Rafiqi: A GPU-Based Deep Learning Model Serving System

Sammy Sidhu
University of California, Berkeley
Dept of Electrical Engineering and Computer Sciences
sammy@cs.berkeley.edu

Jordon Wing
University of California, Berkeley
Dept of Electrical Engineering and Computer Sciences
jordon@cs.berkeley.edu

Aakash Japi
University of California, Berkeley
Dept of Electrical Engineering and Computer Sciences
aakashjapi@cs.berkeley.edu

ABSTRACT

Interest in deep learning has exploded in recent years, due in no small part to substantial advances in the ability to harness Graphics Processors to train deep learning models efficiently. The next frontier in machine learning systems lies in harnessing GPUs to serve these models to users efficiently. We present Rafiqi, an optimized model serving system. Rafiqi provides a REST API to classify against an arbitrary number of user-provided models, and uses GPU caching and adaptive job batching to provide low-latency high-throughput classifications. We demonstrate substantial improvements in flexibility and performance over current systems.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Systems, Deep Learning

1. INTRODUCTION

In recent years, there has been an exponential growth in the general availability of data, which has in turn led to a major expansion in the commercial and research focuses on Machine Learning to extract meaning from this data. Classical machine learning methods, however, usually aren't optimal for the complex, multifaceted problems we want to solve today. For this reason, deep learning[1] has gained popularity, as it allow us to understand more complex and nuanced relationships in our data. In the past, deep neural networks were impractical to use at scale because of the computing power they demanded to train and predict, but this has changed significantly in recent years.

A lot of the recent research in machine learning has focused on utilizing the incredible parallelism afforded by Graphical Processing Units (GPUs) to make Deep Neural Networks much more practical. This is due to the fact that these deep networks are usually made up of a convolution stage as well as a fully connected neuron stage (which can be computed by using series of matrix multiplications), both of which are very parallelizable computations. Modern CPUs tend to have on the order of 4-16 cores but even consumer-grade GPUs have hundreds or thousands of cores. GPUs, then, lend themselves to deep learning, and vastly improve the speed of deep neural networks. Thus far, however, the vast majority of work has focused on using GPUs to optimize the training process [2], with very little literature on optimizing model serving; that is, making inferences from these trained neural networks in real time.

Currently, there are still several significant bottlenecks in performing GPU computations, all of which become more important in smaller scale computations, like model serving. First, any data that that is involved has to be transferred to the GPU via a Direct Memory Access (DMA) memcopy after a device memory allocation. This makes it hard to justify the use of GPUs for smaller computations since the overhead of memory allocation and memory transfer tend to be overshadow the decreased computation latency of the GPU. But, in larger computations (like model training), this cost is amortized over the entire process and becomes neglible. Since most modern neural networks are very deep (having many computation layers), a GPU can still pre-process and classify a single input two to three orders of magnitude faster than a CPU when including overhead.

To increase efficiency during training, bundles of inputs called batches are created. These batches are transferred together to the GPU and can be convolved and matrix multiplied in a 3D stack fashion. This typically utilizes the GPU to a much higher degree compared if each input was transferred, pre-processed and classified sequentially.

However this is hard to accomplish in a real-time setting due to the fact that we cannot predict future requests. This now becomes a tradeoff between throughput and latency: creating larger batches gives better throughput but a longer per-request latency, as we have to wait longer to fill a batch, while smaller batches give stronger latency guarantees but hurt throughput.

Deep Neural Networks tend to be in the order of hundreds of megabytes to gigabytes in size which can be an issue when

one wishes to perform inference on a variety of models. Current systems typically pin models in the GPU and send incoming requests to those models. This is a problem because we can only have at most tens of models running concurrently in a single system, and most are usually inactive.

We propose Rafiqi, a system that takes a different approach to model serving. Rafiqi is an example "model-serving as a service," where the user simply registers all of their models with our system, and we provide a full REST API from which they can make predictions and receive results. Rafiqi incorporates an adaptive batching system that changes with load, allowing the system to maximize throughput and minimize latency under all conditions. Rafiqi also includes a memory management system that allows a large amount models to be served simultaneously and asynchronously. This is accomplished by swapping the least used models out of the GPU when GPU memory overflows.

The outline of this paper is as follows: section 2 discusses the two major components in Rafiqi; sections 3 and 4 provide a detailed analysis of the design of each component; section 5 evaluates the performance of our system; section 6 discusses related work; we outline future work in section 7 and conclude in section 8.

2. HIGH-LEVEL ARCHITECTURE

Rafiqi is broadly divided into two main components. The first is a web server and job manager, which includes queuing, batching, and memory management. The second is an interface to Caffe[3], GPU contexts and memory management. This section provides a high-level discussion of each of these.

2.1 Application Level Architecture

Rafiqi defines two endpoints for clients: $/register$ and $/classify$. To classify an input, clients submit a request to $/classify$ with the name of the model to be used for classification and the input to be classified. The classify endpoint adds the job to a system-side queue, where the job is eventually passed to Caffe for classification. When classification completes, the system returns the top predictions for the input to the user. Classification requests for the same model are batched in groups that adjust in size based on recent demand for that model, since classification time is substantially improved when we can batch classification of multiple images for the same model.

Rafiqi supports classification requests for an arbitrary number of models. In order to be used in a classification request, the user must first send a request to the registration endpoint, which persistently stores the model information. Since models are typically quite large, typically only a subset of registered models are cached in shared memory on the GPU, in order to reduce classification times. Models needed for classification that aren't already in the GPU are transparently swapped with the least recently used models on demand.

2.2 GPU Management

In order to have a high performance system, the code was split into two layers, with a C API as the interface. To perform the actual inference of a data input through a neural network, we use Caffe, a Berkeley-built framework for deep neural nets. To perform the preprocessing for a data input such as resizing, thresholding or normalization, we use OpenCV which provides GPU accelerated image operations. Since Caffe is mostly targeted at training applications, we also had to implement a context pool that allows multiple instances of Caffe to be run simultaneously which is needed if we are to multiplex inferences of the same and different models. The final step was to add a memory management system into Caffe that allows to have more powerful control of where the memory is. This enables us to have an API that lets us move whole models to the GPU or CPU asynchronously and then pipeline prepreprocessing in an efficient manner.

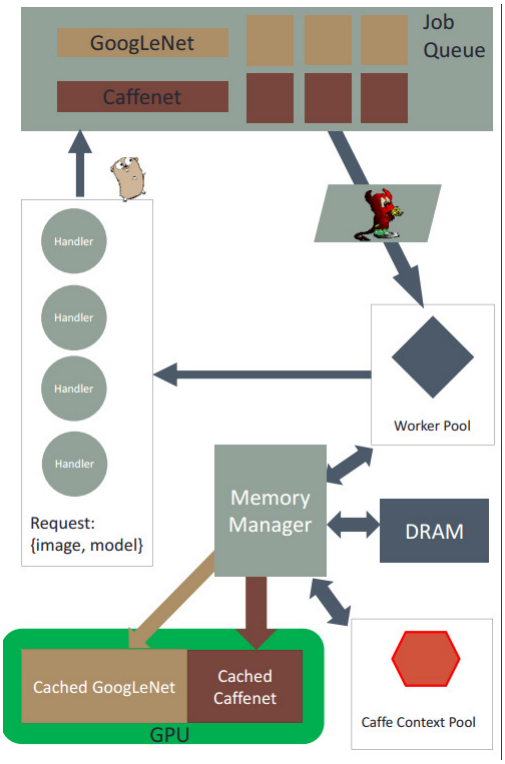


Figure 1: The architecture of Rafiqi

3. WEB SERVER AND JOB PROCESSING

Figure 1 provides a diagram of each of the components in Rafiqi. In this section, we discuss the components of the top half of Rafiqi in turn.

3.1 Registration

When registering a new model, clients provide a unique name for the model, and four files needed by Caffe: the model file, a means file, and a labels file-all on the order of kilobytes-, and the trained weights file-often on the order of 100-300MB. Registered models are persistently stored in BoltDB[4], a lightweight key-value store. When a model is registered, it is immediately initialized, before returning a response to the client. We expect that registration of new models will be relatively infrequent, and do not expect that the lengthy response time for registration will be a practical concern. Note also that registration requests can be

interleaved with classification requests without issue.

3.2 Job Processing

Jobs enter the system through our REST API, where each classification request is sent as a tuple of model identifiers (unique user-specified strings) and inputs (byte arrays). Our server forks a new thread for every incoming request for two reasons: first, each thread will block until it receives a response; second, we expect many concurrent requests to our system, precluding sequential processing of each job. Each thread (which we refer to as a "job handler") will package its incoming request into a job, which consists of a model identifier and a byte array of an image, and add it to our global job queue, and then wait for a classification response.

For our global job queue, we decided to use a "hashy linked-list". This is simply a standard linked-list augmented with a hashtable, where the hashtable maps strings to an array of pointers to the linked-list. We had a number of requirements for this queue. First, we needed constant time append and pop operations, because this queue would have to handle concurrent insertions and removals by potentially hundreds of threads. Even a small delay in these operations could cause a tremendous slowdown in our system, especially under high load. We also needed efficient access to separate sets of nodes in the linked list, because our system heavily utilizes per model request batching[5], requiring us to provide each worker with fast access to the set of jobs under its assigned model. The hashy-linked list ensures constant time append and pop operations: each append and pop operation consist of updating pointers in the linked list, and updating an array inside the hashtable, all of which is constant time. The hashtable also allows constant time accesses to all jobs pertaining to a certain model, letting each worker efficiently pull a set of jobs from the queue.

In addition to the hashy linked-list, we maintain a separate threadsafe queue of model identifiers. This represents all the models that currently have jobs requiring process. We'll refer to this as the "to-process queue."

The system currently has two main queues: the to-process queue, and the hashy linked-list. To bridge these two, we have a background daemon: a constantly-running thread that wakes up after a certain quanta and loops through every model. For each model, it checks if the number of currently queued requests if greater than some threshold; if it is, the daemon adds that model to the to-process queue, else it continues looping. This threshold will dynamically change for each model (which will be described in section 4.3). Once the daemon is done looping, it goes back to sleep for one quanta before repeating.

The daemon loops through all the models in Least Recently Used (LRU) order. It updates this LRU list whenever it adds a model to the to-process queue. We chose LRU mainly because it'll ensure fairness. Consider the case we pass through the list with some static ordering, and some model that we'll call A is besieged with requests. Every model that's after A in our ordering will be forced to wait for A's jobs to complete. If models after A also have a high number of requests, then latency will increase dramatically, which will not scale. With LRU ordering, however, model A will remain near the end of the list, so other models can still have their jobs processed without constantly waiting on A's large batches. This significantly reduces the increased latency that comes with high load, allowing the system to scale gracefully.

Once a model is added to a to-process queue, it needs to be assigned to a worker. We accomplish this using a dispatcher - a background thread that pulls model identifiers off of the to-process queue and assigns them to workers. We used a dispatcher, rather than having workers directly wait on the to-process queue, because it affords us flexibility. Through the dispatcher, we can make a simple interface to start or kill new workers, or view a worker's status. It also provides a layer of abstraction between workers and the queue, which allows us to change the queue without changing workers. And, since the dispatcher doesn't acquire any locks and is usually sleeping, the overhead it introduces is negligible.

We have k workers running at any one time, where k is the number of parallel model executions possible on each GPU (this is determined contexts, which we discuss in section 5). We only have k workers running simultaneously for two reasons. First, our workers are pretty minimal and have very few delays - besides the Caffe call made for classification, we don't have any long, blocking operations - so extra workers wouldn't be helpful in amortizing pre-classification costs. And, because only k classification jobs can execute in parallel, more workers would just lead to more blocked threads. Having fewer workers also means that our queue is less contested, which reduces concurrency delays.

Workers handle all communication with Caffe, and perform the actual classification, which is a blocking call into Caffe's C++ API. This prevents any other part of the system from having to block on C++, reducing other delays. Results are then taken directly from the classification call and passed onto the job's output queue. The Job Handler, which has been waiting on this queue, will now wake up and complete the request by returning the output.

3.3 Adaptive Batching

Our current system, with a fixed batching threshold, will not adapt to changes in system load. A fixed threshold significantly increases job latency during low-load situations, because jobs are forced to wait until enough accumulate to reach the threshold. It also severely reduces throughput under high load, because the threshold will be too low and batch sizes will be too small. So, it's necessary to have the threshold change with current system load to prevent unnecessary performance delays.

We had two major constraints for our adaptive threshold: it needs to increase gradually with load, but decrease very rapidly with a load falloff. A rapid decrease prevents load falloffs from artificially inflating Rafiqi's latency. Rapid decrease is also necessary because threshold that's too high is much more harmful than a low threshold that's too low: if the threshold is too low, the system will continue with reduced throughput, but if the threshold is too high, the system will actually stop making progress until the threshold is met. Then, a gradual increase is needed for a similar reason: we need to ensure that the system is almost always making forward progress. Also, a gradual increase will pre-

vent overreaction to short load spikes,

Our first attempt at an adaptive threshold was to use an additive increase and a multiplicative decrease, similar to TCP's windowing[6]. For the multiplicative decrease, we divided the threshold by two whenever the number of requests in the current quanta was less than the threshold. This works very well in creating a rapid decrease, and with our empirically tested threshold cap of 128, it only takes 7 quanta for the threshold to return to 0. Quanta are an adjustable parameter, but the usual is about 5 milliseconds, so it takes 35 milliseconds for the threshold to return to 0. Then, in the worst case, where the load goes from ≥ 128 to 1, we only add 35 milliseconds of latency.

For additive increase, when the number of requests for that model were greater than the threshold in the current quanta, we simply increase the threshold by the number of requests. This, however, isn't entirely desirable. Additive increase causes the threshold to rise too quickly, adding a lot of latency to the system as the threshold is constantly forced to decrease.

Instead, for additive increase, we decided to use exponential averaging [7]. This changes our update to the following (let t be the threshold for a certain model, and let c be the number of requests for a model in the last quanta):

$$t = \alpha * t + (1 - \alpha) * c$$

Here, alpha is a configurable parameter we can set; we found empirically that our best alpha was about 0.5.

This update significantly improved performance, as exponential averaging induces a bit of lag in our system, preventing the threshold from rising too quickly. It also smooths out sudden variations in load, insulating the threshold low-duration load spikes.

3.4 Memory Management

As noted above, initialization of a model for the first time takes on the order of 30 seconds per model. Additionally, in order to perform a classification on the GPU, models need to be transferred from RAM into shared memory on the GPU, and frame buffers for images need to be allocated. Models are often quite large: Caffe's reference image classification model is roughly 233MB. Since Rafiqi aims to efficiently support requests for an arbitrary number of models, we implement a memory manager to reduce initialization and transfers as much as possible.

Our memory manager uses the shared memory of a single GPU as a cache for model data. Users can configure the maximum amount of GPU memory to be consumed by Rafiqi, which defaults to the total memory of the GPU, and they can also provide a limit on the total number models loaded in the GPU at any time.

For the remainder of this section, we use "the GPU" to refer to the memory-managed GPU. Other GPUs connected to the same physical machine can be utilized for processing, but will not be managed by the memory manager.

The memory manager keeps a global state of all initialized models, and includes their location in RAM, as returned by *model_init*, and whether they are currently cached in the GPU. It also maintains an LRU list of all cached models in decreasing order of most recent classification time. This list is updated whenever a worker begins classification for a model.

It has a single method: $LoadModel$. $LoadModel$ takes a model name, and checks internal caching state for that model. If necessary, it transfers the model to the GPU cache. It updates the model's position in the LRU, and returns a representation of the model.

When a Rafiqi server starts, the memory manager first initializes all registered models. As a result, no classification request will ever incur the cost of an initialization. During this process, it also produces and stores persistently an estimate of the size of the model on the GPU using a sample of consumed GPU memory before and after loading the model. This estimate will be used to determine whether a model can be loaded onto the GPU during a classification. When the preloading process completes, it will leave an arbitrary set of all models loaded in the GPU cache when the server begins accepting requests.

When a classification request is processed by a worker, the worker calls $LoadModel$ to retrieve the internal model representation. If this model is not loaded into the cache, then the memory manager uses the estimate of the model size to determine whether the model could fit within the user-imposed size constraints on the cache. If not, then the memory manager begins evicting the least recently used models from the cache. To maintain consistency, we impose the constraint that only a single model can be in the process of being loaded into the GPU at a time. Once enough space has been freed on the GPU, the model is transferred to the GPU, internal state is updated, the LRU list is updated, and the model is returned to the worker.

3.4.1 Eviction

The process of moving models to and from the GPU is performed via Cgo calls. To avoid completely destroying a model during an eviction, and thereby requiring a complete initialization on the next classification request, we implemented a *to_gpu* function that transfers a model to the GPU and re-allocates the associated image buffers, and a *to_cpu* function that syncs all GPU-cached model to CPU RAM and frees all the data associated with a model that is stored on the GPU.

As noted in the previous section, transferring models to the GPU can be a lengthy process: a call to *to_gpu* for Caffe's reference image classification model can take up to 90ms, over 10x the time needed to perform a classification. In order to reduce the cost of this transfer as much possible, *to_gpu* is implemented asynchronously. Data is transferred by the GPU in the background. If data hasn't finished transferring by the time the classification begins, then the classification blocks on the completion of the transfer.

3.4.2 Limitations

The memory manager makes one crucial assumption: it assumes that no other process is using the GPU. When the memory manager samples GPU memory usage, it uses a CUDA API[8] which only supports sampling total GPU memory usage, not per-process memory usage. As a result, if another process is using the GPU, then it could cause Rafiqi's memory management decisions to be incorrect. However, since the memory manager only operates on a single GPU, other processes would be free to use other GPUs connected to the machine.

3.5 Classification

When the web server receives a classification request, it creates a new thread and passes request information to a request handling function. The request handler reads the input into memory, and creates a job containing the name of the model, the input, and a pointer to a thread-safe queue to be used for the communicating the result of classification. The request handler then sleeps until either data is placed onto the job's result queue, or a timeout expires, indicating that an error has occurred somewhere in the processing of the job. Once data is received on the job's result queue or the timeout expires, the data (or error message, in the case of timeout) is sent to the client in the form of JSON, and the request handling thread exits.

3.6 Implementation

The main web server is written in Google's Go[9], a statically-typed, compiled, garbage-collected language. Our decision to use Go for the web server was motivated by the following: first, Go has an efficient and highly scalable web server as part of the standard library; second, garbage collection and a simple type system enabled more rapid development at a minimal performance cost; third, we made heavy use of concurrency in the queuing and batching components, and Go includes a lightweight green thread implementation, along with concurrency primitives that made concurrency efficient and straightforward to reason about; finally, Go has a mechanism for interacting with C code called Cgo[10, 11]

The use of Cgo incurs a minor performance hit for every C call on the order of 10x, or 120ns per call, as a result of Cgo interactions with the Go scheduler[12]. Since calls into C happen only a small number of times per classification, and the time spent on classification dwarfs the time lost to the Cgo interface, we consider this an acceptable performance cost. It is also important to note that the use of Cgo adds some complexity to reasoning about concurrency: Go threads that make Cgo calls have to be executed in their own OS thread, so as to not block the Go scheduler. As a result, we ensured that only pre-allocated worker routines and registration request handlers make Cgo calls.

4. GPU MANAGEMENT

The GPU management component of Rafiqi is directly responsible for initializing and calling Caffe functions. For every model, k context objects per GPU are created and stored in a threadsafe queue, leading to a total of $k * N * M$, where N is the number of GPUs and M is the number of models. Context objects maintain a reference to an initialized neural network wrapper class that we created, which, in turn, maintains state about the loaded Caffe model. When a classification request is made, a single context is removed from that model's context pool. If none are available, then the worker blocks until one is. Once a context is acquired, the classification request is executed, and once it finishes, the context is returned to the context pool. If the model wasn't in the GPU memory, then the memory manager would call into GPU Management and start the transfer of the model before the classification process begins. All of the GPU management is written in C++, which gives us access to the C++ libraries of Caffe and OpenCV but also allows us to make a C interface that Cgo can call.

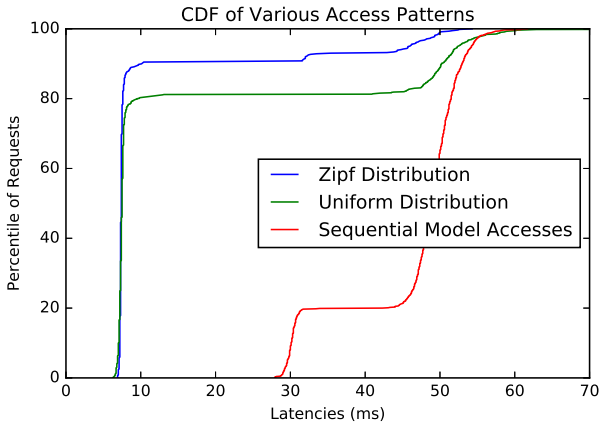
4.0.1 Modifying Caffe's Memory Management

Caffe does not provide a public interface for forcing transfers of neural networks between the CPU and GPU. So we created a fork of Caffe and modified how memory is handled. Internally, Caffe represents models as a sequence of network objects. Each network object is itself a sequence of blobs of data. These blobs implement methods to ensure synchronization between the CPU and the GPU. Our first modifications to the memory management allowed us to make blocking calls that first allocated the frame buffer in the GPU and then transferred the data synchronously. This on average took around 60ms on a Titan X on a 233MB model. Our second version allowed us to perform these actions asynchronously, which then only took 3ms on average. This mostly consisted of allocating the GPU memory. The main challenge in asynchronous memory calls was that we had to synchronize all state before the worker performed a forward pass.

Then, this asynchronous model allowed for preprocessing of inputs to be done in parallel, also asynchronously. To enable asynchronous transfers, we had to force the system to pin hardware pages when keeping the models in CPU DRAM. This allows the actual DMA to occur without page translation. This, however, increases the time to initialize a model, since pinned pages have to be memory allocated. Pinned allocation is much slower than allocating virtual memory, but gives us much better memory bandwidth utilization.

4.0.2 GPU Processing Pipeline

To improve Rafiqi's performance, we also use CUDA Streams to allow the GPU to overlap computations and memory operations. A CUDA stream allows us to perform actions asynchronously with the promise that the operations in a specific stream will be performed serially. This allows the GPU to overlap different streams to utilize more of the GPU's processing power and memory bandwidth. We create a CUDA stream for each context of each model. This allows each model to be transferred to the GPU and start pre-processing the input batches concurrently with other contexts of the same model and even other models. We modified Caffe such that we could pass in a CUDA stream to perform the memcopy on the owner context's stream. Pre-processing for the classifier is done by OpenCV[13] which provides vector operations for CUDA. To implement this in our application, we wrapped OpenCV asynchronous sreams with our CUDA stream that we use per each context. This gives the promise that all operations for each context occur serially, which we need to perform preprocessing, but that the preprocessing of one context may happen before or after the



(a)

Figure 2: An analysis of the latencies of various access patterns of 5 models when the cache is limited to a maximum of 4 at any given time.

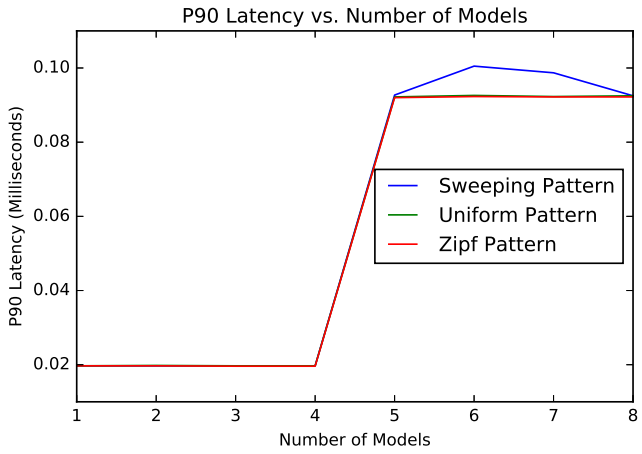


Figure 3: 90th percentile latency as a function of the number of models being accessed, with a maximum of 4 being cached in the GPU at one time.

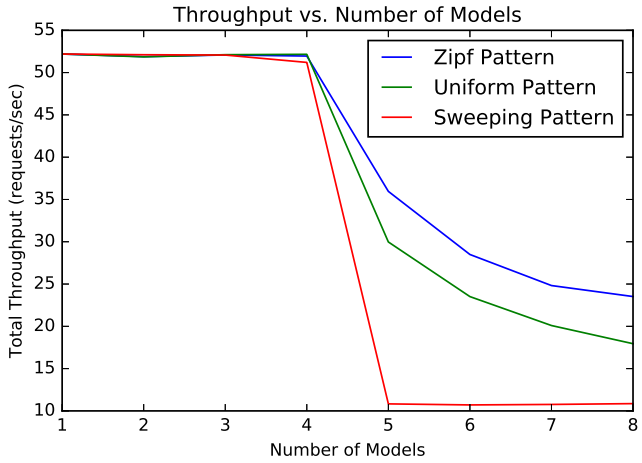


Figure 4: Classifications/Second as a function of the number of models being accessed, with a maximum of 4 being cached in the GPU at one time.

memcopy/preprocessing of another. Without this, all operations would have to be performed serially, which is a major performance issue.

4.0.3 *Context Pool and Slab Allocator*

For each model, we create a threadsafe, blocking queue of contexts that gives us multiple replicas of a single neural network. This allows us to have multiple workers performing batches of a single model concurrently, which is helpful when a model is extremely popular. But, we don't want excessively large batches, however, because that would give inconsistent latencies. Instead, whenever a context is initialized or moved to the GPU, we allocate a new frame buffer of GPU memory. This is used by a slab allocator we made. We then get the benefit of our input matrices being padded and cache aligned, as well as having the data in a fixed place in the GPU RAM. This expedites asynchronous operations during preprocessing. Then, when a worker acquires a context, slab allocator's pointer is reset to the start of the allocated memory. Finally, when the memory manager evicts a model from the GPU, we free the allocated memory of that frame buffer.

5. EVALUATION

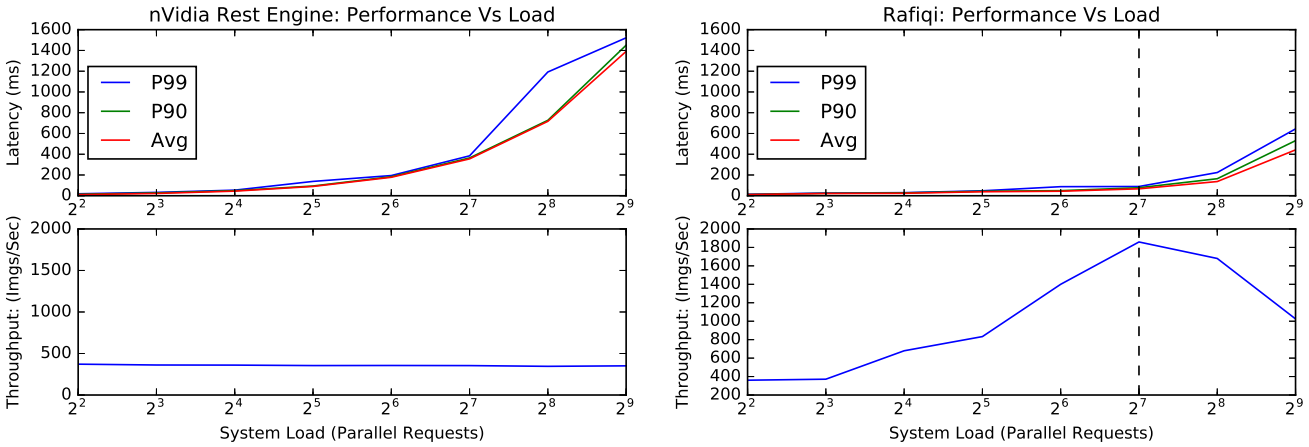
In this section we present the evaluation of our model-serving system. All benchmarks are run using a slightly modified version of Boom[14], a Google-sponsored Go implementation of the standard Apache Benchmarking tool, ab. We use Boom to send a request to the classification endpoint with an image in the request body

5.0.1 *Concurrent Loads*

Our first benchmark was performed on a UC Berkeley Aspire Lab millennium machine, with eight Nvidia Titan X GPUs and a Xeon Intel CPU. We sent an increasing number of concurrent requests for a single model at our system and Nvidia's GPU REST Engine, and recorded how throughput and latency changed as concurrency increased. For this experiment, we used a maximum batch size of 128 and a quanta of 10 milliseconds. As seen in Figure 3, what we found was that as concurrency increased, Nvidia's system didn't show any increase in throughput, because it doesn't batch requests, but it had major increases in latency due to its sequential processing of each request. Meanwhile, our system adapted to changing load - as requests increased, the throughput improved while the latency remained fairly stable. After 2^7 active requests, our throughput began to decrease and latency increased significantly - this is our "inflection point," where batch sizes become too high and cause the system to spend an inordinate amount of time waiting on worker threads. One possible resolution to this issue is using adaptive quanta, a feature we discuss in depth in section 7.

5.0.2 *Access Patterns*

Our next benchmark tested the latency of our system when models overran GPU memory. This benchmark was run on the UC Berkeley AMPLab millennium machines, which are equipped with dual 12-core Intel Xeons and a single NVIDIA



(a) (b)

Figure 5: A comparison of Rafiqi and NVIDIA's GPU REST engine for varying concurrent loads on a single Titan X GPU. The dotted line indicates the steady state of the system, where throughput is maximized and the response return rate is equal to the incoming request rate.

Tesla K20c GPU. All models are identical versions Caffenet retrieved from the Caffe model zoo[15].

In all of the following benchmarks, we limit the maximum number of cached models to at most four. We use a batching daemon quanta of 5ms, 1 GPU context, a max batch size of 8, and 4 worker threads.

We register five copies of Caffenet, forcing our system to swap models in and out of memory. Then, we set our request concurrency to one and used three different access patterns to benchmark our system. We used 1000 requests for every model. First, we used a sequential access pattern, which is the worst case for an LRU memory manager, because it forces the system to swap models at every request ("thrashing"). Then, we used a uniform, random distribution to choose models to request. Finally, we use a power law (Zipf) distribution drawn from [16] using alpha=1.1, which is our approximation of a real-world scenario: the top model receives 50% of requests, the next receives 37.5%, and so on.

The results of this can be seen in Figure 2. As expected, our sequential accesses performed the worst. But, while we had a much higher average latency for this, our 99th percentile was almost the same as the other two, indicating that our GPU memory operations are a fairly static cost that can be amortized over a larger volume of requests. The uniform access pattern, on the other hand, has a much better average latency than the sequential access pattern, but it increases rapidly. However, this is expected: for any given request in a uniform distribution, there's a $1/N$ chance of triggering a swap, where N is the number of registered models. Uniform has a much higher 99th percentile latency than 80th percentile latency, because $1/N = 1/5 = 20\%$ of requests triggered evictions and swaps. Finally, the zipf distribution performed the best because it triggered the fewest swaps, as 50% of requests went to the same model. Zipf also has a long tail, however, because its less-accessed models always trigger evictions.

In our next benchmarks, we plot latency and throughput as a function of the number of models being accessed. We access models with a concurrency level of 1 request in flight at a time. Figures 3 and 4 show the results of this test. The sequential access test is a worst-case evaluation, as it requires a swap on every single incoming request (with more than 4 models and it shows: once models begin swapping, throughput is reduced to 20% of a run without any swapping, and latency is 5 times higher. Since they experience fewer swaps, uniform random and zipf distributions both perform better in throughput, taking roughly a 50% throughput hit, but taking an latency hit that is roughly equal to the worst-case. The substantial latency increase for all patterns is the result of a lack of concurrency in the benchmark: anytime a request swaps, the classification takes 90 additional milliseconds, and no other requests can happen at the same time. Unfortunately, we were unable to benchmark high-concurrency model swapping due to a rare bug that occurred only occasionally during highly-concurrent model-swapping access patterns.

6. RELATED WORK

There have been a number of model-serving system released relatively recently. Many of them are serving systems based on established deep learning frameworks. We discuss several such systems in turn.

Nvidia has invested heavily in GPU-based deep learning. Rafiqi is based on Nvidia's GPU REST Engine[17], which provided the initial Go-C++ framework and initial architecture of the Caffe interface. At present, Nvidia's system is designed to optimize low-latency serving of a single model. As a result, its memory management is limited: a single model is loaded at start-time onto all GPUs, and remains there for the duration of the system's runtime. Since Nvidia prioritized low-latency, their system does not batch requests, although they list this as a candidate future fea-

ture. As shown in section 5, our implementation of batching only modestly raised latency while providing a substantial throughput gain.

However, the GPU REST engine provides multi-GPU support out of the box, while Rafiqi does not currently support utilizing multiple GPUs. This enables Nvidia's system to support higher throughput on multi-GPU systems. We expect that the introduction of multi-GPU support to Rafiqi would substantially improve our performance on such systems. Support for multi-GPU systems in Rafiqi is discussed in detail in section 7.

Tensorflow Serving[18] is a model serving system built on top of Google's deep learning library, Tensorflow. At present, Tensorflow is aimed at large-scale datacenter deployment[19], and does not focus on optimizing single-node performance. Due to an extremely complex setup procedure, we were unable to produce direct comparative evaluations of Tensorflow Serving. However, previous comparative evaluations of deep learning frameworks have indicated that, for large batch sizes on the GPU, a single forward pass in Tensorflow takes roughly twice as long as a forward pass in Caffe[20].

7. FUTURE WORK

We also see room for improvement in the following areas:

7.1 Multi-GPU Support

Many production systems are equipped with multiple high-performance GPUs. Future work could markedly improve the system by load balancing requests across GPUs, as in [21]. In addition, the memory management system could be modified to cache models across GPUs, to increase cache capacity. The scheduler and batching system could incorporate awareness of model cache locality, and prioritize scheduling classification jobs on the GPU where models are cached and where utilization is low.

7.2 Higher Concurrency Model Swapping

Our evaluation demonstrated that model-swapping causes a severe performance penalty without further optimization. In addition to better support for multiple GPUs, future iterations of Rafiqi would benefit from a more sophisticated locking framework for the memory manager, in order to permit higher concurrency of classifications and swaps during model-swapping access patterns. In addition, the scheduler could be made aware of what asynchronous *to_gpu* transfers are in progress, and incorporate the GPU transfer penalty into its batching and scheduling decisions, by avoiding scheduling jobs for a model if an asynchronous transfer is still in progress.

7.3 Adaptive Quanta

Currently, the batching daemon has a static quanta. However, this leads to performance issues: if the quanta is too large, requests will accumulate on the queue, while if the quanta is too small, batch sizes are too small, causing throughput to suffer. An interesting optimization would be to have an adaptive quanta, using exponential averaging, that changes based upon the time it takes for a single classify request. This would involve having a different daemon per model, but would ensure that the daemon wakes up right after the classify completes, creating less waiting time for the worker threads.

8. CONCLUSION

This paper presents the design and architecture of Rafiqi, a model-serving system optimized to reduce the latencies inherent in using the GPU for processing in a low-latency environment. We demonstrate that Rafiqi's batching system substantially improves throughput over NVIDIA's GPU REST engine in highly concurrent single-GPU environments, at a minimal latency cost, while simultaneously supporting requests for and caching of an arbitrary numbers of models.

9. ACKNOWLEDGMENTS

The authors would like to thank Prof. John Kubiatowicz and Prof. Joey Gonzalez of UC Berkeley for their advice and guidance. We would also like to thank the UC Berkeley AMPLab and Aspire Lab for allowing us to use substantial computing resources for this project.

10. REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 05 2015.

[2] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, (New York, NY, USA), pp. 873–880, ACM, 2009.

[3] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[4] "boltdb/bolt."

[5] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja, "Lambda architecture for cost-effective batch and speed big data processing," *2015 IEEE International Conference on Big Data (Big Data)*, 2015.

[6] V. Jacobson, "Congestion avoidance and control," in *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, (New York, NY, USA), pp. 314–329, ACM, 1988.

[7] "Moving average and exponential smoothing models."

[8] N. Inc, "Cuda runtime api :: Cuda toolkit documentation."

[9] G. Inc, "The go programming language."

[10] A. Gerrand, "C? go? cgo!."

[11] G. Inc, "Command cgo."

[12] "The cost and complexity of cgo | blog | cockroach labs," Sep 2015.

[13] G. Bradski *Dr. Dobb's Journal of Software Tools*.

[14] "Go boom."

[15] "Caffe model zoo."

[16] "numpy.random.zipf."

[17] "Nvidia gpu rest engine (gre)," Sep 2015.

[18] "Architecture overview."

[19] "Announcing tensorflow 0.8 - now with distributed computing support!."

[20] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of caffe, neon, theano,

and torch for deep learning," *CoRR*, vol. abs/1511.06435, 2015.

[21] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao, "Dynamic load balancing on single- and multi-gpu systems," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, April 2010.