Nexus: A GPU Cluster Engine for Accelerating Neural Networks Based Video Analysis

Haichen Shen
University of Washington
haichen@cs.washington.edu

Lequn Chen
University of Washington
lqchen@cs.washington.edu

Yuchen Jin
University of Washington
yuchenj@cs.washington.edu

Liangyu Zhao
University of Washington
liangyu@cs.washington.edu

Bingyu Kong
Shanghai Jiao Tong University
bingyukong97@gmail.com

Matthai Philipose
Microsoft Research
matthaip@microsoft.com

Arvind Krishnamurthy
University of Washington
arvind@cs.washington.edu

Ravi Sundaram
Northeastern University
koods@ccs.neu.edu

Abstract

We address the problem of serving Deep Neural Networks (DNNs) efficiently from a cluster of GPUs. In order to realize the promise of very low-cost processing made by accelerators such as GPUs, it is essential to run them at sustained high utilization. Doing so requires cluster-scale resource management that performs detailed scheduling of GPUs, reasoning about groups of DNN invocations that need to be co-scheduled, and moving from the conventional whole-DNN execution model to executing fragments of DNNs. Nexus is a fully implemented system that includes these innovations. On large-scale case studies on 16 GPUs, Nexus shows 1.8-12.7× better throughput than state-of-the-art systems while staying within latency constraints >99% of the time. A long-running multi-application deployment on an 88-GPU cluster violates latency SLOs on 1.3% of requests and stays within 32% of an aggressive lower bound on GPU usage.

1 Introduction

Consider a cloud-scale video analysis service that allows thousands of tenants to analyze thousands of streams each concurrently. Increasingly, the core computations for this workload are Deep Neural Networks (DNNs), which are networks of dense linear algebra computations. Specialized hardware accelerators for DNNs, in the form of Graphic Processing Units (GPUs, which this paper focuses on) and even more specialized Tensor Processing Units (TPUs) have emerged in the recent past. GPU accelerators process DNNs orders of magnitude faster and cheaper than CPUs in many cases. However, GPUs are expensive and very-high-capacity: modern devices *each* provide over 100 TFLOPS. Cost-savings from using them depends critically on operating them at sustained high utilization. A fundamental problem therefore is to distribute the large incoming workload onto a cluster of accelerators at high accelerator utilization and acceptable latency. We address this problem in this paper.

Conceptually, this problem can be thought of as sharding inputs via a distributed frontend onto DNNs on backend GPUs. Several interacting factors complicate this viewpoint. First, given the size of GPUs, it is often necessary to place different types of networks on the same GPU. It is then important to select and schedule them so as to maximize their combined throughput while satisfying latency bounds. Second, many applications consists of *groups* of DNNs that feed into each other. It is important to be able to specify these groups, and to schedule the execution of the entire group on the cluster so as to maximize performance. Third, it is well known that dense linear algebra computations such as DNNs execute much more efficiently when their inputs are *batched* together. Batching fundamentally complicates scheduling and routing because (a) it benefits from cross-tenant and cross-request coordination and (b) it forces the underlying bin-packing-based scheduling algorithms to incorporate batch size. Fourth, the increasingly common use of *transfer learning* in today's workloads has led to specialization of networks, where two tasks that formerly used identical networks now use networks that are only mostly identical. Since batching only works when multiple inputs are applied to the *same* model in conventional DNN execution systems, the benefits of batching are lost.

Nexus is a GPU cluster for DNN execution that addresses these problems to attain high execution throughput under latency Service Level Objectives (SLOs). It uses three main techniques to do so. First, it relies on a novel *batching-aware scheduler* (Section 6.1) that performs bin packing when the balls being packed into bins have variable size, depending on the size of the batch they are in. This schedule specifies the GPUs needed, the distribution of DNNs across them and the

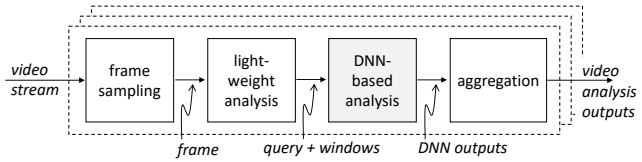


Figure 1: A typical vision processing pipeline. Nexus is designed to provide DNN-based analysis for tens of thousands of streams.

order of their execution so as to maximize execution throughput while staying within latency bounds. Second, it allows groups of related DNN invocations to be written as *queries* and provides automated query optimization to assign optimal batch sizes to the components of the query so as to maximize overall execution throughput of the query while staying within its latency bounds (Section 6.2). Finally, Nexus breaks from orthodoxy and allows batching of *parts of networks* with different batch sizes. This enables the batched execution of specialized networks (Section 6.3).

Nexus is completely implemented as a containerized system deployable on a commercial cloud, and comprises of roughly 10k lines of C++. We have deployed Nexus on a 88-GPU cluster. On focused 16-GPU experiments compared with existing DNN serving systems (Tensorflow Serving [1] and Clipper [8]), we show improvements in throughput of 1.8-4.4× on a traffic monitoring case study, and 1.8-12.7× on a game-stream analysis case study, while satisfying latency SLOs (i.e., achieving a "good rate" of) >99% of the time. On a much larger experiment on a 88-GPU cluster, 7 applications and 11 different models, Nexus achieves a good rate of >98.7% while maintaining similar high throughputs.

2 Background

A vision-based application aggregates visual information from one or more video streams using custom "business" logic. Each stream is processed using a pipeline similar to that in Figure 1. CPU-based code, either on the edge or in the cloud, selects *frames* from the stream for processing, applies business logic to identify what parts (or *windows*) of the image need deeper analysis, applies a DNN *query* to these windows, and aggregates the results in an application specific way, often writing to a database. A query may represent a single DNN applied to the window, but often it may represent a sequence of dependent DNN applications, e.g., running an object detector on the window and running a car make/model detector on all sub-windows determined to be cars.

Typically, a stream is sampled a few times a second or minute, and the DNN query should complete execution in tens to hundreds of milliseconds (for "live" applications) or within several hours for ("batch" applications). The execution of DNNs dominates the computation pipeline, and the cost of executing them dominates the cost of the vision service. Nexus provides a standalone service that implements the DNN-based analysis stage for vision pipelines.

Model	CPU lat. (ms)	GPU lat. (ms)	CPU cost ($) (0.1TF peak)	TPU cost ($) (180TF peak)	GPU cost ($) (125TF peak)
Lenet5	6	<0.1	$0.01	$0.00	$0.00
VGG7	44	<1	0.13	0.01	0.01
Resnet50	1130	6.2	4.22	0.48	0.12
Inception4	2110	7.0	8.09	0.93	0.23
Darknet53	7210	26.3	24.74	2.85	0.70

Table 1: DNN execution latencies and estimated costs per 1000 invocations.[1] Acceleration may be necessary to meet latency deadlines, but can also be cheaper, given low cost/TFLOPS.

2.1 Accelerators and the challenge of utilizing them

As Table 1 shows, a key to minimizing the cost of executing DNNs is the use of specialized accelerators such as GPUs and TPUs, which are highly optimized to execute the dense linear algebra computations that comprise DNN models. The table shows the execution latency and the dollar cost of 1000 invocations for a few common models on CPUs and GPUs. Execution times on CPUs can be orders of magnitude slower than that on GPUs. For many applications, therefore, latency constraints alone may dictate GPU-accelerated execution.

Perhaps more fundamentally, GPUs and TPUs promise much lower cost per operation than even highly accelerated CPUs: Table 1 lower-bounds the cost of executing a model by assuming that models can be executed at peak speed on each platform. Even compared to state of the art CPUs, accelerators can yield a cost advantage of up to 9× (for TPUs) and 34× (for GPUs). On the other hand, accelerators have extremely high computational capacity (e.g., 125 TFLOPS for the NVIDIA V100). *To realize their cost savings, it is critical to sustain high utilization of this capacity*. Sustaining high utilization is hard, however. For instance, the LeNet model of Table 1 consumes 20 MOPs to run, implying that *a single* V100 would require 125 TFLOPS ÷ 20 MOPs = 6.25M inputs/second to run at full utilization!

No single stream, or even most applications, can yield such rates. By aggregating inputs across streams and applications, Nexus is designed to funnel adequate work to each accelerator. However, as we discuss next, having "enough" work is not sufficient to achieve high utilization: it is important to group the right type of work in the right place.

[1] Per-device prices for 1000 invocations assuming peak execution rates on on-demand instances of AWS c5.large (Intel AVX 512), p2.xlarge (NVIDIA K80), p3.2xlarge (NVIDIA V100) and GCP Cloud TPU.

2.2 Placing, packing and batching DNNs

DNNs are networks of dense linear algebra operations (e.g., matrix multiplication and convolution), called *layers* or *kernels*. Networks are also called *models*. By default, the GPU simply executes the kernels presented to it in the order received. The kernels themselves are often computationally intensive, requiring MFLOPs to GFLOPs to execute, and range in size from one MB to hundreds of MBs. These facts have important implications for GPU utilization.

First, loading models into memory can cost hundreds of milliseconds to seconds. When serving DNNs at high volume, therefore, it is usually essential to *place* the DNN on a particular GPU by pre-loading it on to GPU memory and then re-using it across many subsequent invocations. Placement brings with it the traditional problems of efficient *packing*. Which models should be co-located on each GPU, and how should they be scheduled to minimize mutual interference?

Second, it is well known that processor utilization achieved by kernels depends critically upon *batching*, i.e., grouping input matrices into higher-dimensional ones before applying custom "batched" implementations of the kernels. Intuitively, batching allows kernels to avoid stalling on memory accesses by operating on each loaded input many more times than without batching. On an NVIDIA GTX1080, batching improves the throughput of model execution by 4.7-13.3× for batch sizes of 32 for VGG, ResNet, and Inception models relative to executing them individually. Further, our measurements indicate that we can often use a linear model to fit the batched execution latency as follows:

$$\text{batch\_lat}(b) = \alpha b + \beta, \tag{1}$$

where β is the fixed cost to invoke a model and α is the cost of each additional task in the batch. Large batches amortize the fixed cost β and help achieve higher throughputs.

Although batching is critical for utilization, it complicates the resource allocation and scheduling decisions made inside of a cluster. We elaborate on these issues in Section 4. Further, batching is conventionally only feasible when the same model is invoked with different inputs. For instance, we expect many applications to use the same well-known, generally applicable, models (e.g., Resnet50 for object recognition). However, the generality of these models comes at the price of higher resource use. It has become common practice [12, 22] to use smaller models *specialized* (using "transfer learning") to the few objects, faces, etc. relevant to an application by altering ("re-training") just the output layers of the models. Since such customization destroys the uniformity required by conventional batching, making specialized models play well with batching is often critical to efficiency.

3 Related Work

The projects most closely related to Nexus are Clipper [8] and Tensorflow Serving [1]. Clipper is a "prediction serving system" that serves a variety of machine learning models including DNNs, on CPUs and GPUs. Given a request to serve a machine learning task, Clipper selects the type of model to serve it, batches requests, and forwards the batched requests to a backend container. By batching requests, and adapting batch sizes online under a latency SLO, Clipper takes a significant step toward Nexus's goal of maximizing serving throughput under latency constraints. Clipper also provides approximation and caching services, complementary to Nexus's focus on executing all requests exactly but efficiently. Tensorflow Serving can be viewed as a variant of Clipper that does not provide approximation and caching, but also has additional machinery for versioning models.

To the basic batched-execution architecture of Clipper, Nexus builds along the dimensions of *scale*, *expressivity* and *granularity*. These techniques address the challenges brought up earlier in this section, and thus reflect Nexus's focus on executing DNNs on GPUs at high efficiency and scale.

Scale: Nexus provides the machinery to scale serving to large, changing workloads. In particular, it automates the allocation of GPU resources and the placement and scheduling of models across allocated resources. It provides a distributed frontend that scales with requests. These functions are performed on a continuing basis to adapt to workloads.

Expressivity: Nexus provides a query mechanism that (a) allows related DNN execution tasks to be specified jointly, and (b) allows the user to specify the latency SLO just at the whole-query level. Nexus then analyzes the query and allocates latency bounds and batch sizes to constituent DNN tasks so as to maximize the throughput of the whole query.

Granularity: Where Clipper limits the granularity of batched execution to whole models, Nexus automatically identifies common *subgraphs* of models and executes them in a batch. This is critical for batching on specialized models, which often share all but the output layer, as described previously.

Serving DNNs at scale is similar to other large-scale short-task serving problems. These systems have distributed front ends that dispatch low-latency tasks to queues on the backend servers. Sparrow [24] focuses on dispatch strategies to reduce the delays associated with queuing in such systems. Slicer [6] provides a fast, fault-tolerant service for dividing the back end into shards and load balancing across them. Both systems assume that the backend server allocation and task placement is performed at a higher (application) level, using cluster resource managers such as Mesos [17] or Omega [29]. Nexus shares the philosophy of these systems of having a fast data plane that dispatches incoming messages from the frontend to backend GPUs and a slower control plane that performs more

Model A			Model B			Model C		
Batch	Lat	Req/s	Batch	Lat	Req/s	Batch	Lat	Req/s
4	50	80	4	50	80	4	60	66.7
8	75	107	8	90	89	8	95	84
16	100	160	16	125	128	16	125	128

Table 2: Batching profiles for models used in the example. Lat is the latency (ms) for processing a batch, and Req/s is the throughput achieved.

heavyweight scheduling tasks, such as resource allocation, packing and load balancing. Also, the Nexus global scheduler communicates with cluster resource managers to obtain or release resources.

Much work has focused on producing faster models often at small losses in accuracy [5, 27, 34] Further, models of varying accuracy can be combined to maintain high accuracy and performance [8, 15, 18, 20, 30]. Nexus views the optimization, selection and combination of models as best done by the application, and provides no special support for these functions. Our mechanisms are also orthogonal to the scheduling, placement, and time-sharing mechanisms in training systems [14, 26, 31] since DNN serving has to be performed within tight latency SLOs while maintaining high utilization.

4 Scheduling Problems in Batched Model Serving

Batched execution of models improve GPU utilization but also raises many challenges in determining how cluster resources are allocated to different applications and how to batch model invocations without violating latency constraints.

Fundamentally, the algorithm for packing models on GPUs needs to take into account the fact that the processing cost of an input is "squishy", i.e., it varies with the size of the batch within which that input is processed. Further, the latency of execution also depends on the batch size. This new version of bin packed scheduling, which we dub *squishy bin packing*, needs to reason explicitly about batching. Second, batching complicates query processing. If a certain latency SLO (Service Level Objective) is allocated to the query as a whole, the system needs to partition the latency across the DNN invocations that comprise the query so that each latency split allows efficient batched execution of the related DNN invocation. We call this *complex query scheduling*. Third, in addition to batching-aware resource allocation, the runtime dispatch engine also has to determine what requests are batched and what requests are dropped during periods of bursty arrival. We now use examples and measurements to elaborate on these underlying scheduling and resource allocation challenges.

4.1 Squishy bin packing

Consider a workload that consists of three different types of tasks that invoke different DNN models. Let the desired latency SLOs for tasks invoking models A, B, and C be 200ms,

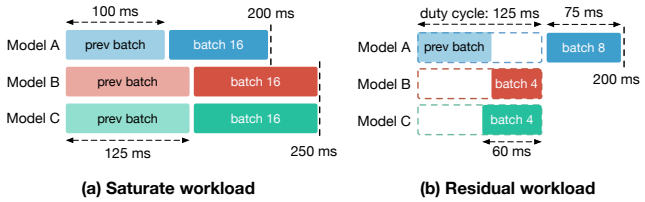


Figure 2: Resource allocation example.

250ms, and 250ms, respectively. Table 2 provides the batch execution latency and throughput at different batch sizes (i.e., the "batching profile") for each model.

We first explore the basic scenario where all three types of tasks are associated with high request rates so that multiple GPUs are required to handle each task type. To maximize GPU efficiency, we need to choose the largest possible batch size while still meeting the latency SLO. Note that the batch execution cost for a given task type cannot exceed half of the task's latency SLO; a task that missed being scheduled with a batch would be executed as part of the next batch, and thus its latency would be twice the batch execution cost. For example, the latency SLO for Model A tasks is 200 ms, so the maximum batch size we can use is 16. Therefore, the maximum throughput that Model A can achieve on a single GPU is 160 reqs/sec, and the number of GPUs to be allocated for Model A should $r_A/160$, where r_A is the observed request rate. Similarly, the number of GPUs for models B and C should be $r_B/128$ and $r_C/128$, where r_B and r_C are the request rates for models B and C respectively. Figure 2(a) depicts the desired schedules for the different models.

We next consider a situation where the request rates for the models are lower, with each one requiring less than a GPU. In this case, the scheduler needs to consolidate multiple types of DNN tasks onto the same GPU to optimize resource utilization. Consider a workload where Model A receives 64 reqs/sec, and Model B and Model C receive 32 reqs/sec each. We consider schedules where multiple models are assigned to a GPU. The GPU then executes batches of different types of models in a round robin manner, and it cycles through them over a time period that we refer to as the *duty cycle*. The worst case latency for a task is no longer twice the batch execution cost but rather the sum of the *duty cycle* and the batch execution cost for that task type.

Given this setup, Model A tasks can be scheduled in batches of 8 as part of a duty cycle of 125ms; note that the resulting throughput is the desired rate of 64 reqs/sec, the batch execution cost for 8 tasks is 75ms, and the worst case execution delay of 200ms matches the latency SLO (see Figure 2(b)). We then check whether the GPU has sufficient slack to accommodate tasks associated with models B or C. Within a duty cycle of 125ms, we would need to execute 4 tasks of either B or C to meet the desired rate of 32 reqs/sec. The

Model X		Model Y	
Lat	Reqs/s	Lat	Reqs/s
40	200	40	300
50	250	50	400
60	300	60	500

Figure 3: Batch execution latency (ms) and throughput of models.

Latency budget		Avg Throughput (reqs/s)		
X	Y	$\gamma = 0.1$	$\gamma = 1$	$\gamma = 10$
40	60	192.3	142.9	**40.0**
50	50	235.3	**153.8**	34.5
60	40	**272.7**	150.0	27.3

Figure 4: The average throughput with three latency split plans for varying γ.

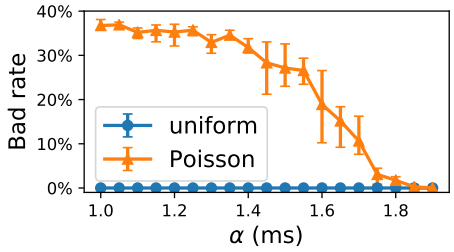


Figure 5: Performance of lazy dropping.

batch execution cost of 4 model B tasks is 50ms, which can fit into the residual slack in the duty cycle. On the other hand, a batch of 4 model C tasks would incur 60ms and cannot be scheduled inside the duty cycle. Further, the worst case latency for model B is the sum of the duty cycle and its own batch execution cost, 175ms(= 125 + 50), which is lower than its latency SLO 250ms. Thus, it is possible to co-locate Model B, but not Model C, on the same GPU as Model A.

We now make a few observations regarding the scenario discussed above and why the associated optimization problem cannot be addressed directly by known scheduling algorithms. First, unlike vanilla bin-packing that would pack fixed-size balls into bins, the tasks here incur lower costs when multiples tasks of the same type are squished together into a GPU. Second, in addition to the capacity constraints associated with the GPU's compute and/or memory capabilities, there are also latency constraints in generating a valid schedule. Third, there are many degrees of freedom in generating a valid schedule. The batch size associated with a model execution is not only a function of the request rate but also of the duty cycle in which the batch is embedded. In Section 6.1, we describe how to extend traditional algorithms to handle this setting.

4.2 Complex query scheduling

Applications often comprise of dependent computations of multiple DNN models. For example, a common pattern is a detection and recognition pipeline that first detects certain objects from the image and then recognizes each object. The developer will specify a latency SLO for the entire query, but since the system would host and execute the constituent models on different nodes, it would have to automatically derive latency SLOs for the invoked models and derive schedules that meet these latency SLOs. We discussed the latter issue in the previous example, and we now focus on the former issue.

Consider a query that executes Model X and feeds its output to Model Y. Suppose we have a 100ms latency budget for processing this query, and suppose that every invocation of X yields γ outputs (on average). When $\gamma < 1$, X operates as a filter; when $\gamma = 1$, X maps an input to an output; when $\gamma > 1$, X yields multiple outputs from an input (e.g., detection of objects within a frame).

Assume that Figure 3 depicts the batch execution latency and throughput of models X and Y. The system has to decide what latency SLOs it has to enforce on each model such that the overall latency is within 100ms and the GPU utilization of the query as a whole is maximized. For this example, we consider a limited set of latency split plans for models X and Y: (a) 40ms and 60ms, (b) 50ms and 50ms, (c) 60ms and 40ms. It would appear that plan (a) should work best since the sum of the throughputs is largest among the three plans, but a closer examination reveals some interesting details.

For workloads involving a large number of requests, let us assume that p and q GPUs execute X and Y, respectively. We then have $\gamma \cdot p \cdot T_X = q \cdot T_Y$, where T_X and T_Y are per-GPU throughputs of X and Y, such that the pipeline won't be bottlenecked by any model. We define the average throughput as the pipeline throughput divided by the total number of GPUs, which is $p \cdot T_X/(p+q)$. We evaluate the average throughputs for the three latency split plans with $\gamma = 0.1, 1, 10$. Figure 4 shows that each of the plans achieves the best performance for different γ values. In fact, there is no universal best split: it depends on γ, which can vary over time.

We note two observations from this example. First latency split for complex query impacts overall efficiency, and it is necessary to account both batch performance and workload statistics to make the best decision. Second, latency split should not be static but rather adapted over time in accordance with the current workload. Section 6.2 describes how Nexus automatically and continually derives latency splits.

4.3 Rate control and adaptive batching

Model serving systems need to perform adaptive batching based on number of requests received. When there is a burst of requests, the system needs to drop certain requests in order to serve the remaining requests within the latency SLO. One approach is to perform lazy dropping, i.e., drop a request only when it has already missed its deadline, and determine the batch size based on the time budget remaining for the earliest request in the queue (as in Clipper [8]). We use simulation to evaluate this approach for different batching profiles (as modeled by Equation 1). We fix latency SLO to 100ms and optimal model throughput on a single GPU to 500 reqs/s,

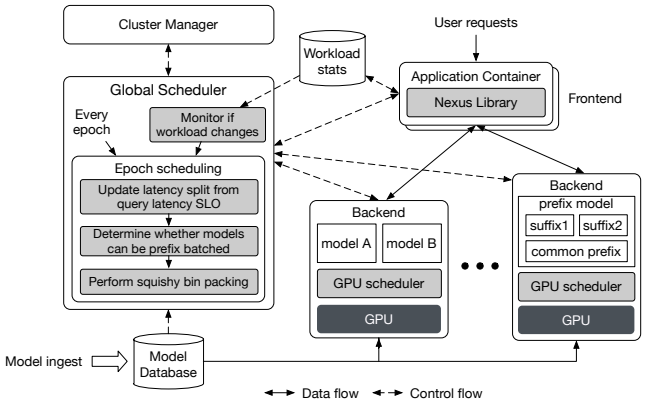


Figure 6: Nexus runtime system overview.

and vary α. Given the fixed throughput, the fixed cost β reduces as we increase α. The workload is generated using uniform and Poisson arrivals with the mean request rate set to 90% of the optimal throughput. We define the *bad rate* to be the percentage of requests that exceed the deadline or get dropped. Figure 5 shows that the lazy dropping strategy performs poorly for Poisson distributions when α is small and β is correspondingly high. Since the system always attempts to execute the oldest received request, it often has to resort to a small batch size in order to meet the deadline, but this causes the dispatcher to further fall behind given the high fixed cost is not amortized over sufficient requests. This experiment indicates that even the runtime needs to consider batch efficiency in determining what tasks to dispatch.

5 Nexus Architecture

Nexus works on three planes (as depicted by Figure 6). The *management* plane allows developers to ingest and deploy applications and models, at a timescale of hours to weeks. The *control* plane, via the *global scheduler*, is responsible for resource allocation and scheduling at a typical timescale of seconds to minutes. The *data* plane, comprised of in-application *Nexus library* instances and backend modules (together, the *Nexus runtime*), dispatches and executes user requests at the timescale of milliseconds to seconds. The global scheduler interacts with the underlying cluster resource manager (e.g., Mesos [17], Azure Scale Sets [23]) to acquire CPUs/GPUs for the frontend/backend. A load balancer (not shown) from the underlying cluster spreads user requests across Nexus's distributed frontend. We sketch the three planes.

Management plane: Models are stored in a *model database* and may be accompanied by either a sample data set or a batching profile. Nexus uses the sample dataset, if available, to derive a batching profile. Otherwise, the profile is updated at runtime based on user requests. Applications are containers that integrate with Nexus Client Library. Developers store application containers in cluster-provided container repositories and may instruct Nexus to ingest a container, at which point it is loaded from the repository onto a frontend CPU.

Control plane: The global scheduler is a cluster-wide resource manager that uses load statistics from the runtime. It uses this profile to add or remove frontend and backend nodes from the cluster, invokes the *epoch scheduler* to decide which models to execute and at what batch size, and which backend to place the models on so as to balance load and maximize utilization. Multiple models may be mapped onto a single backend, albeit with an execution schedule that ensures they do not interfere as in Section 4.1. The mapping from models to backends is captured in a *routing table* that is sent to frontends. The matching execution schedule for each backend is captured in a *schedule* that is sent to backends. On receiving a routing table, frontends update their current routing table. On receiving a schedule, backends load appropriate models into GPU memory and set their execution schedule.

Allocation, scheduling and routing updates happen at the granularity of an *epoch*, typically 30-60s, although a new epoch can also be triggered by large changes in workload. Epoch scheduling involves the following:

- Produce an updated split of the latency SLO for the individual models inside a query (see Section 6.2).
- Combine two or more models that share a prefix and latency SLO into a new *prefix-batched* model (see Section 6.3).
- Perform profile-guided squishy bin packing to allocate the GPU resources for each model. (see Section 6.1).

Data plane: When a user request comes into (a replica of) an application container, the application invokes DNNs via API provided by the Nexus Library. The library consults the local routing table to find a suitable backend for that model, dispatches the request to the backend, and delivers responses back to the application. The application is responsible for packaging and delivering the end-result of the query to the user. A backend module uses multiple threads to queue requests from various frontends, selects and executes models on these inputs in batched mode according to the current schedule, and sends back the results to frontends. It can utilize one or more GPUs on a given node, with each GPU managed by a *GPU scheduler* that schedules tasks on it.

6 Batch-Aware Scheduling and Dispatch

We now describe the algorithms used by the global scheduler and the node dispatcher. First, we consider the case of scheduling streams of individual DNN task requests, given their expected arrival rates and latency SLOs. We next consider how to schedule streams of more complex queries/jobs that invoke multiple DNN tasks. We then describe how the node runtime cycles through DNNs and performs batching.

Notation	Description
S_i	Session i
M_{k_i}	DNN model k_i for session S_i
L_i	Latency constraint for session S_i
R_i	Request rate for session S_i
$\ell_{k_i}(b)$	Execution cost for M_{k_i} and batch size b

Table 3: Notation

6.1 Scheduling streams of individual DNN tasks

We build upon the discussion presented in Section 4.1. The scheduler identifies for each cluster node the models hosted by it. As discussed earlier, the scheduling problem has the structure of bin-packing [3], but we need to address the "squishiness" of tasks and the need to meet latency SLOs.

Inputs: The scheduler is provided the request rate for a model at a given latency SLO. We refer to the requests for a given model and latency SLO as a *session*. Note that a *session* would correspond to classification requests from different users and possibly different applications that invoke the model with a given latency constraint. Table 3 describes the notation used below. Formally, a session S_i specifies a model M_{k_i} and a latency constraint L_i, and there is a request rate R_i associated with it. The scheduler is also provided with the batching profiles of different models. The latency of executing b invocations of M_{k_i} is $\ell_{k_i}(b)$. We assume that throughput is non-decreasing with batch size b.

Scheduling overview: The scheduler allocates one or more sessions to each GPU and specifies their target batch sizes. Each GPU node n is then expected to cycle through the sessions allocated to it, execute invocations of each model in batched mode, and complete one entire cycle of batched executions within a *duty cycle* of d_n. For sessions that have a sufficient number of user requests, one or more GPU nodes are allocated to a single session. The integer programming formulation and a greedy approximation algorithm described below computes the residual workload for such sessions (after allocating an integral number of GPUs) and then attempts to perform bin packing with the remaining smaller sessions.

Scheduling Large Sessions: For session S_i, we first compute the peak throughput of M_{k_i} when executed in isolation on a GPU. With a batch size b, the worst case latency for any given request is $2\ell_{k_i}(b)$, as we explained in Section 4.1. Denote batch size B_i as the maximum value for b that meets the constraint $2\ell_{k_i}(b) \leq L_i$. Therefore, the maximal throughput, denoted by T_i, for session S_i on a single GPU is $B_i/\ell_{k_i}(B_i)$. The number of GPU nodes we allocate to execute just S_i requests is $n = \lfloor R_i/T_i \rfloor$. There will be a residual unallocated load for session S_i after taking into account this allocated load. Note that $n = 0$ for sessions that don't have sufficient requests to utilize an entire GPU. (Function SCHEDULESATURATE in Algorithm 1 provides the pseudocode.)

Optimization problem for scheduling residual workload: We next consider the problem of scheduling the residual loads, i.e., a workload where none of the models have sufficient load to need an entire GPU. The optimization problem can be expressed as an integer programming problem.

Decision Variables	Definition
$g_j \in \{0, 1\}$	Whether GPU j is in use
$s_{ij} \in \{0, 1\}$	Whether session i is assigned to GPU j
$b_{ij} \in \mathbb{R}_{\geq 0}$	Batch size of session i on GPU j

Minimize: $\sum_j g_j$
Subject to:

$$s_{ij} = 1 \rightarrow g_j = 1 \qquad \forall j \qquad \text{(a)}$$

$$\sum_j s_{ij} = 1 \qquad \forall i \qquad \text{(b)}$$

$$s_{ij} = 0 \rightarrow b_{ij} = 0 \qquad \forall i, j \qquad \text{(c)}$$

$$s_{ij} = 1 \rightarrow b_{ij} \geq 1 \qquad \forall i, j \qquad \text{(d)}$$

$$d_j = \sum_{i:t_{ij}=1} \ell_{k_i}(b_{ij}) \qquad \forall j \qquad \text{(e)}$$

$$d_j + \ell_{k_i}(b_{ij}) \leq L_i \qquad \forall i, j\ (s_{ij} = 1) \qquad \text{(f)}$$

$$b_{ij} \geq r_i d_j \qquad \forall i, j\ (s_{ij} = 1) \qquad \text{(g)}$$

The constraints correspond to the following requirements.

(a) Sessions can only be assigned to GPU that are in use.
(b) Each session can only be assigned to one GPU.
(c) b_{ij} is 0 if i is not assigned to GPU j.
(d) b_{ij} is at least 1 if i is assigned to GPU j.
(e) Length of duty cycle as a function of assigned sessions.
(f) Latency SLO constraint.
(g) Scheduled rate meets the request rate requirement.

Note that some of the constraints are not linear, and we omit details on how to express them in a strictly linear way. We used the CPLEX package to solve this formulation on benchmark workloads. Even after incorporating optimizations, such as using a greedy algorithm to provide both an initial feasible solution and an upper bound for the number of GPUs needed, solving the integer program is expensive. For example, generating a schedule for 25 sessions takes hours. We therefore resort to the following greedy scheduling algorithm.

Greedy scheduling algorithm for residual loads: For the bin packing process, the scheduler inspects each residual session in isolation and computes the largest batch size and the corresponding duty cycle in order to meet the throughput and SLO needs. The intuition behind choosing the largest batch size is to have an initial schedule wherein the GPU operates at the highest efficiency. This initial schedule, however, is not cost-effective as it assumes that each GPU is running just one session within its duty cycle, so the algorithm then attempts to merge multiple sessions within a GPU's duty

Algorithm 1 Squishy Bin Packing Algorithm

```
    SquishyBinPacking(Sessions)
1:  nodes, residue_loads ← ScheduleSaturate(Sessions)
2:  nodes ← nodes ⊕ ScheduleResidule(residue_loads)
3:  return nodes

    ScheduleSaturate(Sessions)
4:  nodes, residue_loads ← [], []
5:  for S_i = ⟨M_{k_i}, L_i, R_i⟩ in Sessions do
6:      B_i ← argmax_b(2ℓ_{k_i}(b) ≤ L_i)
7:      T_i ← B_i/ℓ_{k_i}(B_i)
8:      let R_i = n · T_i + r_i
9:      nodes ← nodes ⊕ n GPU nodes for M_{k_i} with batch B_i
10:     residue_loads ← residue_loads ⊕⟨M_{k_i}, L_i, r_i⟩
11: return nodes, residue_loads

    ScheduleResidule(residue_loads)
12: for ⟨M_{k_i}, L_i, r_i⟩ in residue_loads do
13:     b_i ← argmax_b(ℓ_{k_i}(b) + b/r_i ≤ L_i)
14:     d_i ← b_i/r_i
15:     occ_i ← ℓ_{k_i}(b_i)/d_i
16: sort residue_loads by occ_i in descending order
17: nodes ← []
18: for ⟨M_{k_i}, L_i, r_i, b_i, d_i, occ_i⟩ in residue_loads do
19:     max_occ ← 0
20:     max_node ← NULL
21:     for n = ⟨b, d, occ⟩ in nodes do
22:         n′ ← MergeNodes(n, ⟨b_i, d_i, occ_i⟩)
23:         if n′ ≠ NULL and n′.occ > max_occ then
24:             max_occ ← n′.occ
25:             max_node ← n′
26:     if max_node ≠ NULL then
27:         replace max_node for its original node in nodes
28:     else
29:         nodes ← nodes ⊕⟨b_i, d_i, occ_i⟩
30: return nodes
```

cycle. In doing so, it should not violate the latency SLOs, so we require the merging process to only reduce the duty cycle of the combined allocation. The algorithm considers sessions in decreasing order of associated work and merges them into existing duty cycles that have the highest allocations, thus following the design principle behind the best-fit decreasing technique for traditional bin packing.

We now elaborate on this greedy scheduling algorithm (which is also depicted in function ScheduleResidue of Algorithm 1). Denote r_i to be the request rate of a residual load. Suppose we execute the residual load with batch size b, the duty cycle d for gathering b inputs is b/r_i. Then, the worst case latency is $d + \ell_{k_i}(b)$. Therefore, we have the constraint:

$$d + \ell_{k_i}(b) = b/r_i + \ell_{k_i}(b) \leq L_i \tag{2}$$

We begin residual load scheduling (Line 12-15) by choosing for session S_i the maximum batch size b_i that satisfies the above constraint. The corresponding duty cycle d_i is also at its maximal value. Denote *occupancy* (*occ*) as the fraction of the duty cycle d_i occupied by S_i's residual load invocations: $occ_i(b_i) = \ell_{k_i}(b_i)/d_i$.

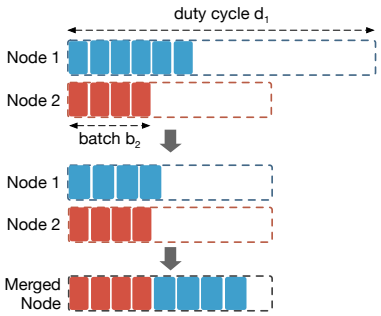


Figure 7: Merge two nodes into one. Use the smaller duty cycle as new duty cycle for both nodes. Update the batch size accordingly and re-estimate the batch latency. If sum of latencies doesn't exceed new duty cycle, the two nodes can be merged.

Next, we start to merge these fractional GPU nodes into fewer nodes (Line 16-30 in Algorithm 1). This part resembles the classic bin packing algorithm that first sorts sessions by decreasing occupancy and merges two nodes into a single node by best fit. The primary difference is how to determine whether two nodes can be merged such that their sessions won't violate the latency SLOs. Figure 7 depicts the process of merging two nodes. Suppose we have two sessions S_1 and S_2 on separate nodes, with request rates r_1 and r_2, assigned batch sizes b_1 and b_2, and duty cycles d_1 and d_2. We use $d = \min(d_1, d_2)$ as the new duty cycle of a combined node. Without loss of generality, we assume $d = d_2$. We then use $b_1' = d \cdot r_1 \leq b_1$ as the new batch size for S_1. Note that the worst case latency of requests in S_1 now becomes $d + \ell_{k_1}(b_1') \leq d_1 + \ell_{k_1}(b_1) \leq L_i$, and we won't violate the latency constraint for S_1 by this adjustment. If $\ell_{k_1}(b_1') + \ell_{k_2}(b_2) \leq d$ and memory capacity permits, a single node can handle the computation of both S_1 and S_2, and we allocate these two sessions to the same node. While the above discussion considers merging two sessions, the underlying principle generalizes to nodes containing multiple sessions.

Finally, we extend the algorithm to be incremental across epochs, thus minimizing the movement of models across nodes. If the overall workload decreases, the scheduler attempts to move sessions from the least utilized backends to other backends. If a backend no longer executes any session, the scheduler releases the backend. If workload increases such that a backend becomes overloaded, we evict the cheapest sessions on this backend until it is no longer overloaded and perform bin packing again to relocate these evicted sessions.

6.2 Scheduling Complex Queries

We now present the query analysis algorithm that operates on dataflow representations of application queries in order to determine the latency SLO splits for the constituent models. The output of this analysis is given as input to the scheduling algorithm of Section 6.1 that works with individual models.

Query analysis extracts the dataflow dependency graph between model invocations in application code. For example, Figure 8 depicts a traffic analysis application that first uses the SSD model to detect objects and recognizes cars and faces correspondingly. We formulate the scheduling of queries as the following optimization problem. Suppose the query involves a set of models M_i with request rate R_i, and the end-to-end latency SLO is L. The objective is to find the best latency SLO split L_i for each model M_i to minimize the total number of GPUs that are required for the query. Because latency L_i is determined by batch size b_i, the optimization problem is equivalent to finding the best batch sizes that minimizes GPU count, while meeting the latency SLO along every path from the root model (M_{root}) to the leaf models.

$$\begin{aligned} &\underset{\{b_v\}}{\text{minimize}} && \sum_v R_v l_v(b_v)/b_v \\ &\text{subject to} && \sum_{u:M_{\text{root}} \rightsquigarrow M_v} l_u(b_u) \le L \qquad \forall v \in \text{leaf} \end{aligned}$$

We use dynamic programming to solve this optimization problem for the case of fork-join dependency graphs, but limit our exposition to the simpler case of tree-like dependency graphs. For example, Figure 8 can be treated as a tree-structured dependency graph models (we can as the output does not invoke additional DNN models. Denote $f(u,t)$ as the minimum number of GPUs required to run models represented by u and the subtree at u within time budget t. For a non-leaf node u, the algorithm allocates a time budget k for node u and at most $t-k$ for the rest of the subtree, and it then enumerates all $k \le t$ to find the optimal split. More formally,

$$f(u,t) = \min_{k:k\le t} \left\{ \min_{b:l_u(b)\le k} R_u \frac{l_u(b)}{b} + \min_{t':t'\le t-k} \sum_{v:M_u \to M_v} f(v,t') \right\}$$

Since the dynamic programming cannot handle continuous state space, we approximate the state space of time budget with L/ε pieces of segments, where ε is the length of a segment. The time complexity is quadratic in L/ε.

6.3 Batch-Aware Dispatch

We now briefly describe the runtime mechanisms that control the execution of DNN tasks on backend nodes.

GPU Multiplexing: DNN frameworks provide no specific support for the concurrent execution of multiple models. For example, if two models that share a GPU execute in two processes or containers, they will independently issue requests to execute layers/kernels to the underlying GPU. The GPU runtime will typically serve these requests in first-come-first-served fashion, resulting in an arbitrary interleaving of the operations for the two models. The interleaving increases the execution latency of both models and makes it hard to predict the latency. Instead, the Nexus node runtime manages the execution of all models on a GPU, so it is able to pick batch sizes and execution schedules for all models in a round-robin fashion to make sure models abide by their latency SLOs. In addition, Nexus overlaps the pre- and post-processing in CPU with the GPU execution to increase GPU utilization.

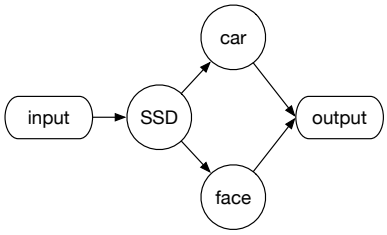


Figure 8: Dataflow graph of traffic analysis application.

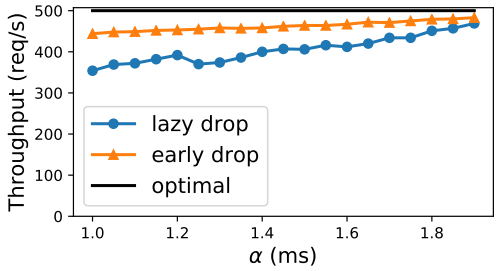


Figure 9: Maximal throughput achieved by lazy drop and early drop policy under various α.

Prefix Batching: Another important observation is that transfer learning [10, 33] adapts a model from one dataset to another or from one task to another by re-training only the last few layers. DNN frameworks assume that if models differ in any layer, they cannot be executed in a batched fashion at all. However, in the common setting of model specialization, several models may differ only by their output layer. Batching the execution of all but the output layer can yield substantial batching gains. Nexus automatically recognizes models that have common prefixes, and splits the models into "common prefix" and "different suffixes" parts. The backend executes the common parts in a batched manner followed by the different suffix parts sequentially to complete execution.

Adaptive Batching: As discussed in Section 4.3, lazy dropping during dispatch could lead to small batch sizes and low efficiency. In Nexus, we use an early drop policy that skips over requests that would cause sub-optimal batching. Specifically, the dispatcher scans through the queue using a sliding window whose length is the batch size determined by the

name	brief description	models used	video input	Nexus features used
game	analyze streamed video games	text, object rec.	Twitch [2] streams, 1 week, 50 streamers	SS, ED, QA-1, PB
traffic	surveil traffic on streets	object det., face rec.	traffic cameras, 1 week, 20 cameras	SS, ED, QA-2
dance	rate dance performances	person det., pose rec.	dance videos from YouTube, 2 hrs	SS, ED, QA-2
bb	gauge response to public billboard	person, face det., gaze, age, sex rec.	game show audience videos, 12 hours	SS, ED, QA-3, PB
bike	find bike-rack occupancy on buses	object, text det./ rec./ trk.	traffic cameras, 1 week, 10 cameras	SS, ED, QA-4, PB
amber	match vehicle to "Amber Alert" description	object det., car make+model rec., text det./rec.	dashcam videos from YouTube, 12 hours	SS, ED, QA-4, PB
logo	audit corporate logo placeement	person icon, pose, text, person det./rec.	NFL, NBA game videos, 24 hours	SS, ED, QA-5, PB

Table 4: Evaluated application and input data. Squishy scheduling, early drop, complex query analysis and prefix-batching are abbreviated as SS, ED, QA and PB. QA-k indicates that the related complex query has k stages. Models for detection, recognition and tracking are abbreviated 'det.', 'ret.' and 'trk.'

global scheduler for a given session. It stops at the first request that has enough budget for batched execution latency of the entire window and drops all earlier requests. We use simulation to compare the lazy drop and early drop policy. Similar to Figure 5, we fix latency SLO to 100ms and optimal throughput to 500 reqs/s. Figure 9 depicts the throughput achieved by lazy drop and early drop policy under different α while 99% of requests are served within latency SLO. The results show that early drop can achieve up to 25% higher throughput than lazy drop.

7 Evaluation

We implemented Nexus in roughly 10k lines of C++ code. Nexus supports the execution of models trained by various frameworks including Caffe [19], Caffe2 [11], Tensorflow (TF) [4], and Darknet [28]. Nexus can be deployed in a cluster using Docker Swarm [9] (used below) or Kubernetes [13]. In our evaluation, we use this implementation to answer the following questions. (1) Does using Nexus result in better cluster utilization while meeting SLOs with respect to existing systems? (2) Does high performance persist when Nexus is used at very large scale? (3) How do the new techniques in Nexus contribute to its performance? (4) What determines how well each of these techniques work?

For a given workload and cluster, we refer to the maximum rate of queries that Nexus can process such that 99% of them are served within their latency SLOs as its *throughput*. We use throughput as the primary measure of cluster utilization.

7.1 Workload

Our basic approach is to run Nexus (and its various configurations and competitors) on either a small (16-GPU) or a large (100-GPU) cluster on various mixes of the applications and input videos specified in Table 4. These applications are modeled closely on widely-known video analysis scenarios, but we implemented each of them since we are unaware of freely available, widely used versions. They encompass a rich variety of characteristics. Some (e.g., `game` and `traffic`, which implements Figure 8) are based on 24/7 live video streams, whereas others (e.g., `dance` and `logo`) apply to footage of individual performances. Some require simple queries (e.g., `game`, designated "QA-1" has 1 stage), and others more complex ones (e.g., the 5-stage `logo`, designated "QA-5", seeks to detect people, find their torsos, look for logos, and if found, detect and recognize the player's number). Most use multiple specialized versions of models and are therefore amenable to prefix batching, designated "PA". For each workload, we have collected several hours and many streams (for live streams) or files (for episodes) of video, which sample and play in a loop to preserve temporal characteristics while allowing arbitrarily long simulations.

7.2 Using Clipper and Tensorflow as Baselines

Clipper and TF Serving assume cluster scheduling and latency SLOs for DNN invocations are handled externally. Careful scheduling and latency allocation are two of Nexus's core contributions. To provide a basis for comparison, we furnish simple default versions of each. A *batch-oblivious* scheduler greedily allocates to each model/SLO a share of the cluster proportional to its request rate and inversely proportional to its maximum single-node throughput. Further, we split the latency for a query evenly across its stages. The oblivious scheduler may map multiple models onto a Clipper GPU, in which case we launch one container per model on the GPU. We rely on Clipper's load balancer to manage model replicas. In contrast, TF does not provide a frontend load balancer, nor does it allow specification of latency SLOs per request. We therefore provide a dispatcher and pick the maximum batch size for each model so its SLO is not violated.

7.3 Single-Application Case Studies

To compare our performance with those of Clipper and TF Serving, we ran the `game` and `traffic` applications separately on a 16-GPU cluster. In each case, we ran an ablation study on Nexus features to gauge their impact.

7.3.1 Game Analysis When analyzing game streams, we seek to recognize 6 numbers (e.g., number of kills, number of players alive) and one object icon on each frame. We use versions of LeNet [21] specialized to the game's font and number of digits to recognize numbers, and ResNet-50 [16] with its last layer specialized to recognize the icon. We include 20 games in the case study, and consider a latency SLO of 50ms (sensitivity to SLOs is analyzed in Section 7.5). The request rates of frames from the 20 games follow the Zipf-0.9 distribution. We noticed that both Clipper and TF show extremely

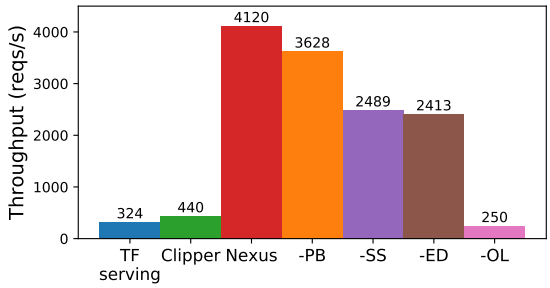


Figure 10: Game analysis case study.

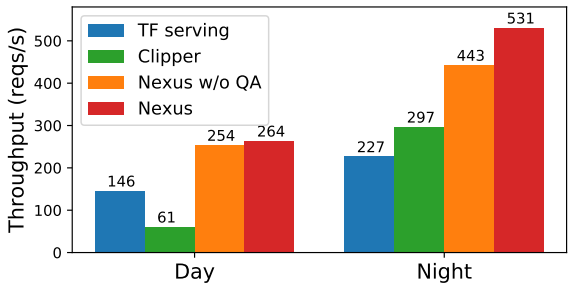


Figure 11: Traffic monitoring case study.

poor throughput on the tiny LeNet model. We conjecture, but could not confirm, this is because of inadequate parallelism between CPU and GPU processing. To be maximally fair to them, we allow the two baselines to invoke just the ResNet model. Their resulting throughput, which we report, is better by over 4× than including LeNet. Finally, we additively turn off prefix batching (PB), squishy scheduling (SS), early drop (ED), and overlapped processing in the CPU and GPU (OL, see Section 6.3). The `game` query has only 1 stage and therefore does not exercise query analysis (QA).

Figure 10 shows the results. **Nexus increases throughput significantly**, 9.4 and 12.7× relative to Clipper and TF Serving on this application. **Several of Nexus's techniques contribute**, with OL the most, and ED the least.

7.3.2 Traffic Monitoring `traffic` uses SSD [7], VGG-Face [25] and GoogleNet-car [32] for object detection, face recognition and car make/model analysis on 20 long-running traffic streams with a latency SLO of 400ms. Figure 11 shows the results, focusing on additional aspects. First, **variation in the content of the video affects throughput**: "Day" hours tend to have more objects to process in each frame, and therefore yield lower throughput than "Night" hours. Secondly, since `traffic` is a two-stage application (detect cars and then recognize their make/model), QA is relevant. Instead of splitting latency evenly, it allocates 304 of 400ms of SLO to SSD during the day, and further adapts to 345ms in the evening. **QA has impact**, but more so at night, perhaps because during the day the system is often over-subscribed.

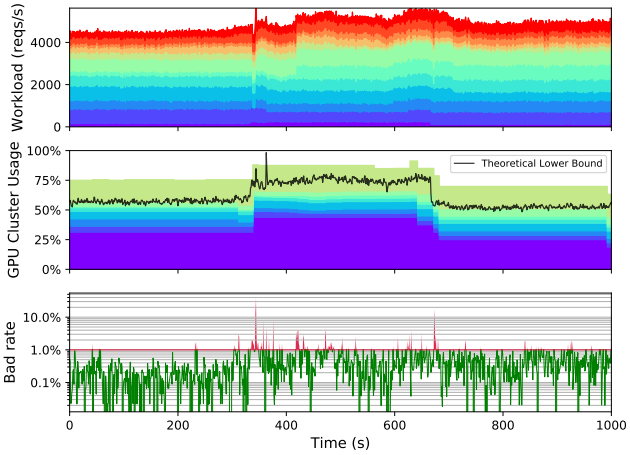


Figure 12: A 1000 sec window from our large-scale deployment.

Disabling QA reduced throughput by 4/17% in the day/night. **Nexus still exhibits better throughput** relative to Clipper/TF, by 4.4/1.8× in the daytime and 1.8/2.3× at night.

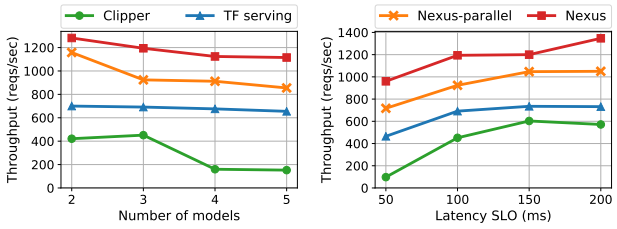
7.4 Large-scale deployment

To check whether Nexus maintains its gains when run at large scale, especially in the face of significant workload variation, we deployed Nexus on a cluster of 88 GPUs on a commercial cloud, running all applications from Table 4 simultaneously for a period of several hours. During this period, we fixed the number of sessions per application (e.g., `game` had 50 sessions, `traffic` had 20), but varied the request rate per session by varying the rate at which each submitted frames.

We focused on two metrics. First, how close to optimal is GPU utilization during this period? Second, how often does Nexus fail to honor SLOs i.e. what is its "bad rate"? To bound the optimal (smallest) number of GPUs needed for a session, we assumed that its model is fully (not just prefix) batchable, that its SLO allows it to run at optimal batch size, and that it has enough requests coming in to be scheduled back-to-back on GPUs. Of course, real sessions often violate one or more of these assumptions and will have lower throughput.

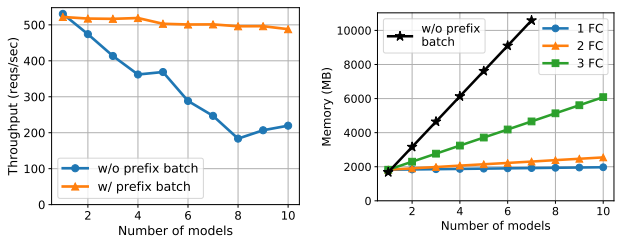
Figure 12 shows Nexus adapting to a change in workload during a 1000-sec window of the deployment. The top panel shows a stacked graph of requests over time, the middle one the number of GPUs allocated (with a black line indicating the lower bound number of GPUs needed), and the bottom one the bad rate, with instantaneous bad rates above 1% marked in red. Around 340s into the window, the number of requests increases and starts varying significantly. Nexus, which is running with 30s epochs, starts dropping requests, detects the change within 12s (this could have been as long as 30s) and allocates more GPUs. It deallocates GPUs (this time with a roughly 10s lag) at the 680s mark when demand subsides.

The figure illustrates that **Nexus responds well to variable workloads at large scale**. It is able to allocate close to the



(a) (b)

Figure 13: Impact on throughput of varying numbers of models served (a) and latency SLOs (b) under GPU multiplexing.



(a) (b)

Figure 14: Impact on throughput (a) and memory use (b) of varying numbers of batched models under prefix batching.

aggressive theoretical lower bound (on average it stays within 32%), while maintaining a low bad rate (1.3% on average).

7.5 Sensitivity analysis of Nexus features

We now present micro-benchmarks to analyze the main components of Nexus. Overall, we find that Nexus's core techniques are quite robust to variations in key design parameters.

GPU Multiplexing. The Nexus runtime (Section 6.3) focuses on minimizing interference on GPU between executing models (by avoiding interleaving during their execution), and idling while switching between models (by overlapping pre/post processing on CPU with model execution on the GPU, and not waiting for fixed target batch sizes to fill up before dispatch to the GPU).

Figure 13 analyzes the importance of these features by comparing the throughput of Nexus with those of Clipper, TF Serving, and a version of Nexus ("Nexus-parallel") that issues models in parallel and does not control interference. This experiment runs increasing numbers of copies of the Inception model with a latency SLO of 100ms. Throughput of all four models suffer, TF Serving less than Clipper because it runs models in a round-robin fashion whereas Clipper deploys them in independent containers that interfere. Nexus achieves 1.4–2.1× throughput compared to TF serving, and 1.9–9.8× throughput compared to Clipper on a single GPU. Nexus-parallel fares better because it avoids idling (but still suffers from interference), and Nexus fares the best. We see similar trends across other models. Figure 13(b) compares the throughput while varying the latency SLO from 50ms to 200ms, with the number of models fixed at 3. When latency SLO becomes higher, the greater scheduling slack gives Nexus-parallel higher throughput.

Prefix Batching. Figure 14 examines how the throughput and memory benefits of prefix batching scale as the number of variants of Resnet50 that differ only in the last layer increases, on a single GPU. Figure 14(a), compares prefix batching to unbatched execution of the variants. Without prefix batching, the variants have to execute on smaller "sub-batches" to satisfy their SLOs, yielding worse aggregate throughput. With prefix batching, since many models can execute in one batch, the sub-batches can be aggregated into a large batches that maintain up to 110% higher throughput.

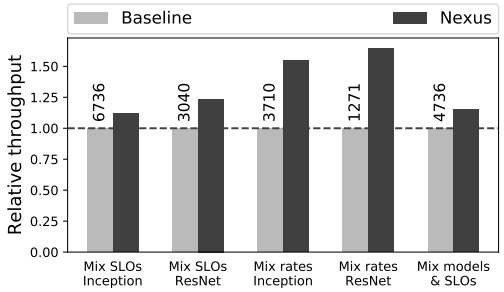


Figure 15: Impact on throughput of varying model- and SLO-mixes under squishy scheduling.

Similarly, when the (unshared) model suffixes are small ("1 FC", indicating one "fully connected" unshared layer, in Figure 14(b)), additional model variants use negligible extra GPU memory. As the number of unshared layers increase ("2 FC" and "3 FC" add 2 and 3 fully connected layers to the shared prefix), the memory benefits fall. Without prefix batching (black line), however, we quickly run out of GPU memory even if a model has only one unshared layer.

Squishy Scheduling. We now examine the sensitivity of squishy scheduling to model types, request rates and SLOs. We compare the throughput of Nexus with squishy scheduling to a baseline using batch-oblivious scheduling instead. Both need to allocate 16 sessions on 8 GPUs under 5 scenarios: (a) Inception or (b) ResNet models with mixed SLOs ranging from 50ms to 200ms, (c) Inception or (d) ResNet models with mixed request rates following Zipf-0.9 distribution, (e) 8 different model architectures, each associated with two SLOs, 50ms and 100ms. Figure 15 depicts the relative throughput of standard Nexus with regard to baseline. Nexus outperforms baseline across all mixes, with the highest gains (up to 64%) coming from handling varying request rates, and the lowest (11%) coming from handling varying request mixes.

Complex Query Analysis. To evaluate the performance gain of the query analyzer, we compare the throughput of Nexus with and without the query analyzer. The baseline simply

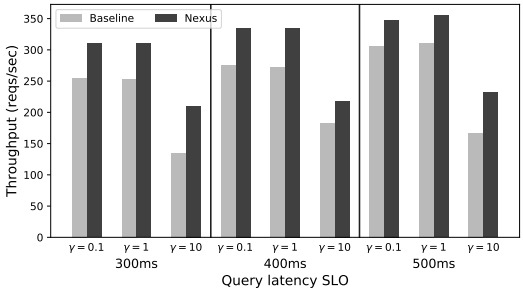


Figure 16: Impact on throughput of varying query latency SLO and γ (see Section 4.2) under complex query analysis.

splits the latency SLO evenly across the various stages in the query. The query includes two stages: (a) first stage executes SSD, and then (b) invokes Inception model for γ times. The experiment is performed on 8 GPUs. We vary the latency SLO from 300ms to 500ms, and choose γ to be 0.1, 1, and 10. Figure 16 shows that Nexus with the query analyzer achieves 13–55% higher throughput than the baseline.

8 Conclusion

We proposed a scalable and efficient system design for serving Deep Neural Network (DNN) applications. Instead of serving the entire application in an opaque CPU-based container with models embedded in it, which leads to sub-optimal GPU utilization, our system operates directly on models and GPUs. This design enables several optimizations in batching and allows more efficient resource allocation. Our system is fully implemented, in C++ and evaluation shows that Nexus can achieve 1.8-12.7× more throughput relative to state-of-the-art baselines while staying within latency constraints (achieving a "good rate") >99% of the time.

References

[1] Serving a tensorflow model. https://www.tensorflow.org/tfx/serving/serving_basic.

[2] Twitch. https://www.twitch.tv/.

[3] *Bin-Packing*, pages 449–465. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[5] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. Fast cholesky factorization on gpus for batch and native modes in magma. *Journal of Computational Science*, 20:85–93, 2017.

[6] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, et al. Slicer: Auto-sharding for datacenter applications. In *OSDI*, pages 739–753, 2016.

[7] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):834–848, 2018.

[8] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627, 2017.

[9] Docker. Docker swarm. https://github.com/docker/swarm.

[10] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *International conference on machine learning*, pages 647–655, 2014.

[11] Facebook. Caffe2. https://caffe2.ai/.

[12] Google. Cloud automl vision. https://cloud.google.com/vision/automl/docs/.

[13] Google. Kubernetes: Production-grade container orchestration. https://kubernetes.io/.

[14] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 485–500, 2019.

[15] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 123–136. ACM, 2016.

[16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[17] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[18] K. Hsieh, G. Ananthanarayanan, P. Bodik, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying large video datasets with low latency and low cost. *arXiv preprint arXiv:1801.03493*, 2018.

[19] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[20] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[21] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[22] Microsoft. Custom vision. https://azure.microsoft.com/en-us/services/cognitive-services/custom-vision-service/.

[23] Microsoft. Virtual machine scale sets. https://docs.microsoft.com/en-us/azure/virtual-machine-scale-sets/virtual-machine-scale-sets-overview.

[24] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[25] O. M. Parkhi, A. Vedaldi, A. Zisserman, et al. Deep face recognition. In *BMVC*, volume 1, page 6, 2015.

[26] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, page 3. ACM, 2018.

[27] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[28] J. Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[29] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes.

Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[30] H. Shen, S. Han, M. Philipose, and A. Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[31] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, 2018. USENIX Association.

[32] L. Yang, P. Luo, C. Change Loy, and X. Tang. A large-scale car dataset for fine-grained categorization and verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3973–3981, 2015.

[33] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[34] D. Yu, F. Seide, G. Li, and L. Deng. Exploiting sparseness in deep neural networks for large vocabulary speech recognition. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2012.