Irina: Accelerating DNN Inference with Efficient Online Scheduling

Xiaorui Wu
City University of Hong Kong
Hong Kong SAR, China

Hong Xu
City University of Hong Kong
Hong Kong SAR, China

Yi Wang
Peng Cheng Laboratory
Shenzhen, China

ABSTRACT

DNN inference is becoming prevalent for many real-world applications. Current machine learning frameworks usually schedule inference tasks with the goal of optimizing throughput under predictable workloads and task arrival patterns. Yet, inference workloads are becoming more dynamic with bursty queries generated by various video analytics pipelines which run expensive inference only on a fraction of video frames. Thus it is imperative to optimize the completion time of these unpredictable queries and improve customer experience.

We propose the preliminary design of the first online inference task scheduling system, called *Irina*, that takes *completion time* under *unpredictable* workload as its primary objective. Irina augments the design space of inference task scheduling with three new strategies, namely batching, stacking, and preemption, in order to more flexibly schedule the tasks and reduce overall latency. Simulation results with empirical inference execution data shows that Irina can improve average task completion time by 1.3x–2.5x over TensorFlow Serving scheduling.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Theory of computation** → **Distributed algorithms**; • **Software and its engineering** → **Scheduling**.

This work was done when Xiaorui was visiting University of Chicago. The authors would like to thank Junchen Jiang for helpful discussions about this project.

APNet '20, August 3-4, 2020, ONLINE

1 INTRODUCTION

Low-latency DNN (deep neural network) inference over live data streams has emerged as an important workload, most notably in live video analytics [1, 27]. Enabled by advanced vision models, live video analytics is ubiquitously used, but unlike other DNN applications (e.g., model training), its workload can be highly *unpredictable*. Interesting events in real-world videos tend to occur intermittently and such temporal patterns have been exploited by many video pipelines (e.g., [14, 27]) to opportunistically avoid expensive DNN inference on most video frames. With wider camera deployments, more video streams—each with unpredictable workload—could create more workload variance as a whole.

Unfortunately, recent inference frameworks are not well-suited for maintaining low inference latency under unpredictable workloads. While these frameworks optimize throughput or utilization through optimal batching and scheduling, they are most effective only when the workload is largely predictable. For instance, Nexus [21] splits time into epochs and in each epoch it schedules inference tasks from a predetermined list in a way that exploits the inherent parallelism of GPU. It assumes that the workloads in one epoch are stable and predictable. However, these schemes become less efficient when future tasks arrive at unpredictable intervals and/or with dynamic GPU/CPU demands; e.g., many video analytics pipelines use expensive DNNs only on a small subset of frames with unpredictable intervals (e.g., [14]). Other frameworks exploit the DNN architectures to speedup inference, such as opportunistically caching intermediate feature maps across DNNs that share layers [13]. None of them, however, explicitly deals with unpredictable workloads or minimizes the delay per inference task.

In this preliminary work, we present *Irina*, a novel DNN inference scheduler tailored to reducing delay under unpredictable workloads. It explores several strategies that are largely overlooked in existing systems to efficiently share the available (GPU) resources and minimize average inference delay.

- **Batching:** When receiving new queries (e.g., inference on a new set of images), Irina dynamically decides whether to batch them with the ongoing inference task (if they use the same DNN). As we elaborate in §2.2,

DNN inference almost always underutilizes the GPU cores, so opportunistically batching these tasks could greatly reduce average inference delay.

- **Preemption:** When the new query and the ongoing query use different DNNs (i.e., cannot be batched), Irina will dynamically decide if it is beneficial to preempt the ongoing query and start the new query instead. This can be particularly efficient when the new query is short or has a low delay tolerance.
- **Stacking:** Finally, Irina supports packing two tasks together when there is enough spare resource on a currently underutilized GPU. This is particularly useful when the arrivals of queries using large DNNs are interleaved with those using small DNNs.

Irina implements these techniques transparently without requiring modifications to the query code.

While some strategies, for example dynamic batching and preemption, have been used in other context, our contributions lie in harnessing their potential to reduce online DNN inference delay under unpredictable workloads. We also investigate the systems challenges to enable these control knobs in common DNN frameworks. We use empirical DNN inference execution data from a small GPU testbed to conduct simulation to evaluate the performance of Irina. Our preliminary results show that Irina can achieve 1.3x – 2.5x speedups over state-of-the-art schedulers.

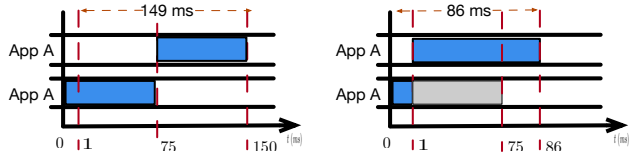
2 MOTIVATION

In DNN inference, multiple applications and DNN models share the GPU cluster. Existing inference task schedulers commonly use canonical scheduling principles (FIFO or shortest job first) to sequentially schedule tasks and dedicate a GPU for a task at one time [4, 6]. This is simple to implement but misses many opportunities to minimize job completion times (JCT). In many cases, customer experience is influenced by query response time and any additional delay of inference results can cause revenue losses and should be minimized by the scheduler.

This section presents concrete examples to make a case for three scheduling strategies, which would later become the building blocks of Irina. Throughout the examples we assume two types of inference task—one that requires more GPU memory and relatively long delay (e.g., FasterRCNN-ResNet152 for object detection) and one that requires less GPU memory and relatively short delay (e.g., LeNet [17] for license plate recognition). The performance of these DNN models are given in Table 1 (used in §2.1 and §2.3) and Table 2 (used in §2.2). The data is extracted from the MLperf inference benchmark [3] with minor simplifications for ease of exposition.

Model	Batch Size	Latency (ms)	Throughput (reqs/s)
A	4	60	66
	8	75	106
	16	85	188
	32	150	213
B	128	8	16000
	256	10	25600

Table 1: Execution information for DNN models used in the examples in §2.1 and §2.3.



(a) Current schedule **(b) Batching**

Figure 1: An illustrative example (with realistic delay numbers) where opportunistically batching two tasks of the same DNN reduces the average inference delay.

2.1 Opportunistic Batching

Let us consider a scenario where 8 queries for model A arrive at time 0 and 8 queries for the same model arrive one millisecond afterwards. The current FIFO scheduler will simply execute these two batches sequentially as shown in Figure 1a: the second batch experiences a JCT of 149ms and the average JCT is 112ms.

Now, Figure 1b shows an alternative schedule which reduces the average JCT to 86ms for the all 16 queries. The idea behind it is to opportunistically merge the two batches of queries of the same model into a larger batch. It leverages a common property of DNNs—large batch size only marginally inflates the execution time (as illustrated in Table 1 which is consistent with benchmarked performance of many DNNs [2, 15]). For example, execution time of model A increases only by 10 ms when the batch size increases from 8 to 16. The reason is that small batches may not fully utilize all the cores in a GPU and a larger batch size can utilize the idle GPU resources without much latency increase.

In practice, batching is commonly used to speed up training of DNN (especially vision models) or inference over constant-rate data streams, both of which, however, have *pre-determined* batches. In contrast, Figure 1b illustrates that similar ideas, when properly used, can reduce inference delay of unpredictable workloads as well.

2.2 Online Job Stacking

So far, we have considered cases where there is one model running in each GPU at a time. Though it is common to run

inference tasks on separate GPUs, this can clearly lead to under-utilization of GPU resource, especially as more cameras are deployed and continuously generate video streams, many of which need to be analyzed at edge/cloud servers. This motivates our second scheduling strategy—stacking.

Before introducing stacking, it is useful to differentiate two types of utilization in GPU. First, GPU utilization is defined as the percentage of time one or more GPU kernels are running over the last second according to the common GPU performance analysis tool `nvidia-smi`[1]. If the utilization is under 100%, it means the workload cannot fully utilize the GPU and other workloads can still be added to the GPU, though the execution delay may raise. Second, memory utilization represents how much GPU memory has been used. Different from GPU utilization, if GPU memory is already used up, adding more workloads causes runtime crash.

Model	Batch Size	Latency (ms)	Avg. GPU (%)	Peak Memory (%) / On-Peak Time (ms)
C	10	200	70%	65/100
D	100	10	28%	35/5

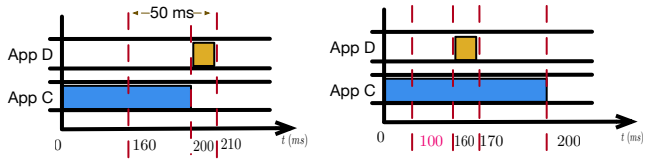
Table 2: Execution information of two models for the stacking example. Here "on-peak time" indicates the starting time of peak memory usage since the task execution.

Now consider two models, C and D, whose execution information is listed in Table 2, including the average GPU utilization and peak memory utilization of the model during execution. On-peak time is the duration that the model memory usage reaches its peak. As explained, we cannot overuse GPU memory. We assume that model C submits 10 queries at time 0 and model D submits 100 queries in 160 ms. SLOs (deadlines) for both applications are 250 ms. Existing schedulers run these two tasks sequentially (i.e., one GPU can only run on task at a time) and as a result, model D's queries must wait for four times the execution delay of themselves (as depicted in Figure 2a).

In contrast, we propose to stack these two tasks for concurrent execution on the same GPU. We find from Table 2 that although the sum of peak memory usage for both models is 100%, but their peaks happen at different times, which indicates that stacking them does not lead to runtime crash. Their combined GPU utilization is also below 100%. The new scheduling result with stacking is shown in Figure 2b where the average JCT drops to 27.3 ms.

Many DL frameworks such as Tensorflow already support running multiple models concurrently on one GPU and some prior work also seek to optimally pack tasks to better utilize memory space. What is new in Figure 2b is that online inference tasks can also benefit from being dynamically stacking

[1]https://nvidia.custhelp.com/app/answers/detail/a_id/3751/~/useful-nvidia-smi-queries



(a) Sequential Execution **(b) Stacking**

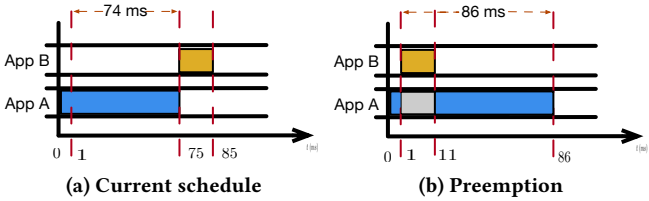
Figure 2: An illustrative example where stacking inference tasks in realtime leads to better GPU utilization.

to improve GPU utilization, especially when long running tasks are interleaved with short ad-hoc queries.

2.3 Dynamic Job Preemption

Finally, most of existing inference schedulers focus on maximizing average throughput, so they do not proactively preempt currently running tasks. This can sometimes cause resource wastage and even reduce overall throughput. On the other hand, preemption is known to improve JCT and query response time, since in theory it allows theoretically optimal policies (e.g., the shortest job first) in the online setting.

Let us consider the two inference applications, one using model A and one using model B (whose properties are given in Table 1), and both have an SLO (deadline) of 200 ms. We assume that 8 queries of model A arrive at the very beginning and then 256 queries pf model B arrive one millisecond after that (as depicted in Figure 3a). FIFO schedulers will run A's queries first then B's queries, leading to 85ms JCT for model B and an average JCT of 84.69ms across all queries. Since queries of model B only take 10ms to finish, if we allow preemption as shown in Figure 3b where B's queries get executed as soon as they arrive, the overall average JCT now drops to 12.30ms. Note that throughput is slightly lowered with preemption though, since the makespan increases from 85ms to 86ms.



(a) Current schedule **(b) Preemption**

Figure 3: An illustrative example of online inference task arrivals where preemptive scheduling reduces average the inference delay.

3 DESIGN

We now present the preliminary design of Irina, which integrates the three complementary strategies introduced in the last section. Two immediate challenges must be resolved: First, the ability to gracefully terminate a DNN inference task and switch to another task is key to both preemptive scheduling and opportunistic batching, but how to effectively enable preemption in existing inference frameworks that do not consider preemption in the first place? Second, how to select the best strategy for scheduling inference tasks online? This section elaborates our initial results towards addressing these key questions.

3.1 Enabling Preemption

Existing ML frameworks do not allow preemption while executing an inference task. Though it seems straightforward to stop a running task in GPU, it is tricky to do it safely and gracefully. Generally, an inference task is started by a CPU thread on the host machine. Killing the CPU thread is the simplest way to stop the running task. For example, code blocks 1 and 2 describe sample code to kill or cancel the thread that launches the inference task. However, both pthread APIs assume that there are functions to process the SIGQUIT signal with `pthread.kill` or cancellation points in the thread function for `pthread.cancel`, which requires users to modify their inference code. If there is no functions to process SIGQUIT signal, the signal may change to SIGKILL which kills the process and causes an abnormal exit. If there is no cancellation points in the thread function, `pthread.cancel` cannot perform and just wait for the thread function to finish. Moreover, during inference, the framework usually dynamically allocates the GPU memory which needs to be taken care of upon preemption.

```
// Create thread p1 to execute inference tasks
pthread_create(&p1, NULL, execute_fn, (void*)&tsk_queue);
pthreak_kill (p1, SIGQUIT);
// Insert a new task to the queue
tsk_queue. insert (new_task, 0);
// Create a new thread to execute inference tasks
pthread_create(&p2, NULL, execute_fn, (void*)&tsk_queue);
```

Listing 1: Preemption by killing the CPU thread.

```
// Create thread p1 to execute inference tasks
pthread_create(&p1, NULL, execute_fn, (void*)&tsk_queue);
pthread_cancel(p1);
pthread_join(p1, NULL);
// Insert a new task to the queue
tsk_queue. insert (new_task, 0);
// Create a new thread to execute inference tasks
pthread_create(&p2, NULL, execute_fn, (void*)&tsk_queue);
```

Listing 2: Preemption by canceling the CPU thread.

Alternatively one can kill the inference process on the CPU to avoid the abnormal exit, but CUDA is not safe across multiple processes as it does not support fork [7]. Thus both solutions are inadequate to address the challenges.

We aim to design a general preemption mechanism that can be applied to different ML frameworks while enabling graceful termination of the running task on GPU. To this end, we choose to work with the *dataflow graph* of the inference model. The dataflow graph is a direct acyclic graph (DAG) that describes the execution and data dependency of the operations (e.g. conv2d, relu, etc.) of an ML model. It is used in all ML frameworks as an execution blueprint of the ML training and inference pipelines [18] We add a new operation called *exit* to the dataflow graph to enable preemption. The exit operation maintains the dynamic GPU memory allocation information of the current task and safely frees the memory upon receiving the preemption signal. In addition, it returns a specially result to the ML framework to indicate that the task finishes in order to prevent it from crashing. This is because the function in the main process sometimes waits for a return value even when the inference thread is terminated. Irina re-writes the dataflow graph by inserting the exit operation between any two consecutive operations, which works across all ML frameworks. If the scheduler wants to tell the running task to stop, Irina will send a signal to the exit operations in the modified dataflow graph. The next immediate exit operation will then receive the signal and execute the exit function.

3.2 Scheduling Policy

The second design challenge is to devise an online scheduling policy that minimizes the JCT under unpredictable task arrivals. As can be seen from the examples in §2, each strategy's performance gains depend on the task arrival patterns and the running task's progress and it is not clear which strategy is best. Although stacking seems to be the best strategy as long as it is feasible since it does not affect the ongoing tasks, this is not always true in online DNN inference. For example, the new task may have to wait until the ongoing task's peak usage is over to avoid overusing GPU memory, which can slow down the average completion time.

To optimally leverage the three strategies, Irina's scheduler first computes for each new task the best-case schedule and average JCT under each strategy and then chooses the strategy with the smallest average JCT as the final result. In the following, we present the scheduling policies for each of the three strategies.

Preemption scheduling: We begin with the preemption algorithm in Irina. The scheduler should first retrieve the current scheduling information from different backends. We assume the models are all loaded in advance, so there is no

Input	Description
i	an inference task
B_i	batch size of task i
M_i	DNN model of task i
$L_i(\cdot)$	execution latency of task i given batch size B_i
$R_i(\cdot)$	resource requirement of task i given batch size B_i
G_n	remaining resources on GPU n

Table 3: Notations for the Irina's scheduler.

additional delay to load a model. (This is possible, because usually the DNN models are much smaller than the runtime-generated intermediate data, but our scheduler can be extended to account for the additional model-loading delay.) Our scheduler still processes inference tasks according to their arrival order. For a given task i, the scheduler requires information as listed in Table 3, especially its execution latency and the resource requirement both as functions of the batch size. This information can be accurately obtained from offline profiling since it largely depends on the GPU hardware and the hyperparameters. The scheduler selects GPUs which have the required model M_i preloaded into their memory and have enough resources, as candidates for preemption. For each candidate GPU, the scheduler computes the average JCT by assuming that task i preempts the running task on this GPU, as long as preemption does not violate the SLO (deadline) of both tasks. Lastly, among the candidate GPUs where preemption improves average JCT upon simple FIFO, our scheduler assigns task i to the one with the smallest average JCT.

Batching scheduling: To see if the given task i should be batched with an ongoing task, Irina first identifies all GPUs that are running tasks of the same DNN model M_i and filters them with the total resource requirement of the merged batch size according to $R_i(\cdot)$. Among all candidate GPUs, the scheduler computes the average JCT of batching task i with their current task, removes them if SLO is violated by the hypothetical batching, and selects the schedule with the smallest average JCT as the merging target.

Stacking scheduling: Stacking requires two models to run concurrently and safely on one GPU. Recall that the (peak) resource requirement and the timestamp information is contained in $R_i(B_i)$. Because stacking does not affect the JCT of the running task, the scheduling policy can be simpler. First, the scheduler identifies busy GPUs whose resource capacity can support the combined resource requirements of its running task and the new task i. Note that the timestamp information has to be taken into account to determine the actual peak requirement of the combined workload. Then, it computes the average JCT when stacking i to each candidate GPU, and assigns i to the one with the smallest JCT.

4 SIMULATION RESULTS

In this preliminary work, we conduct numerical simulation to evaluate the performance of Irina. We choose the speedup of the average JCT as the performance metric. There is a little difference for average JCT in DNN inference:

$$Average\ JCT = \frac{\sum_{i=1}^{n}(T_i - t_i) \times B_i}{\sum_{i=1}^{n} B_i}$$

Here t_i and T_i represent task i's arrival time and finish time, respectively, and B_i is the batch size. We use the number of samples processed in each inference task because each request is generated by a unique user. The average JCT thus reflects user's perceived latency on average.

In order to collect real task execution data, we adopt a server with 32-core CPUs, 64GB RAM, and NVIDIA RTX 2080 GPU with 8GB RAM. The server runs Ubuntu 18.04, Nvidia driver version 440.33, CUDA 10.2, cuDNN6.5 and libTorch 1.3.1. We use PyTorch to training the image classification models with ImageNet and TorchScript as the DNN inference framework. We use common models: AlexNet [16], VGG-16 [22], YOLOv3 [19], GoogLeNet [23], and obtain their inference execution information. Figure 4 shows the execution latency for the models with different batch sizes. Both AlexNet and GoogLeNet are indifferent to the batch size: there is little difference in execution latency when the batch size increases from 1 to 32. AlexNet, in particular, adopts the largest batch size while maintaining the lowest execution cost, which indicates that it is suitable for preemption and stacking. In contrast, YOLOv3 and VGG-16 appear more complex. With a batch size of 16 for example, YOLOv3 suffers the longest execution latency in the table.

We select some data points from our measurements for our simulation as summarized in Table 4. To simplify, we assume that each inference task runs on the same GPU. Generally speaking, the online inference workloads contain regular and ad-hoc requests. We choose YOLOv3 as the DNN model for the predictable regular requests since object detection is used in various monitoring systems that generates queries with fixed intervals. We assign AlexNet, GoogLeNet, and VGG-16 for the on-demand unpredictable queries since they are used to identify say vehicle models, animal types, or other ad-hoc purposes. Queries arrive dynamically over time and each query appears as a regular query with a probability we control. If a query is assigned to be ad-hoc it is uniformly assigned to one of the three CNN models. We implement the default scheduling policy in TenserFlow Serving as our baseline. TensorFlow Serving aims to maximize the throughput via large batching, so it waits for as many requests of the same model as possible under SLOs for scheduling.

Figure 5 shows the simulation results. We find the performance of Irina varies depending on the probability of having ad-hoc tasks. Comparing to baseline, Irina achieves

Model Name	Batch Size	Latency(ms)	Memory Utilization
YOLOv3	8	68	30%
	16	117	55%
VGG-16	8	22	55%
	16	33	86%
GoogleNet	32	14	46%
AlexNet	128	9	16%

Table 4: Models and their execution information used in the simulation.

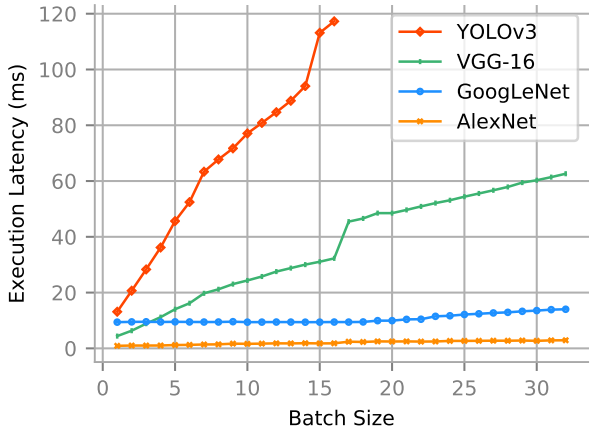


Figure 4: The execution latency for the selected models measured on our testbed.

1.3x − 2.5x speedup on average task completion time. When ad-hoc tasks are unlikely to arrive, there is not many scheduling opportunities for Irina and it can only deliver moderate gains. Baseline also performs well in these cases. On the other hand, when the ad-hoc tasks appear more frequently, Irina has ample opportunities to apply the three scheduling strategies.

5 RELATED WORK

Clipper [6] and Tensorflow Serving [4] are two popular serving systems. Tensorflow Serving is designed to serve the DNN models trained by Tensorflow on CPU and GPU. It provides the complete service stack for DNN inference. Tensorflow Serving maximizes system throughput via batching [8]. Clipper can be viewed as an extension of Tensorflow Serving. It simplifies model deployment with a modular architecture and employs techniques like caching to optimize performance. However, neither system focuses on inference task scheduling.

Nexus [21], InferLine [28], and Nanily [24] are recent works focusing on large scale distributed DNN inference. Nanily proposes adaptive batching to schedule the requests

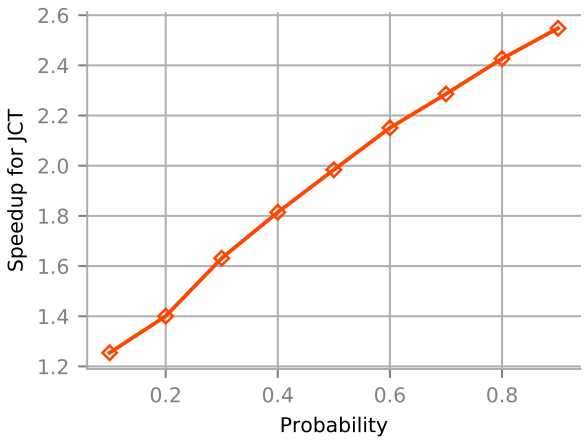


Figure 5: Irina's speedup in average JCT.

under SLOs and pre-schedules the predictable requests. Nexus focuses on DNN inference on video analysis and assume the workloads are stable in each epoch. It exploits the large batch execution on the predictable requests to improve the throughput. MainStream [13] proposes to re-use the same sub-models in different requests in order to enlarge batch size and improve the inference throughput. InferLine focuses on scheduling the complex execution pipelines which may contain multiple models. It exploits the different configurations for the models in the complex execution pipelines and re-uses some base models to reduce the latency and improve the throughput. Although these systems can schedule the recurring tasks efficiently, when it comes to the unpredictable ad-hoc queries they all rely on FIFO which does not work well for completion time. Salus [26] modifies the internal execution engine of TensorFlow to maximize the number of models concurrently running on GPU. Olympian [10] aims to schedule multiple inference models fair sharing the single GPU and reduce the makespan. Some works [11, 12] proposes space-time multiplexing to reduce the latency when the same models execute concurrently in a GPU. INFaas [20] provides different configurations for different batch sizes, hardwares and accuracies, which provides an easy-to-use interface for the user.

6 DISCUSSION

We discuss several concerns one may have about Irina.

Overhead of adding the exit op. In our design §3.1, Irina only modifies the dataflow graph to add the *exit* op ahead-of-time. To keep implementation simple, we insert exit ops only between two consecutive layers. In general, most models contain tens of layers, so adding exit op brings little overhead on execution time. When there is a more complex model with thousands of layers [9], inserting thousands of exit ops

becomes inefficient. One can combine multiple small layers into a group and add the exit Op at the group level to reduce the overhead.

Availability of task execution information. Irina has focused on scheduling inference tasks with unpredictable arrivals. However, the models used by the workloads are known in advance by the applications and to the scheduler. Irina can then profile each model offline using the corresponding backend runtimes and different hardware, and store the static and dynamic information in a metadata store. Static metadata maintains the architecture, framework, accuracy, and the preprocessing method for a given application. Dynamic metadata includes the relationship across hardware, batch sizes, and execution costs, including GPU core utilization, memory utilization, and execution latency. Irina estimates the average job completion time using the empirical model execution cost and then schedules the new coming tasks.

Interference among concurrent GPU models. Stacking is used to saturate GPU resource. However, some work [11, 12] shows the interference among the concurrently running models on a GPU may slow the execution, and may affect the stacking performance. In this work, to simplify the motivation and simulation, we do not consider this effect. Different from [11, 12], we only consider a long-running and a short task as the potential stacking candidates. The short task can finish much faster compared to the long task and the interference effect is minimal in this sense. In addition, Irina's model pool holds all the models which will be used in the execution. We can profile the concurrently running models offline to explicitly take into account the interference effect in JCT.

The intermediate data during execution. As Irina preempts a running task, there are intermediate computation results from executing the DNN model in the GPU. One may choose to save and reuse them when this task is re-executed. However, this solution requires redesigning the data loading method before launching the task in current frameworks. Further, saving the intermediate results in GPU occupies limited GPU memory. If we swap them to the host memory, the extra delay of swapping out and in may even be worse than simply re-computing everything from scratch as some work already showed [5, 25]. Therefore, here we simply discard all the intermediate results when preempting a task.

7 CONCLUSION AND FUTURE WORK

In this work, we have presented the preliminary design of Irina, an efficient online scheduling system for DNN inference workloads. Irina aims to reduce the average inference latency via three key strategies that were previously overlooked in inference scheduling: (1) Preemption that schedules the small task which is easy to be blocked by other tasks in existing schedulers; (2) Batching which utilizes the large batch execution for queries arriving at different times for the same model; and (3) Stacking which runs queries of more than one model on the same GPU to improve the utilization and reduce the queueing time. We performed simulation studies and demonstrate with empirical data that Irina reduces the average JCT by 1.3x – 2.5x compared to the default scheduler in TensorFlow Serving. We are now implementing a complete prototype of Irina and will conduct comprehensive testbed experiments to assess Irina's full potential.

REFERENCES

[1] 11 reasons cloud video surveillance is moving to the cloud. https://www.een.com/vsaas-video-surveillance-moving-to-cloud/.

[2] Cnn model inference benchmarks for some popular deep learning frameworks. https://github.com/nicklhy/DLInfBench/tree/master/results/.

[3] Mlperf inference v0.5 results. https://www.mlperf.org/inference-results/ November 6th, 2019.

[4] Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017), pp. 1387–1395.

[5] Beaumont, O., Herrmann, J., Pallez, G., and Shilova, A. Optimal memory-aware backpropagation of deep join networks. *Philosophical Transactions of the Royal Society A 378*, 2166 (2020), 20190049.

[6] Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)* (2017), pp. 613–627.

[7] Guide, D. Cuda c programming guide. *NVIDIA, July* (2013).

[8] Hanhirova, J., Kämäräinen, T., Seppälä, S., Siekkinen, M., Hirvisalo, V., and Ylä-Jääski, A. Latency and throughput characterization of convolutional neural networks for mobile computer vision. In *Proceedings of the 9th ACM Multimedia Systems Conference* (2018), pp. 204–215.

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

[10] Hu, Y., Rallapalli, S., Ko, B., and Govindan, R. Olympian: Scheduling gpu usage in a deep neural network model serving system. In *Proceedings of the 19th International Middleware Conference* (2018), pp. 53–65.

[11] Jain, P., Mo, X., Jain, A., Subbaraj, H., Durrani, R. S., Tumanov, A., Gonzalez, J., and Stoica, I. Dynamic space-time scheduling for gpu inference. *arXiv preprint arXiv:1901.00041* (2018).

[12] Jain, P., Mo, X., Jain, A., Tumanov, A., Gonzalez, J. E., and Stoica, I. The ooo vliw jit compiler for gpu inference. *arXiv preprint arXiv:1901.10008* (2019).

[13] Jiang, A. H., Wong, D. L.-K., Canel, C., Tang, L., Misra, I., Kaminsky, M., Kozuch, M. A., Pillai, P., Andersen, D. G., and Ganger, G. R. Mainstream: Dynamic stem-sharing for multi-tenant video processing. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)* (2018), pp. 29–42.

[14] Kang, D., Emmons, J., Abuzaid, F., Bailis, P., and Zaharia, M. Noscope: optimizing neural network queries over video at scale. *arXiv preprint arXiv:1703.02529* (2017).

[15] Kochura, Y., Gordienko, Y., Taran, V., Gordienko, N., Rokovyi, A., Alienin, O., and Stirenko, S. Batch size influence on performance of graphic and tensor processing units during training and inference phases. In *International Conference on Computer Science, Engineering and Education Applications* (2019), Springer, pp. 658–668.

[16] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

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

[18] Looks, M., Herreshoff, M., Hutchins, D., and Norvig, P. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181* (2017).

[19] Redmon, J., and Farhadi, A. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[20] Romero, F., Li, Q., Yadwadkar, N. J., and Kozyrakis, C. Infaas: A model-less inference serving system. *arXiv preprint arXiv:1905.13348* (2019).

[21] Shen, H., Jin, Y., Kong, B., Philipose, M., Krishnamurthy, A., and Sundaram, R. Nexus: A gpu cluster for accelerating neural networks for video analysis.

[22] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[23] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 1–9.

[24] Tang, X., Wang, P., Liu, Q., Wang, W., and Han, J. Nanily: A qos-aware scheduling for dnn inference workload in clouds. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)* (2019), IEEE, pp. 2395–2402.

[25] van de Leemput, S. C., Teuwen, J., and Manniesing, R. Memcnn: a framework for developing memory efficient deep invertible networks.

[26] Yu, P., and Chowdhury, M. Salus: Fine-grained gpu sharing primitives for deep learning applications. *arXiv preprint arXiv:1902.04610* (2019).

[27] Zhang, B., Jin, X., Ratnasamy, S., Wawrzynek, J., and Lee, E. A. Awstream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 236–252.

[28] Zumar, C. Inferline: Ml inference pipeline composition framework.