception [21] graph

Toward these ends, we present ModelFuse: a system for optimizing and serving computation graphs. ModelFuse consists of:

1. A dynamic graph refactoring component that efficiently analyzes and restructures computation graphs by merging their constituent pre-processing steps and layer operators

2. An efficient serving framework, based on Clipper, that supports collocation of multiple logical models on the same hardware resource

Each component of ModelFuse is useful in isolation; the graph refactoring component can help to achieve more performant deployments on existing serving platforms, such as Tensorflow Serving, while the serving framework affords finer-granularity control over model placement on hardware resources.

3.2 Related Work

3.2.1 Clipper

Clipper is a serving system that uses a variety of techniques to improve inference. The Clipper system studies many optimization techniques, such as adaptive batching, caching, and model selection. In addition, it defines a general abstraction using containers, and that allows it to serve models built in a wide variety of machine learning frameworks.

Clipper also learns the best batch size for the throughput-latency tradeoff, allowing for automatic hyperparameter tuning. It also caches outputs and hashes inputs to improve latency for repeated queries, demonstrating the usefulness of traditional systems solutions in an inference system.

Finally, Clipper uses a multi-armed bandit algorithm to select which models to serve from in an ensemble, creating an abstraction for feedback that can allow for an inference system to adaptively respond to real-world responses to the outputs from the inference system. In general, it defines a powerful abstraction for serving and a far more general solution.

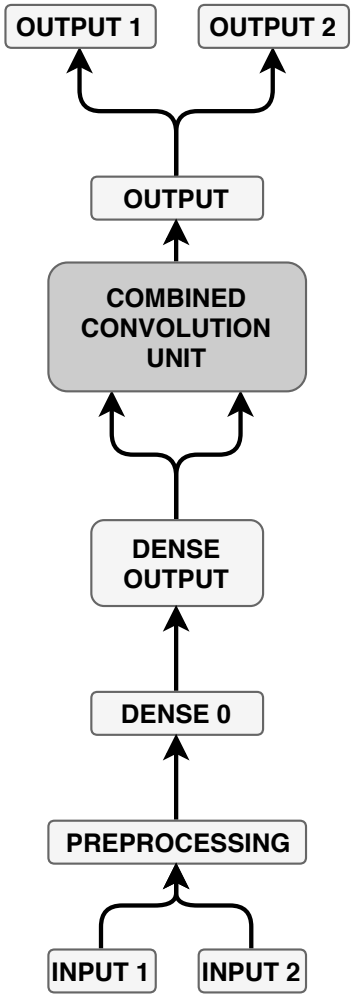


Figure 3.2: The operator graph of two fused Inception models

3.2.2 Tensorflow Serving

TensorFlow [1] is an open-source machine learning framework built by Google. It supports easy customization of machine learning algorithms by allowing users to specify primitive operations in a dataflow graph. TensorFlow supports algorithms and tools for training a variety of models. TensorBoard is of the tools that we used in our experimentation, and is useful for visualizing model graph structure. TensorFlow also includes a serving system for serving TensorFlow models in production.

TensorFlow Serving is the native serving system offered by TensorFlow which focuses on providing low latency serving of models. Specifically, it offers libraries for serving custom models, multiple models in a single process, multiple versions of a single model, and variable batch sizes.

3.2.3 TensorRT

TensorRT [6] is a deep-learning inference optimizer built by NVIDIA. TensorRT offers graph structure optimizations, precision optimizations, kernel auto-tuning, and memory reuse optimizations. While TensorRT supports multiple frameworks such as Caffe [9], TensorFlow, and PyTorch [17], it is most compatible with TensorFlow and has recently been integrated with TensorFlow.

TensorRT's graph-based optimizations fall under two categories: vertical fusion, and horizontal fusion. Vertical fusion of graph operators involves fusing sequential operators into a single combined operator. An example of vertical fusion would be the merging of the sequential `MAX POOL` and `1x1 CONV` layers in Figure 3.1. Horizontal fusion involves fusing layers that aren't necessarily sequential, but input data and filter size. An example of horizontal fusion would be the merging of all of the `1x1 CONV` layers in Figure 3.1 that take input from the `DENSE OUTPUT` layer. We would expect the result to look something like the graph in Figure 3.2. Layer fusion can offer significant performance improvements because every operation requires a kernel launch, which often slower than the actual kernel computations. Thus, by fusing layers into fewer kernels, we avoid kernel launches and their overhead. Furthermore, we also avoid the cost associated with reading and writing the intermediate data into memory.

3.2.4 ModelBatch

ModelBatch [15] is a system aimed at improving model search (via training and inference) by using batching techniques. In particular, the paper takes the approach of using multiple CUDA streams to increase parallelism during training and inference time, while batching preprocessing on the CPU.

However, the CUDA stream approach struggles to integrate into existing frameworks, since it requires having a CUDA stream to launch. While the details of their implementation are not released, the extended abstract suggests that ModelBatch experiments involve custom GEMM kernels. ModelBatch claims to launch kernels for each model, but it's unclear how arbitrary Tensorflow models are translated to CUDA kernels.

3.3 System Design

3.3.1 Fusion

The layer fusion module in ModelFuse currently works for Tensorflow graphs. The static computation graph abstraction that Tensorflow offers makes it possible to analyze the entire graph and run graph algorithms on the graph. Tensorflow also

offers convenient abstractions for loading these graphs back in via the SavedModel API.

However, this does not fundamentally limit to this approach to Tensorflow - the machine learning community's development of the ONNX format for all frameworks to export models to means this approach will soon be convenient in most popular frameworks. The fusion module of ModelFuse takes in as input two model graphs and outputs one, combined model graph. This graph can be served via any serving system.

Preprocessing ModelFuse can use Tensorflow graphs to identify computationally similar preprocessing operations across different computation graphs. Either the user specifies the name of the input tensor or the system analyzes the static graph and looks for input placeholder tensors that might represent inputs to the graph.

ModelFuse has been pre-built with Tensorflow subgraphs that contain preprocessing operations for several standard input types (MNIST [13], ImageNet [18], etc.). The system compares the tensor shapes of the graph it is analyzing with these preprocessing signatures. Users can also specify a new preprocessing signature by tagging those operations with a preprocessing scope. The preprocessing components are executed on the GPU by concatenating the input batches of each model and executing the Tensorflow subgraph with the GPU enabled. It is also possible to have optimized CUDA kernels that are executed instead.

Operator Fusion ModelFuse uses a similar approach to preprocessing signatures to identify mergeable operators. It stores lists of operators that correspond to specific layers. These lists are easily populated by analyzing the graphs created by the layers defined in the Tensorflow layers API. ModelFuse implements combined operators, and those are interpreted as a set of possible legal transformations over the graphs that are being analyzed. Currently implemented transformations include Dense layers and Convolution layers. All possible transformations are then scored via a cost model that takes into account what part of the graph the operator is in (one wouldn't want to merge an operator that is later on in one graph but earlier in another, since this would lead to near-sequential operation). The best graph is returned.

3.3.2 Serving Framework

While a component of dynamic layer and preprocessing fusion is necessary for achieving improved evaluation performance and hardware utilization, it is not sufficient; the optimized computation graph produced by the aforementioned component must be executed via a model serving framework. Though Clipper and Tensorflow Serving support the online evaluation of computation graphs, they do not offer the flexibility with regard to model placement that is required in order to fully realize the benefits of graph fusion.

3.3.2.1 Terminology

For the purpose of motivating serving framework design decisions, it is important to draw a distinction between the concept of a **model** and a **computation graph**. For the remainder of this section, we refer to a model as a mathematical function that maps a vector of batch inputs of a fixed data type and size to a vector of batch outputs of a fixed output data type and size. A computation graph is a collection of input pre-processing operations and layer operators. Note that, through the fusion of computation graphs, a single computation graph can support the evaluation of several models.

3.3.2.2 Hardware Limitations

We begin by discussing a critical hardware constraint by which serving systems need to abide. While high-level abstractions to facilitate task parallelism on CPUs have long been available to user applications, the same support is not yet available in the context of GPUs. Though the CUDA streams API provides support parallel GPU kernel evaluations [7] on NVIDIA-brand hardware, most popular machine learning frameworks fail to utilize it. Furthermore, CUDA is not a first-class language in the data science and model deployment landscape, and CUDA programs are not easily incorporated into computation graphs defined in frameworks based on higher-level languages, such as Tensorflow. Therefore, serving frameworks are restricted to executing a single logical evaluation process on a GPU at any given time. Attempting to execute multiple evaluation processes on the same GPU will, at best, fail to deliver significant performance improvements due to CUDA context switches and, at worst, produce erroneous or undefined behavior.

3.3.2.3 Clipper and Tensorflow Serving

Both Clipper and Tensorflow Serving abide by the restrictions on evaluation imposed by GPU hardware limitations. Clipper encapsulates each deployed model in a Docker container and maintains the invariant that only a single active container be deployed on a given hardware accelerator. This container-based architecture allows Clipper to support models written in a variety of machine learning frameworks. Similarly, Tensorflow Serving initializes a dedicated Tensorflow session for inference on a given GPU and exposes a single point of entry for computation graph inputs, effectively enforcing the association between a computation graph and a single model. However, these solutions are problematic because each computation graph deployed via either serving system is associated with a single mathematical model.

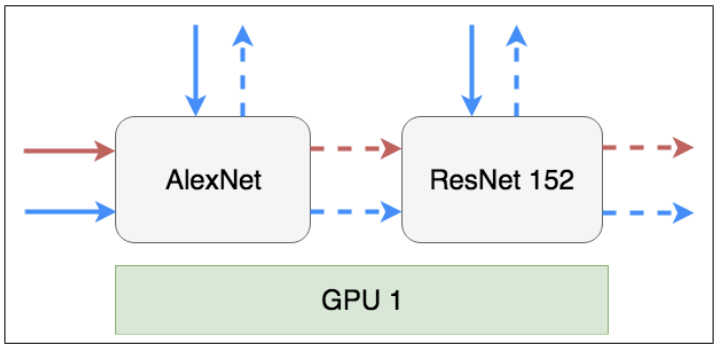


Figure 3.3: Clipper's containers can now take queries to one of many models in the same container that are all being evaluated on the same hardware resource.

3.3.2.4 Improvements Upon Clipper

ModelFuse improves upon the Clipper serving framework by providing support for logically associating multiple models with a single computation graph. Rather than maintaining the explicit invariant that each container be associated with a single model, ModelFuse allows a container to specify that it serves an arbitrary number of different models. Inputs are delivered to a container in the form of a dictionary keyed by model names. There are no restrictions placed on the models' input data types or sizes; a container is valid as long as its evaluation function produces a set of model-keyed outputs where the number of outputs per model matches the corresponding number of inputs. This API change allows containers to specify a unique codepath and, therefore, a unique point of entry for each model associated with it. Figure 3.3 demonstrates the additional flexibility afforded by ModelFuse over the original Clipper design. In the original Clipper system, a sequential computation graph consisting of AlexNet [12] operators and ResNet [8] operators within a single container must be treated as one model. As a result, AlexNet can only be evaluated if the entirety of the depicted sequential pipeline is evaluated; this inflexible code path is depicted via red arrows. In contrast, ModelFuse allows the container to expose the AlexNet and ResNet operators as separate models. As a result, there are now three code paths through the container, represented as blue arrows. Accordingly, AlexNet can be evaluated directly, without incurring the additional latency of ResNet evaluation.

Relaxing the constraint on the number of models that can be logically associated with a given container also necessitates a redefinition of the expression for a container's query batch size. While quantifying batch size in terms of a number of queries is meaningful when a container is associated with a single model, this unit is not appropriate when the container is hosting models with heterogeneous input characteristics. Accordingly, ModelFuse allows users to specify a container's optimal query batch size in terms of mebibytes of data. We argue that this quanitification is a better choice given the relevance of appropriate memory management and allocation

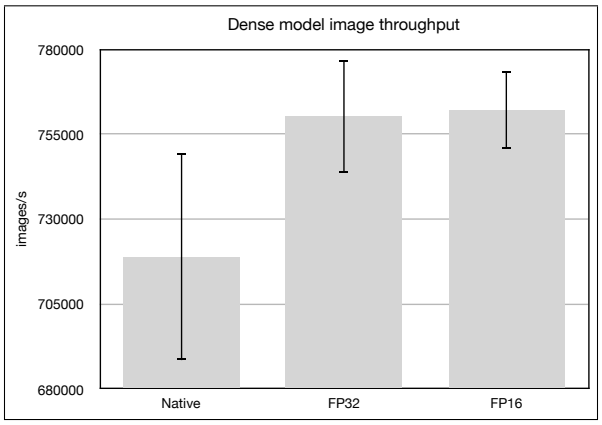


Figure 3.4: TensorRT's performance on MNIST Dense model.

to GPU inference. Failure to correctly estimate GPU memory utilization under peak load can result in erroneous behavior during inference.

Finally, ModelFuse intelligently handles deduplication of ensembled inputs to efficiently leverage the multi-model container setup. When an input is submitted for evaluation by multiple models, only a single copy of the input is sent to each target container. If the target container is logically associated with several models on which the input needs to be evaluated, a single input reference is maintained within the evaluating container. This reference appears in the set of model-keyed inputs once for each target model, avoiding unnecessary copies.

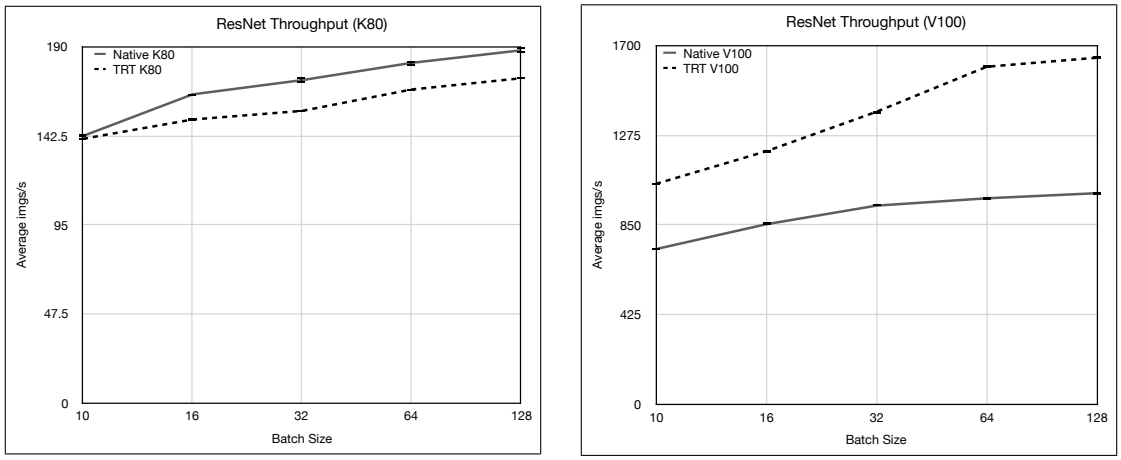
3.4 Evaluation

3.4.1 TensorRT

As discussed earlier, TensorRT is a proof of concept of layer fusion being useful in the serving context. ModelFuse competes with the layer fusion component of TensorRT, which was designed ad hoc and with specific kernels for modules in specific models that are useful for the customers of TensorRT. Benchmarking TensorRT functioned as an effective baseline for us to compare ModelFuse to. As far as we know, this benchmarking of TensorRT is not found elsewhere.

We performed several experiments with TensorRT that are worth reporting in this context.

We measured the improvement that TensorRT had on a standard Dense layer neural network for MNIST that used dropout in Figure 3.4. In this experiment, we used a Tesla K80 on a p2.xlarge instance on Amazon AWS. Note that the naïve throughput experiences far more variance, and TensorRT outperforms fairly significantly.



(a) Throughput on Tesla K80.

(b) Throughput on Tesla V100.

Figure 3.5: TensorRT's performance on ResNet on a Tesla K80 and Tesla V100.

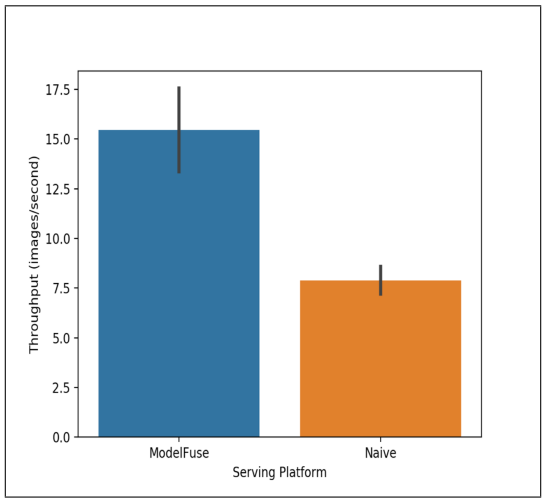
When evaluating TensorRT on ResNet, we observed an interesting characteristic of the specific hardware on which we ran these experiments. We noticed that when we ran the model on a Tesla K80 GPU, the performance of TensorRT's optimized kernel was actually worse, as seen in Figure 3.5a. However, when run on a Tesla V100, a newer model with 33% extra GPU memory and HBM2, a much higher bandwidth memory system, the performance with the optimized kernel was significantly better. We see this in Figure 3.5b. This suggests that the reason for performance degradation when performing inference using these optimized kernels could be that the kernels themselves are too large to fit in memory, meaning serving is now memory-bound.

On the V100, we see significant improvements in the throughput and latency numbers for ResNet. TensorRT works specifically in the single model case, so we felt it stood to reason that a solution that performs similar optimizations in the multi-model serving case would definitely be useful.

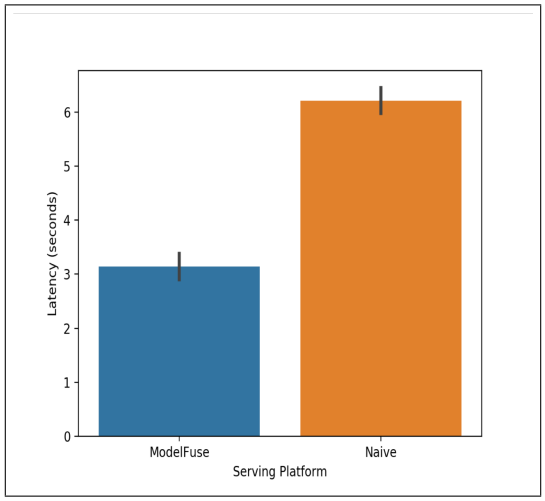
3.4.2 Preprocessing

Preprocessing is able to provide a significant win, especially in the case of an ensemble models.

We evaluated a scenario where an ensemble of AlexNet and ResNet was being naïvely served. Each model received the same batch of data at different endpoints, and the models were evaluated sequentially by the serving system on a Tesla K80 on a p2.xlarge machine. ModelFuse detected the Imagenet preprocessing signature, and it factored this out. We can see the performance characteristics in Figure 3.6b and Figure 3.6a. The performance wins demonstrate an almost 2× improvement in throughput while halving latency.



(a) Throughput improvements with ModelFuse.



(b) Latency improvements with ModelFuse.

Figure 3.6: ModelFuse performance improvements.

3.4.3 Layer Fusion

We performed layer fusion experiments with an ensemble of eight digit recognition models that evaluated on the MNIST dataset. The models used both dense and convolutional layers that were fused in a similar fashion to Figure 3.2. The experiment ran on a Tesla K80 GPU on a p2.xlarge machine. The results are displayed in Figure 3.7. We can make a couple of interesting observations here. First, we see expected significant wins on the smaller batch sizes. This is expected because the combination of the operators makes most sense in the case where the computation is not compute-bound. For smaller sizes, the transfer overhead dominates the amount of time taken to compute a response to the query. As the batch size increases, we see a dip in the 8, 16, and 32 batch sizes, which doesn't quite fit the trend we would expect. We ensured to run the experiments with over 1000 trials to rule out statistical error. We attribute this unexpected discrepancy to memory alignment issues. As batch size increases, the GPU utilization begins to saturate, and the operations become more compute-bound, demonstrating why we see smaller wins. It is worth noting that further improvement on implementing kernels that merge other operators will provide even more significant improvements here.

3.4.4 Clipper

All of the benefits of previous gains we saw from ModelFuse are enabled for users of Clipper, through the ability to run multiple models in one container. That contribution is independently useful to the open-source community. We can demonstrate

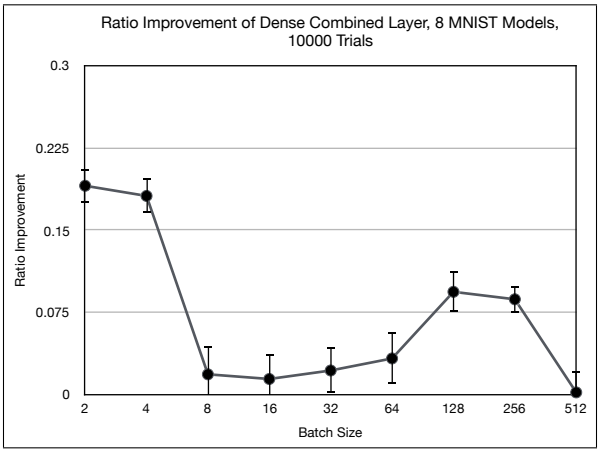


Figure 3.7: Latency improvements with ModelFuse through layer fusion.

the gains in the case where one would want to serve an instance of AlexNet and an instance of ResNet. In a standard Clipper, one would have to evaluate them sequentially, with no way of separating out queries. The full computation graph would have to run, which damages throughput in particular.

Figure 3.8 demonstrates the improvement that these specific changes in Clipper have added. Specifically, we see that the ModelFuse impact when serving both models sequentially isn't significant in Clipper. This proves the overhead of serving both models from one container does not negatively impact throughput. Serving just ResNet has a slightly higher throughput with ModelFuse - this is because ResNet is fairly large as compared to AlexNet. As a result, removing Alexnet's inference from the pipeline isn't as consequential. Meanwhile, AlexNet sees almost 3× improvement in throughput. This is a win because it means that smaller models can now be served in an ensemble on one container with much lower latency.

3.5 Conclusion

In this work, we looked to improve hardware utilization by evaluating multiple models on a single hardware resource. We presented ModelFuse, which successfully merged layer operators across models being served. We found that pre-processing aggregation yielded significant performance benefits. Additionally, we found that, while dynamic layer fusion offered significant speedups, the gains varied as a function of batch size. Larger batch sizes saw better performance than medium-sized batches, both of which demonstrated lower performance than small batch sizes. Finally, our experiments indicated that the Clipper modifications we made to support multi-model serving can offer significant improvements in reducing the serving latency for ensembles.

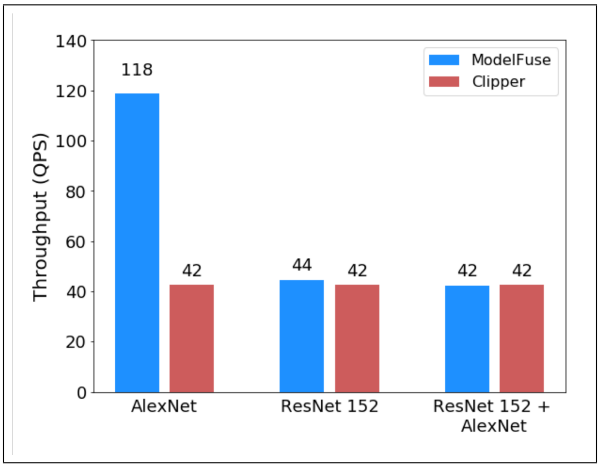


Figure 3.8: Clipper improvements allow for serving ensembles in a more granular way and increases throughput in the case of queries to just one of those models.

In the future, we plan to further extend our merging capabilities to support different types of kernels. Additionally, we plan to implement kernels at the CUDA level to avoid unnecessary overhead involved with the higher-level TensorFlow operator API. Finally, we plan to perform further experimentation to measure GPU utilization in order to better motivate our kernel development.

Chapter 4

Conclusion

Given the increasing dependence on machine learning in services across a variety of domains, it is important for researchers to not only consider training performance, but also the performance at inference time. In this paper, we explored two facets of machine learning serving.

With Shockwave, we began by exploring how we might leverage a number of hardware accelerators to improve serving latency and throughput for a single model. Shockwave's serving architecture involves sharding a model graph onto a network of hardware accelerators, which communicate with each other directly. We found that although we were leveraging specialized compute units, the bottleneck was still in computation rather than network performance.

Next, with ModelFuse, we explored how we might leverage a single hardware accelerator to best serve multiple models. ModelFuse merges multiple model graphs together at the operator level. This means that we combine similar kernels across models to save on kernel launch overhead and data transfer costs. We found that the types of operator fusions that we explored have promising performance improvements.

While these explorations each offer a specific view into how we might improve machine learning model serving, together, they offer a glimpse into what an ideal model serving system ought to do. Ideally, a serving system would be able to provision resources optimally to the models being served. If there is a single model being served, it should be able to adapt its resources to all serve that model. If there are more models being served, it should know how which graphs to combine and which to serve alone. The explorations mentioned above examine these problems in isolation and work towards a system that can efficiently and optimally provision resources for serving machine learning models.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Leo Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, August 1996.

[3] Doug Burger. Microsoft unveils project brainwave for real-time ai, Aug 2017.

[4] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. *CoRR*, abs/1612.03079, 2016.

[5] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 249–264, Berkeley, CA, USA, 2016. USENIX Association.

[6] Allison Gray, Chris Gottbrath, Ryan Olson, and Shashank Prasanna. Deploying deep neural networks with nvidia tensorrt, Apr 2017.

[7] Mark Harris. Gpu pro tip: Cuda 7 streams simplify concurrency, Jan 2015.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[9] 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*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.

[10] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[11] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. Firesim: Fpga-accelerated, cycle-accurate scale-out system simulation in the public cloud. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA18), Los Angeles, CA, June 2018*, 2018. To appear.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[13] Yann LeCun, Corinna Cortes, and Christopher JC Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2, 2010.

[14] Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, and Krste Asanović. The hwacha vector-fetch architecture manual, version 3.8.1.

[15] Deepak Narayanan, Keshav Santhanam, and Matei Zaharia. Accelerating model search with model batching. *SysML*, 2018.

[16] 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. *CoRR*, abs/1712.06139, 2017.

[17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[18] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014.

[19] Kaz Sato. Experience google's machine learning on your own images, voice and text, Sep 2016.

[20] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[21] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.