Piccolo: A Cost-Efficient Deep Neural Network Inference System with Latency Constraints

***Abstract*—Advances in deep neural network (DNN) have contributed to the development of DNN-based applications to process video streams in real-time. The inference system hosted in the cloud is required to schedule the DNN workload efficiently to minimize the total cost, while satisfying the latency constraints of the applications. Piccolo, an inference system that provides DNN-based applications as a service, achieves the cost minimum objective by optimizing the multi-dimensional scheduling problem to achieve high throughput, splitting the end-to-end latency constraint of multi-DNN applications across modules to utilize the latency efficiently, and adding a proper amount of dummy requests to the workload to increase the cost efficiency for machines with small workload. Evaluation shows that Piccolo can save up to 2 times the total cost than the state of the art while achieving the latency objectives. Compared to the optimal solution using brute force searching, Piccolo achieves the same total cost for over 97% workload while being 1000 times faster.**

I. Introduction

Deep neural network (DNN) has achieved the state of the art results in multiple fields [1], [2], such as computer vision, speech recognition and natural language processing. Therefore, the application developers are staring to build streaming applications using DNN models [3]–[10]. These DNN-based applications usually have latency constraints to guarantee good user experiences. For example, the traffic monitoring application [11], which extracts pedestrian and vehicle information from videos collected by the surveillance cameras, often expects results to be returned within a few milliseconds to provide timely supports when emergency occurs.

To satisfy the latency objective, the DNN-based applications are usually deployed in the inference system to leverage the plentiful GPU resources in the cloud. The inference system uses the classic *client-server* architecture, where the client sends requests to the inference system, which will query corresponding DNN models and return the results back to the client. While providing services to the clients, the application developers are charged by the cloud provider according to the amount of computing resources used by the DNN-based applications. Inference for DNN-based application accounts for nearly 90% of computing costs in the cloud [12]. Therefore, when running the inference system, one primary goal for the application developers is to utilize the expensive computing resources efficiently, so as to minimize the total cost while satisfying the latency objective.

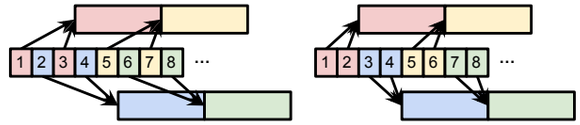
This paper considers the design and implementation of a system, called Piccolo, which provides cost-efficient DNN inference services hosted in the cloud. Piccolo is a cloud-based inference system, which applies client inputs (*e.g.,* images or video streams) to DNN models. Piccolo schedules incoming workload efficiently to achieve high resource utilization and tracks the dynamic workload in real-time for resource auto-scaling. When achieving the cost minimum goal, Piccolo makes the following contributions.

The first contribution of Piccolo is a novel resource scheduling method to maximize the system throughput under the latency constraint. Existing systems [3], [5], [6] dispatch requests among machines in individual request and only support a fixed amount of configurations for each module, which unnecessarily increases the latency and reduces the resource efficiency. Piccolo dispatches requests among machines in batched request to minimize the worst case latency, which makes module configuration with higher throughput feasible. Besides, Piccolo uses a novel combination of batching, spatial multiplexing and heterogeneous hardware to derive the optimal scheduling with multiple configurations per module, which increases the resource efficiency.

The second contribution of Piccolo is a novel latency splitting method to minimize the total cost for multi-DNN applications. Given the end-to-end latency constraint for the entire application, existing solutions either use quantized interval [3] or local runtime information [4], [5] to split the latency into per-module latency budget, leading to poor latency splitting results. Piccolo proposes the latency-cost efficiency to measure the amount of cost that each module configuration can save per unit of latency budget, and then uses a heuristic to split the latency to minimize the total cost of the entire application. Moreover, Piccolo reassigns the remaining latency, which is the gap between the latency budget and the runtime latency, to modules to further reduce the total cost.

The third contribution of Piccolo is a novel dummy generator to increase the cost efficiency of machines with small workload. Machines with small workload can *not* choose configurations with high cost efficiency due to long batch collecting time. Though the dummy generator will increase the request rate for the given module, it enables machines with small workload to choose configurations with high cost efficiency, which reduces the total cost.

Piccolo is completely implemented as a containerized system deployed on a cluster of 16 GPUs with a total of 23k lines of Python code. Evaluation shows that Piccolo is able to minimize the total cost while satisfying the latency objectives. Compared to Piccolo, existing DNN serving systems (Nexus [3], Scrooge [4], InferLine [5] and Clipper [6]) require an average of 4.4% to 44.3% extra total cost. For certain workload, existing systems require more than 2 times the total cost of Piccolo's. For each workload, we generate the optimal



(a) (b)

Fig. 1: (a) round-robin dispatch; (b) batch dispatch.

Module M_1			Module M_2			Module M_3		
b	d	t	b	d	t	b	d	t
5	0.100	50	2	0.125	16	2	0.167	12
20	0.250	80	4	0.160	25	4	0.200	20
100	1.000	100	8	0.267	30	8	0.320	25

Table I: Profile for module M_1, M_2 and M_3, where b is batch size, d is execution duration in sec and t is module throughput in req/sec.

total cost with brute force searching and show that Piccolo derives the optimal solution for more than 97% workload, while Piccolo is more than 1000 times faster. In the ablation study, we quantify the importance of Piccolo's design choices.

II. Background and Motivation

In this section, we start with the scheduling policy of existing solutions, followed by three design dimensions distinguishing Piccolo from prior work: module throughput under the latency budget, latency-cost efficiency for total cost and configuration optimization for small workload.

How existing systems decide module configuration? Multiple factors affect the scheduling decisions. First, batching is widely used to increase the throughput for DNN modules by leveraging GPU's parallel computing capability with a larger batch size [3], [6]. Second, running multiple DNN modules concurrently on the same GPU can increase the GPU utilization by overlapping the module computation with data transfer via CUDA streams [4], [13]. Third, among the multiple types of computation hardware provided by the cloud provider, the most cost-efficient hardware type is module dependent [5], [14]. Therefore, when scheduling the DNN modules, the scheduler needs to find the optimal module configuration (*e.g.,* a combination of batch size, concurrency level and computation hardware) under the latency budget, leading to a multi-dimensional optimization problem.

To deal with the problem, existing solutions [3], [4] often rely on a two-round greedy heuristic. It first calculates the expected latency for all module configurations, and then greedily chooses the optimal one that maximizes the module throughput while the expected latency is within the latency budget. The inference system defines the maximum latency of all requests as *the worst case latency* L_{wc}, and optimizes the scheduling policy to guarantee that L_{wc} is smaller than the target latency objective. Existing solutions [3], [5], [6] dispatch requests among machines in round-robin as shown in Figure 1a, and use two times the execution duration as L_{wc}, where the first duration is the model execution time and the second one is the batch collection time.

For the optimal configuration c_{opt}, the scheduler will allocate $n = \lfloor T/t_{opt} \rfloor$ machines at c_{opt}, where T is the request rate for the given module and t_{opt} is its module throughput at c_{opt}. The workload of $n \cdot t_{opt}$ is called the *majority workload* T_{maj}. The remaining workload of $T - n \cdot t_{opt}$ is called the *residual workload* T_{res} if $T - n \cdot t_{opt} \neq 0$. The scheduler will re-run the greedy heuristic to get module configuration c_{res} for T_{res}, due to its batching delay. Therefore, the scheduling decision is usually a configuration set $\langle c_{opt}, c_{res} \rangle$. For instance, assuming $T = 285$ for module M_1 from Table I with latency SLO of 2.0 sec, existing solution will choose c_{opt} of batch size 100 for $T_{maj} = 200$, and c_{res} of batch size 5 for $T_{res} = 85$.

(1) How to schedule DNN modules to maximize the throughput under its latency budget? We argue that the scheduling decision of existing solutions can *not* maximize the throughput under the latency budget. Their round-robin dispatch strategy unnecessarily increases the worst case latency, leading to a lower overall resource efficiency. For example, in the above example, L_{wc} for T_{maj} is 2.0 sec under the round-robin dispatch strategy as shown in Figure 1a. However, if the scheduler dispatches request in batches instead of individual request as shown in Figure 1b, L_{wc} can be reduced from $2d$ to $1.5d$, where d is execution duration. By reducing L_{wc}, the scheduler can leverage the saved latency budget (*e.g.,* $0.5d$) to choose a larger batch size for higher resource efficiency.

Moreover, existing solutions use a fixed of two configurations $\langle c_{opt}, c_{res} \rangle$ for the majority and residual workload. We argue that by removing the limitation on number of configuration, the total cost can be further reduced. In the above example, existing solutions require two machines with batch size of 100 for T_{maj} and another two machines with batch size of 5 for T_{res} (where one of these two will run at partial capacity), leading to a total cost of $2 + 1 + 35/50 = 3.7$[1] machines. However, for T_{res}, if we use one machine with batch size of 20 and another machine with batch size of 5, the total cost can be reduced to $2 + 1 + (85 - 80)/50 = 3.1$ machines. Existing solutions can *not* support multiple configurations due to runtime complexity [3] or problem formulation [4]–[6].

Takeaways. Existing solutions' round-robin dispatch strategy unnecessarily increases the worst case latency and only supports a fixed of two module configurations for each module, which can *not* guarantee cost minimum scheduling. When scheduling DNN module, Piccolo should dispatch the requests properly to reduce the worst case latency and support multiple module configurations.

(2) How to split the end-to-end latency constraint to per-module latency budget? Recent applications start to use multiple DNN modules to improve the overall performance [3]–[5]. For example, the traffic monitoring application [11] uses three DNN modules to detect objects and to classify vehicles and pedestrians. For such application, only the end-to-end latency constraint for the entire application is provided. The scheduler is responsible to split it to per-module latency budget

[1]Similar to [4], [15], for machine at partial capacity, Piccolo calculates the cost according to its occupancy. See §III-A for details.

leverage
充分利用

argue
主张，争论，认为，论证

concurrent
同时的
concurrency
并发(性)

overlap
相交，重叠
与..部分相同

heuristic
启发式的

dispatch
分发 派遣 处决

budget
预算

pedestrian
步行者，行人

???

ID	Module M_2 (b)	Module M_3 (b)	Total cost
1	0.25 (2)	0.65 (8)	4.73
2	0.40 (4)	0.50 (4)	4.00
3	0.55 (8)	0.35 (2)	4.67

Table II: Three latency splitting strategies in sec with the chosen batch size in brackets and total cost in unit of machine numbers.

to minimize the total cost for all modules.

A larger latency budget for a given module can improve the maximum throughput for it, but might increase the total cost for the entire application, since the latency budget for the rest of modules is reduced. For example, for an application with two modules M_2 and M_3 from Table I in sequence, where M_2's output will be the input to M_3, we assume an input rate of 50 and 40 req/sec respectively with an end-to-end SLO of 0.9 sec. Among the three latency splitting strategies in Table II, strategy 2 achieves the minimum total cost. Though strategy 3 allocates more latency budget for module M_2 with higher throughput of M_2, it has a higher total cost due to a limited latency budget for module M_3.

Finding the optimal latency splitting strategy is known to be NP-Hard [3]–[5]. To get feasible solutions, Nexus [3] uses discretized split to reduce the search space, however, the search space is still exponential to number of modules and the reciprocal of the discretized interval. Scrooge [4] allocates the latency budget according to the duration of majority configuration, while InferLine [5] splits latency across modules in a greedy way, both of which can lead to sub-optimal solution.

Takeaways. For multi-DNN application, existing solutions' heuristic algorithms can not utilize the latency budget efficiently, leading to sub-optimal total cost. Piccolo should split the latency budget across each module properly to minimize the total cost for the entire application.

(3) How to optimize the cost efficiency for small workload? As discussed above, among a given module's active machine set, part of them will run at full capacity (*e.g.,* all machines for the majority workload T_{maj}), while the rest of machines will run at partial capacity (*e.g.,* some machines for the residual workload T_{res}). The module configuration for T_{res} often has a lower throughput than the one for T_{maj} due to lower request rate for T_{res}.

For example, in the above example of module M_1, the current optimal configuration will allocate four machines with batch size of 100, 100, 20 and 5. The achievable maximum throughput for these four machines will be 100, 100, 80 and 50 req/sec respectively. These four machines are using the same hardware, however, the throughput for the last two is much lower, suggesting a lower cost efficiency. The last two machines can *not* choose the configuration with a higher cost efficiency (*e.g.,* batch size of 100) due to their relatively low assigned request rate. If we choose a new machine with batch size of 100 to handle the residual workload of 85 req/sec instead, the worst case latency will be $1.0 + 100/85 = 2.18$ sec, which violates the latency SLO of 2.0 sec.

One interesting finding we have is that adding a proper amount of dummy requests can surprisingly reduce the total cost. For example, in the above example, if we add dummy requests of 15 req/sec, then the total workload of 300 req/sec only requires three machines with batch size of 100, reducing the total cost from 3.1 to 3.0 machines. Specifically, though the dummy requests will add extra workload, it can reduce the total cost by increasing the cost efficiency of machines for the residual workload. However, choosing the proper amount of dummy requests is challenging, since doing it naively (*e.g.,* adding dummy requests of 10 or 20 req/sec) will only add extra workload for the module without any cost reduction.

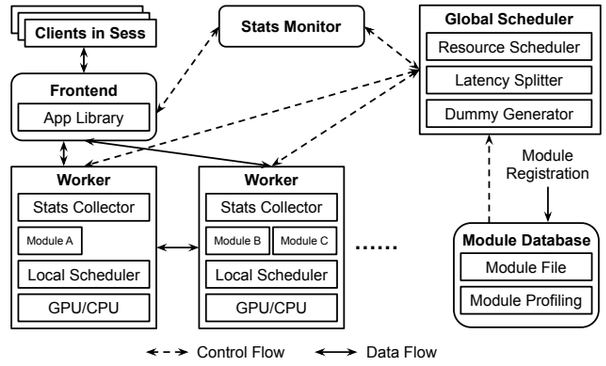


Fig. 2: Piccolo overview.

Takeaways. When scheduling DNN module, Piccolo should add a proper amount of dummy requests to increase the cost efficiency of machines for the residual workload, which can further reduce the total cost. To the best of our knowledge, none of existing solutions have considered adding dummy requests to reduce the total cost.

III. System Design

In this section, we start with an overview of Piccolo, followed by a detailed description of its components.

A. Overview

Terminology. Similar to prior work [3], [4], Piccolo classifies all requests that use the same application as a *session*. Each session has a unique *session id*, associated with three key elements: (1) an application directed acyclic graph (DAG) that specifies the application structure, where a node in the DAG represents a DNN module for GPU computation or a processing module for CPU computation, and an edge represents the computation dependencies between nodes; (2) a latency SLO that specifies the target end-to-end latency for the results of the entire application; and (3) the request rate for each module in the application.

To capture the resource usage of the CPU/GPU module in the DAG, Piccolo provides a profiling library for each module, which contains the execution duration of the given module under various configurations (*e.g.,* combinations of different batch size, concurrency level and computation hardware). Similar to prior work [3]–[5], [10], the profiling is collected offline once when the application is registered at the inference system, and will *not* affect the runtime latency of requests.

The cost is defined as the actual amount of computation resources used by the workload. Specifically, Piccolo takes advantage of multiple types of CPU/GPU hardware provided by the cloud provider at various prices to support hardware heterogeneity. Each session is charged by the resource occupancy on the hardware according to frame-rate proportionality [15], since the unused resource occupancy can be used by other jobs to avoid resource fragmentation [13], [16].

How Piccolo minimizes the total cost. As shown in Figure 2, for a given session, the global scheduler decides the module configuration for each module in the application and packs the module on the selected worker efficiently, so as to use the minimum total cost to serve the workload while satisfying its end-to-end latency constraint. Clients send requests to the frontend, which redirects requests to workers for module execution. Besides, the statistics monitor tracks the runtime information for scheduling adjustment.

Piccolo achieves the cost-minimum goal as follows:

► It schedules DNN modules efficiently to maximize the throughput under the latency budget (§III-B). Piccolo's request dispatcher distributes requests among machines properly to achieve a lower worst case latency than the state-of-the-art without extra computation resources, which makes module configurations with higher throughput feasible. Moreover, Piccolo's configuration generator supports multiple module configurations for each module to increase the resource efficiency.

► It splits the end-to-end latency constraint to per-module latency budget to minimize the total cost (§III-C). To do so, Piccolo proposes a novel latency-cost efficiency factor to capture the amount of cost that each module configuration can save per unit of latency budget, and then uses a greedy-based heuristic to split the latency for cost-minimum purpose. Besides, Piccolo reassigns the remaining latency budget to module's residual workload to further reduce the cost.

► It adds a proper amount of dummy requests to increase the cost efficiency of machines for the residual workload (§III-D). Piccolo's dummy generator monitors the residual workload and uses dummy requests to avoid the usage of module configuration with low cost efficiency, leading to a lower total cost.

B. Resource Scheduling

Problem, Challenges and Approach. To reduce the cost, Piccolo must decide the module configuration and scheduling policy to maximize the throughput for each module in the DAG under the latency budget, leading to two subtasks. First, for each module configuration, Piccolo needs to dispatch the requests among machines properly and estimate its worst case latency. Second, among module configurations that can satisfy the latency constraint, Piccolo needs to choose proper ones to maximize the throughput. As discussed in §II, existing solutions are not able to deal with these two subtasks properly. Piccolo addresses these two subtasks with its novel request dispatcher and configuration generator as follows.

Request Dispatcher. For each candidate set of module configurations, Piccolo will rank all machines according to their module configuration's throughput-cost ratio r, which is the module throughput divided by the unit price of the hardware type, and dispatch requests among machines in the order of throughput-cost ratio in sequence. For machines with the same throughput-cost ratio (*e.g.,* machines that uses the same module configuration), Piccolo will dispatch requests among them in batched requests. On the contrary, existing solutions [3], [5], [6] dispatch requests among machines in individual request. We call Piccolo's dispatch strategy throughput-cost (TC) dispatch policy, as opposed to existing solutions' round-robin (RR) dispatch policy.

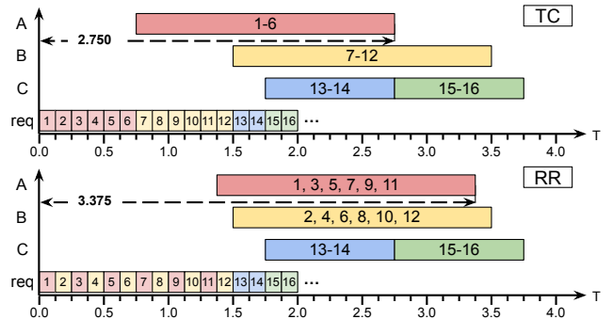


Fig. 3: Request dispatching results for throughput-cost dispatch policy (top) and round-robin dispatch policy (bottom).

TC dispatch policy can achieve a lower worst case latency than RR dispatch policy. For example, Figure 3 shows the request dispatch results using TC policy and RR policy respectively for a workload of 8 req/sec. There are three machines A, B and C, where both A and B use module configuration of batch size 6 with execution duration of 2.0 sec, and C uses module configuration of batch size 2 with execution duration of 1.0 sec. All three machines have homogeneous hardware with the same unit price, so the throughput-cost ratio rank will be $r_A = r_B > r_C$. For the first 16 requests, Piccolo's TC dispatch policy will dispatch req 1-6 to A, req 7-12 to B and req 13-16[2] to C, leading to a worst case latency of 2.75 sec. On the contrary, RR dispatch policy will send requests among A and B back and forth for the first 12 requests, and then send req 13-16 to C, leading to a worst case latency of 3.375 sec, which is much larger than TC dispatch policy's.

For a module, given a configuration set of n machines in the order of throughput-cost ratio: $r_1 \geq r_2 \geq \cdots \geq r_n$, we denote the module configuration of machine i as c_i and the assigned request rate on machine i as f_i. The *remaining workload* for machine i is defined as $w_i = \sum_{j=1}^{n} s_{ij} \cdot f_j$, where s_{ij} is 1 if $r_j \leq r_i$ and is 0 if $r_j > r_i$. Namely, w_i represents the total amount of request rate that is assigned to machines with throughput-cost ratio no larger than machine i's. For example, in the above example, the remaining workload for both machine A and machine B is $3 + 3 + 2 = 8$ req/sec, while the one for machine C is 2 req/sec.

[2]Request 13-16 will be divided into two batches to match C's module configuration of batch size 2.

Algorithm 1 Generating the configuration set for module M

```
1:  function GenerateConfig(T_M, L_M, P_M)
2:      rw ← T_M, configs ← []
3:      k ← 0, c ← P_M[k] = ⟨b, d, t⟩
4:      while rw ≠ 0 do
5:          if GetWCL(c) ≤ L_M then
6:              n ← rw/t
7:              if n ≥ 1 then
8:                  configs ← configs ⊕ ⟨c, ⌊n⌋⟩
9:                  rw ← rw - ⌊n⌋ × t
10:             else
11:                 configs ← configs ⊕ ⟨c, n⟩
12:                 rw ← 0
13:         else
14:             while GetWCL(c) > L_M do
15:                 c ← P_M[++k]
16:                 if k ≥ len(P_M) then
17:                     return False, []
18:     return True, configs
```

Theorem 1. Using TC dispatch policy, the worst case latency for machine i is $L_{wc}(i) = d_i + b_i/w_i$, where d_i and b_i is the execution duration and batch size under module configuration c_i. The worst case latency for the module is $\max_{i=1}^{n} L_{wc}(i)$.

Proof. Piccolo's TC dispatch policy sends batched request among machines in the order of throughput-cost ratio. For each machine i, its remaining workload w_i is actually the equivalent *batch collecting rate* for its module configuration in each iteration, leading to a batch collecting duration of b_i/w_i. Therefore, machine i's worst case latency $L_{wc}(i)$ will be $d_i + b_i/w_i$, and the module's worst case latency will be the maximum of each machine's $L_{wc}(i)$. □

As shown in Theorem 1, Piccolo's TC dispatch policy can achieve a worst case latency of $d_i + b_i/w_i$, which is much smaller than existing solutions' $2d_i$ worst case latency. By achieving a smaller worst case latency, Piccolo is able to choose a configuration with larger throughput or allocate more latency budget to other modules for multi-DNN applications, both of which lead to a smaller total cost (§IV).

Configuration Generator. Given the configurations alongside the worst case latency, Piccolo then chooses the configuration set to maximize the throughput. Unlike existing solutions which only support a maximum of two module configurations, Piccolo supports multiple module configurations.

Piccolo derives the configuration set for module M using Algorithm 1, where the input is the request rate T_M, the latency budget L_M and the profiling P_M ordered by throughput-cost ratio (Line 1). Piccolo maintains the current unallocated workload rw, whose initial value is T_M (Line 2). The configuration index k, starting at 0, records the current configuration c (Line 3). Function $GetWCL()$ estimates the worst case latency as described in Theorem 1. Algorithm 1 generates the configuration set greedily. If the current configuration c can satisfy the latency budget (Line 5), Piccolo will use c to update $configs$. When the number of machine n is larger than 1, $\lfloor n \rfloor$ machines will run at c at full capacity (Line 8) and corresponding amount of workload will be deducted from rw (Line 9). Otherwise, 1 machine will run at c at partial capacity of n (Line 11) and set rw to 0 (Line 12). If c can *not* satisfy the latency budget, Piccolo will find the next feasible configuration. If no such configuration exists, Piccolo returns an error alongside an empty configuration set.

Algorithm 2 Deriving per-module latency budget

```
1:  function SplitLatency(T, L, P)
2:      DAG ← GetDefaultDAG()
3:      flag ← True
4:      while flag do
5:          flag ← False, LC_max ← −∞
6:          M_max ← NULL, c_max ← NULL
7:          for M in DAG do
8:              T_M ← T[M], P_M ← P[M]
9:              c_prev ← DAG[M]
10:             for c_new in P_M do
11:                 LC ← (p_new×T_M/t_new − p_prev×T_M/t_prev) / (GetWCL(c_prev) − GetWCL(c_new))
12:                 if LC > LC_max then
13:                     if GetLat(DAG, M_max, c_max) ≤ L then
14:                         flag ← True, LC_max ← LC
15:                         M_max ← M, c_max ← c
16:         if flag then
17:             updateDAG(DAG, M_max, c_max)
18:     for M in DAG do
19:         L_M ← (GetWCL(DAG[M]) / GetLat(DAG, M, DAG[M])) × L
20:     return [L_M for M in DAG]
```

C. Latency Splitting

Problem, Challenges and Approach. For applications with multiple modules, a proper latency splitting strategy is critical to guarantee minimum total cost. Deriving the optimal latency splitting is known to be NP-hard [3]–[5]. Therefore, Piccolo proposes *latency-cost efficiency* and designs a greedy heuristic to split the latency. Besides, for a given configuration, there could be a gap between its worst case latency and latency budget. Piccolo reassigns the remaining latency budget to modules to further reduce the total cost.

Latency Splitter. Piccolo defines the latency-cost efficiency LC as the amount of cost that a given configuration can reduce per unit of latency budget when switching from the previous configuration to the new one. For example, for module M_1 from Table I, we assume the previous configuration has batch size of 5 to deal with a workload of 100 req/sec and hardware is homogeneous with unit price of 1.0. The other two configurations with batch sizes of 20 and 100 respectively are the candidates to switch to. According to the definition, the latency-cost efficiency for configuration with batch size of 20 is $\frac{1.0\times100/50 - 1.0\times100/80}{(0.25+20/100)-(0.1+5/100)} = 2.50$, while the one for configuration with batch size of 100 is 0.54.

Based on the latency-cost efficiency, Piccolo uses Algorithm 2 to partition the end-to-end latency among the modules in the application. It uses DAG to store the current configuration for each module, starting at a default DAG (Line 2), which has the minimum worst case latency. In most cases, each module in the default DAG has batch

size of 1 and concurrency level of 1 on hardware with the highest unit price, which corresponds to the *least cost-efficient* configuration. Function $GetLat()$ calculates the end-to-end latency under the current module configurations as specified by DAG. Among all configurations of all modules in the DAG, Algorithm 2 chooses the next configuration whose latency-cost efficiency (Line 11, where p is the unit price) is the maximum, while the latency constraint can be satisfied, and then updates the corresponding module's configuration to the new one (Line 17). Algorithm 2 repeats the above process until no configuration can be updated while preserving the latency constraint. The latency budget for each module is resized linear to its worst case latency (Line 19).

By preferring configuration with higher latency-cost efficiency instead of higher throughput, Piccolo utilizes the end-to-end latency more efficiently. For example, in the above example of module M_1 from Table I, Algorithm 2 prefers batch size of 20 over 100 for the current iteration. Though batch size 20's throughput is lower than batch size 100's, its cost reduction per unit of latency budget is higher ($2.50 > 0.54$), leaving more latency space for future iterations to further reduce the cost. By doing so, Piccolo achieves a lower total cost than throughput-based method (§IV).

Reassigning the remaining latency budget. Piccolo's configuration generator (§III-B) generates the configuration set that maximizes the throughput while the worst case latency of the given configuration set is smaller than the latency budget derived by Algorithm 2. However, there can still be a gap between the worst case latency and the latency budget, which can be used to further reduce the cost when scheduled properly. Piccolo re-assigns the remaining latency budget to each module's residual workload greedily. By doing so, Piccolo allows the residual workload to choose a configuration with larger throughput, which reduces the total cost (§IV).

D. Futher Reducing the Cost

Problem, Challenges and Approach. Piccolo's configuration generator (§III-B) and latency splitter (§III-C) maximize the throughput under the latency constraint. However, the throughput-cost efficiency of the last few configuration ordered by the throughput-cost ratio is much lower due to relative low request rate for the residual workload. Therefore, to handle the low throughput-cost limitation for residual workload, Piccolo uses dummy generator to increase its throughput-cost ratio.

Dummy Generator. As discussed in §II, Piccolo might have to choose configurations with relatively low throughput-cost efficiency for the residual workload due to queueing delay. These configurations can *not* fully utilize the computation capability of the underlying hardware, leading to low hardware utilization. Therefore, Piccolo adds a proper amount of dummy requests to the given module, which can *surprisingly* reduce the cost, though the total request rate (the original request rate plus the dummy request rate) is higher.

Given a module with K distinct configurations ordered by throughput-cost ratio: $\frac{t_1}{p_1} > \frac{t_2}{p_2} > \cdots > \frac{t_K}{p_K}$, where each module configuration c_i with throughput t_i and unit price p_i is allocated n_i machines, Piccolo defines the *leftover workload* for c_i as $u_i = \sum_{j=i+1}^{K} n_j t_j$. Different from the remaining workload in §III-B, the leftover workload represents the total amount of request assigned to machines with throughput-cost ratio less than the given module configuration's.

Theorem 2. In the cost-minimum configuration, the leftover workload u_i for the ith configuration c_i ordered by throughput-cost ratio should be less than its throughput. Namely, $\forall i \in \mathbb{K}, u_i = \sum_{j=i+1}^{K} n_j t_j < t_i$.

Proof. We use proof of contradiction, where $\exists k \in \mathbb{K}, u_k = \sum_{j=k+1}^{K} n_j t_j \geq t_k$. For u_k, we define the total number of machine as $n_\alpha = \sum_{j=k+1}^{K} n_j$, the average unit price as $p_\alpha = \sum_{j=k+1}^{K} n_j p_j / n_\alpha$, and the average throughput as $t_\alpha = u_k / n_\alpha$. The total cost for u_k is $C_0 = \sum_{j=k+1}^{K} n_j p_j$. Since $u_k \geq t_k$, we allocate a new machine at c_k to handle t_k workload for higher throughput-cost. Depending on the worst case latency for the remaining $u_k - t_k$ workload, there will be two cases.

In the first case, the original configuration for $u_k - t_k$ can still satisfy the latency constraint, then Piccolo will switch t_k amount of workload from its original configuration to c_k for higher throughput-cost efficiency and keep $u_k - t_k$ workload unchanged. We define $T_1 = \lfloor u_k / t_k \rfloor \cdot t_k$, then the new total cost for u_k is $C_1 = p_k \cdot \lfloor u_k / t_k \rfloor + p_\alpha \cdot (u_k - T_1)/t_\alpha$. We have $C_0 - C_1 = T_1(p_\alpha / t_\alpha - p_k / t_k)$. According to definition, $t_k / p_k > t_\alpha / p_\alpha$, so $C_0 - C_1 > 0$. Namely, the cost-minimum assumption is violated.

In the second case, the original configuration for $u_k - t_k$ can not satisfy the latency constraint, then Piccolo will choose a new configuration c_β to deal with the remaining $u_k - t_k$ workload to guarantee the latency SLO. Similarly, the new total cost for u_k is $C_2 = p_k \cdot \lfloor u_k / t_k \rfloor + p_\beta \cdot (u_k - T1)/t_\beta$. We have $C_0 - C_2 = T_1(\frac{p_\alpha}{t_\alpha} - \frac{p_k}{t_k}) - (u_k - T_1)(\frac{p_\beta}{t_\beta} - \frac{p_\alpha}{t_\alpha})$. According to definition, $T_1 > u_k - T_1$, and for most modules, $\frac{p_\alpha}{t_\alpha} - \frac{p_k}{t_k} \approx \frac{p_\beta}{t_\beta} - \frac{p_\alpha}{t_\alpha}$, so $C_0 - C2 > 0$. Namely, the cost-minimum assumption is violated. □

Piccolo calculates the proper amount of dummy requests for each module to further reduce the total cost. For a given module M with K distinct module configurations ordered by the throughput-cost ratio, Piccolo goes through each configuration $c_i = \langle b_i, d_i, t_i \rangle$ to check whether its leftover workload can be processed by its own configuration with dummy request of $dum_i = \frac{b_i}{L_M - d_i} - u_i$, where L_M is the latency budget for the given module derived from Algorithm 2. Based on Theorem 2, c_i's leftover workload should be less than t_i. Therefore, if $dum_i \geq 0$, Piccolo will add dummy request of dum_i for the given module. For example, in the above example of module M_1 with request rate of 285 req/sec, for configuration of batch size 100, $u_i = 85$ and $dum_i = \frac{100}{2.0 - 1.0} - 85 = 15$ req/sec. By doing so, Piccolo can further reduce the cost (§IV).

IV. Evaluation

We compare Piccolo against four recent inference systems, then perform an ablation study that quantifies the importance of Piccolo's various design decisions.

System	Batch	Spatial	Hetero	Worst case latency	Num of configs	Latency splitting strategy	Latency reassign	Dummy
Piccolo	✓	✓	✓	$d + b/w$	any	latency-cost efficiency	✓	✓
Nexus [3]	✓			$2d$	2	quantized interval		
Scrooge [4]	✓	✓	✓	$d + b/t$	2	local throughput based		
InferLine [5]	✓		✓	$2d$	1	greedy		
Clipper [6]	✓			$2d$	1	evenly splitting		

Table III: Comparison of Inference Systems

A. Methodology

Implementation. We have implemented all the features of Piccolo described in §III. Piccolo has roughly 8k lines of Python code for its core scheduling functionality, and an additional 15k lines for the app library. Each component of Piccolo runs in a separate container for resource isolation and elastic scalability. Piccolo supports the execution of modules trained by various frameworks including Caffe [17], Tensorflow [18], PyTorch [19].

Testbed. We deploy Piccolo on a cluster with 16 GPUs (8 P100 GPUs and 8 V100 GPUs, each with 6 vCPUs, 112 GB RAM and 128 GB SSD) from Microsoft Azure [20]. We deliberately design the testbed to demonstrate Piccolo's capability to minimize the total cost for heterogeneous hardware.

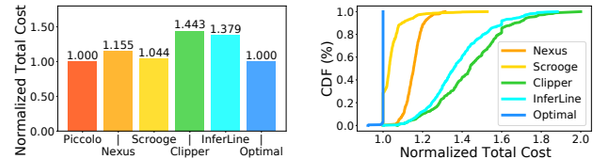
Workloads. For fair comparison, we choose five multi-DNN applications used in existing systems [3]–[6]: *traffic* [11] uses SSD-Inception [21] to detect vehicle and pedestrian in traffic video, *face* uses PRNet [22] to detect facial keypoints, *pose* uses OpenPose [23] to recognize human poses, *caption* uses S2VT [24] to generate text description of video streams and *actdet* [25] detects human activities across camera footage. The video streams we use in our evaluation come from public datasets or the application developers [3], [24]–[26].

Comparison Alternatives. We compare Piccolo against four alternative systems: Nexus [3], Scrooge [4], InferLine [5] and Clipper [6], as well as the optimal solution derived from brute force searching. Similar to Piccolo, all four alternative systems schedule DNN-based applications to minimize the total cost while satisfying the latency objectives. To serve multi-DNN application, Nexus uses quantized interval to split latency among modules, Scrooge proposes an MILP-based solution to maximize the resource efficiency across modules, InferLine leverages a greedy heuristic to select proper module configuration for each module. Clipper has limited support for multi-module application, so we split latency across modules equally as in [3], [4].

Metrics. For all experiments, we mainly focus on the total cost of the system for the given workload. Following existing literature [3], [4], [15], each session is charged by the resource occupancy on the hardware according to frame-rate proportionality. The ideal total cost should be as small as possible.

B. Comparison Results

Figure 4a shows the average normalized total cost for Piccolo, four alternative systems and the optimal solution, while Figure 4b shows the cumulative distribution function (CDF) of normalized total cost. Among five inference systems, Piccolo achieves the minimum total cost for *all* workload.



(a) **(b)**

Fig. 4: (a) Normalized total cost; (b) CDF of normalized total cost.

Compared to Piccolo, the four alternative systems require an average extra cost of 4.4%-44.3% and a maximum extra cost of 31.4%-101.8%. Table III shows the differences in design choices between Piccolo and the four alternatives. Since the performance differences might come from a combination of multiple factors, we conjecture that the advantages of Piccolo come from the following ones. In §IV-C, we provide a detailed ablation study to quantify the importance of each design choice on cost reduction.

First, the resource scheduling methods of the alternative systems can *not* maximize the resource efficiency. Piccolo supports batching, spatial multiplexing and heterogeneous hardware, while most of the alternative systems only support part of them, leading to a smaller search space for scheduling optimization than Piccolo's. Besides, Piccolo's TC dispatch policy achieves a lower worst case latency than alternative systems'. Therefore, Piccolo can leverage the saved latency to choose configuration with larger throughput or allocate more latency budget to other modules, both of which can reduce the total cost. Moreover, Piccolo supports multiple configurations for each module for higher resource efficiency, as opposed to alternative systems' fixed 1 or 2 configuration.

Second, the latency splitting methods of the alternative systems can *not* utilize the latency efficiently. Early inference systems [5], [6] have limited support for multi-module application, leading to poor latency splitting strategies. Recent inference systems [3], [4] use quantized latency interval or throughput-based latency splitting strategies, which has high runtime complexity or sub-optimal result. Besides, Piccolo reassigns the remaining latency budget, which is the gap between the per-module latency budget and the actual worst case latency, to further minimize the total cost, while the alternative systems have no such capability.

Third, the novel dummy generator of Piccolo increases the throughput-cost efficiency of the residual workload for each module, which further reduces the total cost, while the alternative systems have no such capability.

Besides, as shown in Figure 4b, compared to the optimal

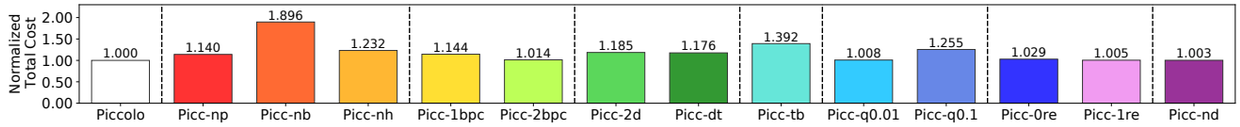


Fig. 5: The average normalized total cost for Piccolo and alternative Piccolo without corresponding functionalities.

Fig. 6: Normalized throughput for Picc-np, Picc-nb and Picc-nh.

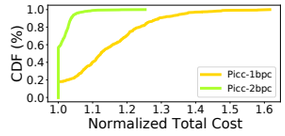


Fig. 7: CDF of normalized total cost for Picc-1bpc and Picc-2bpc.

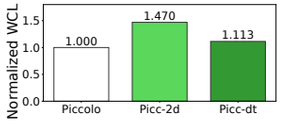


Fig. 8: Normalized worst case latency for Picc-2d and Picc-dt.

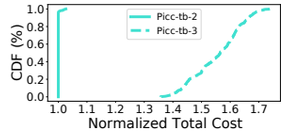


Fig. 9: CDF of normalized total cost for Picc-tb with 2 or 3 modules.

solution derived from brute force, Piccolo's total cost is larger than the optimal for only 2.87% workload with up to 7.69% extra total cost. However, the average runtime of the optimal solution using brute force algorithm is 10.29 sec per scheduling, while Piccolo's average runtime is only 0.008 sec. Namely, Piccolo derives the optimal solution for 97.13% workload, while Piccolo is more than 1000 times faster.

C. Ablation Study

In this section, we disable each of Piccolo's novel features to quantify their importance on cost reduction.

The importance of multi-dimensional scheduling. Piccolo supports batching, spatial multiplexing and heterogeneous hardware to maximize the resource throughput under the latency constraint. To evaluate the benefit of it, we compare Piccolo against: (1) Picc-np, which disables spatial multiplexing, (2) Picc-nb, which disables batching, and (3) Picc-nh, which uses only one type of hardware.

Figure 5 includes the average normalized total cost for the three alternatives. Picc-nb has the highest total cost with an average extra cost of 89.6%, suggesting the importance of batching on achieving the cost minimum goal. Picc-np and Picc-nh requires an average of 14.0% and 23.2% extra total cost, which also indicates the necessity of using spatial multiplexing and heterogeneous hardware to reduce the cost. To understand why, Figure 6 shows the normalized throughput for a given module under 663 workload. The average throughput for Picc-np, Picc-nb and Picc-nh is 11.4%, 42.9% and 47.0% lower than Piccolo's. Interestingly, Picc-np's throughput for the given module is larger than Piccolo among 47/663 workload. This is because after disabling spatial multiplexing, Picc-np splits the latency among modules differently, leaving the given module a larger latency budget. For all 47/663 workload, the average throughput of Picc-np for the other modules is much lower than Piccolo's, leading to a higher total cost for Picc-np.

The importance of multiple configurations. Piccolo supports multiple configurations to maximize the throughput. To evaluate the benefit of it, we compare Piccolo against: (1) Picc-1bpc, which assigns each module one configuration [5], [6], and (2) Picc-2bpc, which assigns each module at most two configurations [3], [4].

Figure 5 includes the average normalized total cost for Picc-1bpc and Picc-2bpc. Using 1 or 2 configurations requires an average of 14.4% and 1.4% extra total cost. Figure 7 shows the CDF of the normalized total cost, where Picc-1bpc has the same total cost as Piccolo for 17.95% workload and a maximum of 61.60% extra cost. Meanwhile, Picc-2bpc has the same total cost as Piccolo for 56.26% workload and a maximum of 25.30% extra cost. Among all workload, Piccolo assigns 43.74% workload with more than 2 configurations to achieve a lower total cost.

The importance of TC dispatching. Piccolo uses TC dispatching to minimize the worst case latency for each module in the application. To quantify its benefit, we compare Piccolo against: (1) Picc-2d, which uses RR dispatch policy [3], [5], [6] as discussed in §III-B, and (2) Picc-dt, which uses machine's local throughput information to dispatch request [4].

Figure 5 includes the average normalized total cost for Picc-2d and Picc-dt, which require an average of 18.5% and 17.6% extra total cost. To understand why, Figure 8 shows the average normalized worst case latency under various workload. For a given configuration, Picc-2d's RR dispatch policy and Picc-dt's machine-based dispatch policy require an extra latency budget of 47.0% and 11.3% on average. Therefore, Piccolo's TC dispatch policy can reduce the worst case latency, leading to a smaller total cost.

The importance of latency-cost efficiency. Piccolo uses a heuristic algorithm based on latency-cost efficiency to split the end-to-end latency into per-module latency budget. To demonstrate the benefit of it, we compare Piccolo against Picc-tb, which splits the end-to-end latency according to throughput instead of latency-cost efficiency.

Figure 5 includes the average normalized total cost for Picc-tb, which requires an average of 39.2% extra total cost. Figure 9 shows the CDF of normalized total cost for Picc-tb on application with two or three modules. Picc-tb with two modules has the same total cost as Piccolo for 94.36% workload and a maximum extra total cost of 2.6%, while Picc-tb with three modules has higher total cost for *all* workload with an extra total cost from 35.7% to 73.8%. This is because the more modules the application have, the latency splitting

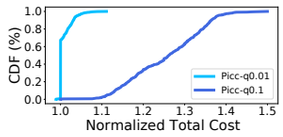


Fig. 10: CDF of normalized total cost for Picc-q0.01 and Picc-q0.1.

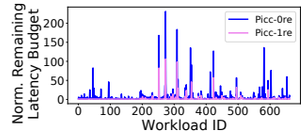


Fig. 11: Norm. remaining latency budget for Picc-0re and Picc-1re.

space is larger. Therefore, latency splitting strategies like Picc-tb is more likely to reach local optimal for applications with more modules, while Piccolo's splitting method based on latency-cost efficiency can utilize the latency budget properly.

The importance of not-quantized. To derive the per-module latency budget, another alternative is to quantize the end-to-end latency SLO into discrete intervals for each module [3]. To show the benefit of Piccolo's latency splitting strategy, we compare Piccolo against: (1) Picc-q0.01, which quantizes the end-to-end latency SLO into discrete interval in step of of 0.01 sec, and (2) Picc-q0.1, which quantizes in step of 0.1 sec.

Figure 5 includes the average normalized total cost for the two alternatives, while Figure 10 shows the corresponding CDF. Picc-q0.1 requires an average of 25.5% extra total cost and a maximum of 50.0% extra total cost. This is because the 0.1 sec discrete interval is too coarse to split the latency SLO properly. The average total cost for Picc-q0.01 is close to Piccolo's with an average of 0.8% and a maximum of 11.1% extra total cost. Interestingly, for around 1% workload, the total cost of Picc-q0.01 is smaller than Piccolo's. This is because the quantized method is another kind of brute force searching, similar to the one we used to find the optimal. However, as discussed in §IV-B, brute force method is too slow. The average runtime of Picc-q0.01 is 849 ms. Namely, though Picc-q0.01 has lower cost for 1% workload, it has higher cost for 35.44% workload and a runtime complexity more than 100 times of Piccolo's.

The importance of remaining latency reassignment. Piccolo reassign the remaining latency budget to modules to further reduce the total cost. To quantify the benefit of it, we compare Piccolo against: (1) Picc-0re, which doesn't reassign the remaining latency budget, and (2) Picc-1re, which only reassigns the remaining latency budget to one of the modules greedily.

Figure 5 includes the average normalized total cost for Picc-0re and Picc-1re, which require an average of 2.9% and 0.5% extra total cost. Figure 11 shows the normalized remaining latency budget. The remaining latency budget for Picc-1re is 2.11 times of Piccolo's, while the one for Picc-0re is 6.23 times of Piccolo's with a maximum of more than 200 times larger. Therefore, reassigning the remaining latency budget can increase the latency utilization and reduce the total cost.

The importance of dummy requests. Piccolo adds a proper amount of dummy requests to increase module's throughput-cost efficiency. To quantify its importance, we compare Piccolo against Picc-nd, which disables dummy generator.

Figure 5 includes the average normalized total cost for Picc-nd, which requires an extra total cost of 0.3% on average. Specifically, Picc-nd has the same total cost for 81.15% workload, while for the rest workload, Picc-nd requires up to 5.9% extra total cost (CDF omitted for brevity).

V. Related Work

DNN Inference Systems. Multiple DNN inference systems have been proposed recently, but Piccolo differs from them with its unique designs. Nexus [3] is designed to schedule DNN models under latency constraints. However, it uses round-robin dispatch policy, which can not achieve the cost-minimum goal. Scrooge [4] supports multi-dimensional scheduling, however, it only considers a maximum of two configurations and uses throughput-based heuristic to split the latency, leading to sub-optimal solution. Clipper [6], InferLine [5] and AutoDeep [27] only consider one configuration per module and have limited support to split the latency. Edge systems [15], [28]–[30] are designed for small cluster and the scheduling algorithm does not scale.

INFaaS [10] chooses appropriate DNN model among multiple variants by using a state machine to track variant performance. MArk [31] uses VM to serve incoming ML workload for cost-efficiency, and serverless functions to reduce tail latency under workload burst. Clockwork [32] disables just-in-time compilation and uses time sharing to guarantee performance predictability. However, these systems only support applications with at most one GPU module. Triton [16], Space-Time [33] and OoO [34] use CUDA streams to run multiple DNN modules concurrently on the same GPU and support kernel fusion to accelerate DNN inference, but require manual specification for module configuration.

DNN Training Systems. Gandiva [13] and AntMan [35] use spatial multiplexing and heterogeneous hardware to accelerate the training process. Tiresias [36], Themis [37] and OMGS-SGD [38] schedule training jobs in the cluster to minimize the job completion time or increase the resource utilization. The design of Piccolo is inspired by these DNN training systems, however, their scheduling capability can not satisfy inference job's millisecond latency objectives.

VI. Conclusion

In the paper, we presented a cost-efficient DNN inference system called Piccolo, which schedules DNN workload efficiently to minimize the total cost while satisfying the latency objectives. Piccolo dispatches requests across machines based on throughput-cost and assigns multiple configuration for each module to maximize the resource utilization. Besides, it splits the end-to-end latency into per-module latency budget using latency-cost efficiency and reassigns the remaining latency budget to maximize the latency efficiency. Moreover, Piccolo adds a proper amount of dummy requests to increase the cost efficiency of machines with small workload, leading to a lower total cost. Our system is fully implemented and evaluation shows that Piccolo can save up to 2 times the total cost than the state of the art, while satisfying the latency objective. Piccolo derives the optimal solution for more than 97% workload, while Piccolo is more than 1000 faster.

REFERENCES

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[3] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[4] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. Scrooge: A cost-effective deep learning inference system. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 624–638, 2021.

[5] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 477–491, 2020.

[6] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[7] Tianhang Zheng and Baochun Li. Poisoning attacks on deep learning based wireless traffic prediction. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 660–669. IEEE, 2022.

[8] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scalemulti-tenantgpu clusters for dnn training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[9] Daniel Uvaydov, Salvatore D'Oro, Francesco Restuccia, and Tommaso Melodia. Deepsense: Fast wideband spectrum sensing through real-time in-the-loop deep learning. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[10] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. Infaas: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021.

[11] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, 2017.

[12] Reduce inference costs on Amazon EC2. https://aws.amazon.com/blogs/machine-learning/reduce-inference-costs-on-amazon-ec2-for-pytorch-models-with-amazon-elastic-inference/.

[13] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[14] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.

[15] Yitao Hu, Weiwu Pang, Xiaochen Liu, Rajrup Ghosh, Bongjun Ko, Wei-Han Lee, and Ramesh Govindan. Rim: Offloading inference to the edge. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, pages 80–92, 2021.

[16] Nvidia triton inference server. https://developer.nvidia.com/nvidia-triton-inference-server, 2022.

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

[18] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[20] https://azure.microsoft.com/en-us/, 2022.

[21] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[22] Yao Feng, Fan Wu, Xiaohu Shao, Yanfeng Wang, and Xi Zhou. Joint 3d face reconstruction and dense alignment with position map regression network. In *Proceedings of the European conference on computer vision (ECCV)*, pages 534–551, 2018.

[23] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7291–7299, 2017.

[24] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. Sequence to sequence-video to text. In *Proceedings of the IEEE international conference on computer vision*, pages 4534–4542, 2015.

[25] Xiaochen Liu, Pradipta Ghosh, Oytun Ulutan, BS Manjunath, Kevin Chan, and Ramesh Govindan. Caesar: cross-camera complex activity recognition. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, pages 232–244, 2019.

[26] https://www.earthcam.com/, 2022.

[27] Yang Li, Zhenhua Han, Quanlu Zhang, Zhenhua Li, and Haisheng Tan. Automating cloud deployment for deep learning inference of real-time online services. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1668–1677. IEEE, 2020.

[28] Chenghao Hu and Baochun Li. Distributed inference with deep learning models across heterogeneous edge devices. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 330–339. IEEE, 2022.

[29] Jiawei Zhang, Suhong Chen, Xudong Wang, and Yifei Zhu. Deepreserve: Dynamic edge server reservation for connected vehicles with deep reinforcement learning. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[30] Tianxiang Tan and Guohong Cao. Fastva: Deep learning video analytics through edge processing and npu in mobile. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1947–1956. IEEE, 2020.

[31] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (ATC 19)*, pages 1049–1062, 2019.

[32] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[33] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic space-time scheduling for gpu inference. *arXiv preprint arXiv:1901.00041*, 2018.

[34] Paras Jain, Xiangxi Mo, Ajay Jain, Alexey Tumanov, Joseph E Gonzalez, and Ion Stoica. The ooo vliw jit compiler for gpu inference. *arXiv preprint arXiv:1901.10008*, 2019.

[35] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on gpu clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.

[36] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong 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.

[37] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient gpu cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[38] Shaohuai Shi, Qiang Wang, Xiaowen Chu, Bo Li, Yang Qin, Ruihao Liu, and Xinxiao Zhao. Communication-efficient distributed deep learning with merged gradient sparsification on gpus. In *IEEE INFOCOM 2020*, pages 406–415. IEEE, 2020.