ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences

Mingcong Han[1,2], Hanze Zhang[1,4], Rong Chen[1,2], and Haibo Chen[1,3]

[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University [2]Shanghai AI Laboratory

[3]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

[4]MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, China

Abstract

Many intelligent applications like autonomous driving and virtual reality require running both latency-critical and best-effort DNN inference tasks to achieve both real time and work conserving on GPU. However, commodity GPUs lack efficient preemptive scheduling support and state-of-the-art approaches either have to monopolize GPU or let the real-time tasks to wait for best-effort tasks to complete, which causes low utilization or high latency, or both.

This paper presents REEF, the first GPU-accelerated DNN inference serving system that enables microsecond-scale kernel preemption and controlled concurrent execution in GPU scheduling. REEF is novel in two ways. First, based on the observation that DNN inference kernels as mostly idempotent, REEF devises a reset-based preemption scheme that launches a real-time kernel on the GPU by proactively killing and restoring best-effort kernels at microsecond-scale. Second, since DNN inference kernels have varied parallelism and predictable latency, REEF proposes a dynamic kernel padding mechanism that dynamically pads the real-time kernel with appropriate best-effort kernels to fully utilize the GPU with negligible overhead. Evaluation using a new DNN inference serving benchmark (DISB) with diverse workloads and a real-world trace on an AMD GPU shows that REEF only incurs less than 2% overhead in the end-to-end latency for real-time tasks but increases the overall throughput by up to 7.7×, compared to dedicating the GPU to real-time tasks. To demonstrate the feasibility of our approaches on closed-source GPUs, we further ported and evaluated a restricted version of REEF on an NVIDIA GPU with a reduction of the preemption latency by up to 12.3× (from 6.3×).

1 Introduction

Deep Neural Network (DNN) inference has been widely adopted by modern intelligent applications, such as autonomous driving [2, 37, 41, 80], virtual reality [58, 83], speech/image recognition [32, 75], and healthcare [19, 24], just to name a few. Many of them demand real-time inference serving in mission-critical tasks, where GPUs have emerged as a popular accelerator to serve DNN inferences [15, 33, 47, 89].

Although the low-latency demand of DNN inferences can be fulfilled by dedicating the whole GPU to sequentially serve requests from a single DNN application [10, 80, 91], it is hard to fully exploit the massive parallelism of the GPU [47]. Hence, it is a common practice to share a GPU among multiple applications with different timing constraints in emerging intelligent systems [41, 78], which can greatly improve overall throughput, as shown in Fig. 1(a). For example, autonomous vehicles use DNNs to recognize obstacles and traffic lights [9, 59], which are latency-critical tasks (called *real-time* tasks in this paper). Meanwhile, other tasks with no hard real-time requirement [78] (called *best-effort* tasks in this paper), such as monitoring human driver's emotion and fatigue, are also served within the GPU using DNNs [19, 48, 84].

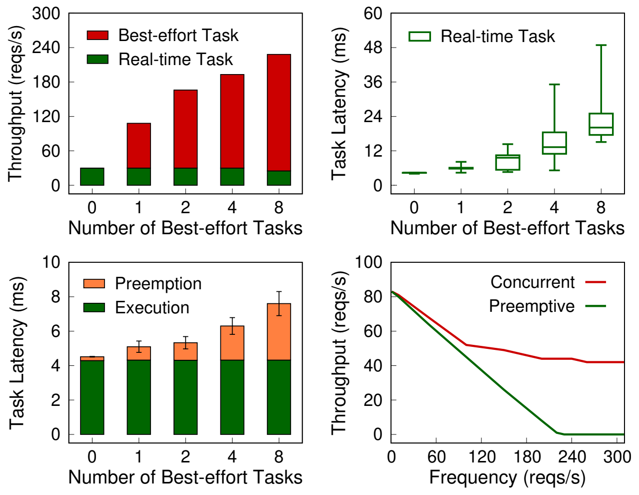

Fig. 1: *(a) The overall throughput of DNN inferences (both real-time and best-effort tasks) and (b) the end-to-end latency of real-time tasks when using concurrent GPU scheduling (i.e., multiple GPU streams [46, 49, 60]), (c) the end-to-end latency of real-time tasks when using preemptive GPU scheduling (i.e., wait-based preemption [12, 77, 90]), and (d) the throughput of best-effort tasks as the frequency of real-time tasks increases.* **Workload**: *VGG [68] (real-time) and ResNet [30] (best-effort).* **Testbed**: *one AMD Radeon Instinct MI50 GPU with 16 GB of memory (see §7 for details).*

Typically, DNN inferences have two potentially conflicting goals for GPU scheduling. First, the real-time tasks should be treated as first-class citizens on the GPU without interference from other tasks to achieve *low end-to-end latency*. Second, both real-time tasks and best-effort tasks should be served concurrently on the GPU to achieve high overall throughput (*work-conserving*).

State-of-the-art GPU libraries (e.g., CUDA [52] and ROCm [3]) commonly provide multiple GPU streams (e.g., CUDA Streams [60]) to concurrently execute multiple tasks on the same GPU. However, as shown in Fig. 1(b), although the end-to-end inference latency of real-time tasks is low (about 4 ms) and stable when monopolizing the GPU, the tail latency of real-time tasks significantly increases by over an order of magnitude (close to 50 ms) when running concurrently with best-effort tasks. This, unfortunately, is unacceptable for real-time scenarios [85].

Similar to operating systems using preemptive scheduling to provide real-time guarantees, an intuitive approach is to provide preemption for GPU scheduling, which is unfortunately missing in commodity GPUs [70]. Prior work [12, 77, 90] proposed a wait-based approach to passively waiting until the completion of running blocks, which may cause a preemption delay of several milliseconds. Although it may be sufficient for traditional GPU workloads, this approach is still far from optimal for DNN inference tasks since the preemption latency is non-trivial compared to the execution time of real-time inference tasks, as shown in Fig. 1(c). Further, when the real-time inference requests arrive at a high frequency (e.g., camera (120 reqs/s) [16] or multiple sensors [41]), the best-effort tasks may even get starved, as shown in Fig. 1(d).

This paper presents REEF, the first DNN inference serving system for commodity GPUs with microsecond-scale kernel preemption and controlled concurrent execution in GPU scheduling to achieve both *real time* and *work conserving*. Specifically, the arriving real-time task should instantly preempt the GPU from the running best-effort kernels without waiting for their completion. Meanwhile, the best-effort kernels should be executed concurrently by using GPU resources leftover from the real-time kernels.

A key insight of REEF is that each kernel in DNN inference is mostly *idempotent*. This implies that the running best-effort kernels can be proactively killed and restored without saving contexts. Based on this, REEF proposes a *reset-based preemption* scheme. To thoroughly flush hundreds of outstanding kernels in both GPU runtime and devices, REEF designs different approaches to resetting different software queues and retrofits the GPU driver to exactly use existing hardware mechanisms to reset compute units while preserving device memory of the GPU. It can improve both kernel preemption and restore. Therefore, REEF can launch a real-time task on the GPU in tens of microseconds, regardless of the number of preempted kernels and their execution time.

REEF further proposes a *dynamic kernel padding* mechanism based on the observation that the execution time of GPU kernels in DNN inferences is deterministic and *predictable*. This implies that the pending best-effort kernels can be carefully selected to pad the real-time kernel without performance interference, based on offline profiling in advance. REEF extended GPU compiler to construct a template of padded kernels by using function pointers. Further, to eliminate the overhead of indirect function calls on the GPU, REEF introduces proxy kernels to address register allocation problem and avoid unnecessary context saving at runtime. Therefore, REEF can concurrently execute the real-time task with best-effort tasks at the expense of negligible performance and memory overhead (less than 1% and about 10 KB).

We have implemented **REEF** by extending Apache TVM [73] (a compiler for deep learning) and AMD ROCm [3] (an open-source GPU computing platform). We evaluate REEF using a new DNN Inference Serving Benchmark (DISB) with diverse workloads and models, as well as a real-world trace from Apollo [7] (an open autonomous driving platform). Our experimental results show that REEF only incurs less than 2% of the end-to-end latency overhead for real-time tasks but increases overall throughput by up to 4.3×, compared to dedicating the GPU to real-time tasks. Our approach further reduces the preemption latency by over one order of magnitude against the state-of-the-art, less than 40 microseconds for all models. To demonstrate the feasibility of our approaches on closed-source GPUs, we further ported and evaluated a restricted version of REEF on an NVIDIA GPU with a reduction of the preemption latency by up to 12.3× (from 6.3×).

Contributions. We summarize our contributions as follows.

- An in-depth understanding on the characteristics of GPU-accelerated DNN inferences such as idempotence and the issues of state-of-the-art GPU scheduling schemes (§2).
- A new reset-based preemption scheme that can launch a real-time kernel on the GPU in a few microseconds, regardless of the number of preempted kernels (§4).
- An elegant mechanism that can dynamically pad the real-time kernel with best-effort kernels to fully exploit the massive parallelism of the GPU (§5).
- An implementation (§6) on both AMD and NVIDIA GPUs and an evaluation that demonstrates the advantage and efficacy of REEF over state-of-the-art (§7).

The source code of REEF is publicly available at `https://github.com/SJTU-IPADS/reef`. The DNN Inference Serving Benchmark (DISB) framework can be obtained separately from `https://github.com/SJTU-IPADS/disb`.

2 Background and Motivation

2.1 Characterizing GPU-Accelerated DNN Inference

Deep neural network (DNN) comprises multiple instances of versatile layers, such as convolutional, pooling and fully-connected layers. GPUs have been widely exploited to accelerate DNN inference serving [20, 28, 64]. To serve inference requests on GPUs, the pre-trained DNN model (e.g., ResNet [30]) is loaded into GPU memory ahead of time. Fig. 2 outlines the implementation of GPU-accelerated DNN inference. For each arriving request, all kernels of the DNN

```
# device codes
__global__ void conv_relu(in, weight, out):
  sum = 0;
  for i in range(0,3)
     for j in range(0,3)
        sum += in[..] × weight[..]
  out[..] = ReLU(sum)
__global__ void dense(in, weight, bias, out):
  sum = 0;
  for i in range(0,512)
     sum += in[..] × weight[..]
  out[..] = sum + bias[..]

# host codes
void inference(...):
  memcpyH2D(in, in_host, in_sz)  # copy in to GPU
  conv_relu <<<dim(32), ..>>> (in, .., buf_conv)
  ...  # launch other kernels
  pooling <<<dim(64), ..>>> (.., buf_pool)
  dense <<<dim(10), ..>>> (buf_pool, .., buf_dense)
  softmax <<<dim(1), ..>>> (buf_dense, .., out)
  memcpyD2H(out_host, out, out_sz)  # copy out to CPU
```

Fig. 2: *An example of DNN inference using a model like ResNet.*

model are executed in turn with the input, and the resulting output is returned to the DNN application.

DNN inference is now used by both *real-time* (RT) tasks, such as obstacle and traffic lights recognition [9, 59], and *best-effort* (BE) tasks, such as emotion and fatigue monitoring [19, 48, 84]. The real-time tasks are latency-critical, because violating the end-to-end latency requirement may cause system failures or even safety problems. In addition, such requests are usually issued periodically at various frequencies by input sensors (e.g., camera and LiDAR [7, 41]). On the contrary, the best-effort tasks have no hard timing requirement, but are repetitively executed in the background.

Idempotence. The GPU-accelerated DNN model for inference tasks consists of a sequence of kernels, which implement one or several DNN layers. We observe that GPU kernels in DNN models are mostly *idempotent* as they consist of almost only dense linear algebra computations without side effects.[1] Hence, the kernel can always produce the same output with the same input no matter it has been retried or not. Meanwhile, in the DNN model, the (k)-th kernel always uses the outputs of the (k-1)-th kernel and static arguments (e.g., weight) as inputs, e.g., `conv_relu` and `dense` kernel in Fig. 2. Therefore, the execution of DNN inference task can be restored from any kernel before the interrupted kernel and will not change the inference results.

Massive kernels. Unlike traditional GPU applications that only contain a few kernels (e.g., at most 14 kernels in Rodinia [11]), it is common to see hundreds of kernels in modern DNN models (see Table 1). In response, large amounts of kernels—usually hundreds or more—would be submitted in

[1]We validated using our tool that all 320 GPU kernels of the 11 DNN models from Apache TVM's test suite [72] are idempotent.

Table 1: *The amount of GPU kernels in DNN models evaluated in §7 and the execution time (in millisecond). The codes are generated by TVM [15] and run on AMD Radeon Instinct MI50 GPU.*

Model	**ResNet**	**DenseNet**	**VGG**	**Inception**	**Bert**
#Kernels	307	207	55	146	205
Exec. Time	13.6	3.5	4.4	8.3	5.4

advance to hide the lengthy kernel launching time. Further, to fully exploit the GPU, the serving system may concurrently execute multiple kernels from different inference tasks using the same or different DNN models. Therefore, the performance penalty of preempting the GPU would be significant (a few milliseconds) and even comparable to the execution time of hundreds of kernels.

Latency predictability. We observe that the execution time of GPU kernels in DNN inferences is deterministic and predictable when running individually on the GPU (no interference). The reasons are two-fold. First, the kernel is mostly linear algebra computations such as matrix multiplication and convolution, which contains neither conditional branches nor inconstant loops. Second, all kernel arguments (e.g., input and weights) and the output are fixed-size arrays. Therefore, the execution time of such kernels is independent of the input of inference request and can be measured and accurately predicted in advance. In practice, we observe that the variance in kernel execution time of DNN models is typically only a few microseconds (see Fig. 3(a)). This is also confirmed in recent literature [6, 28, 47].

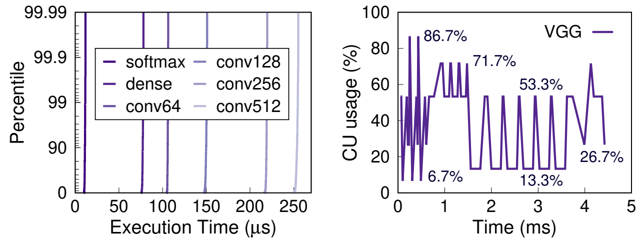

Fig. 3: *(a) The CDF of execution time of several typical kernels in VGG, and (b) the timeline of CU usage during VGG execution on a GPU with 60 CUs. Note that the execution time of GPU kernels in VGG covers a fairly wide range from 10 µs to 255 µs (see Fig. 10).*

Varied parallelism. The GPU kernels in DNN inferences usually exhibit completely different parallelism due to varied input scales. For example, as shown in Fig. 2, the `pooling` kernel uses 64 thread blocks, while the `softmax` kernel just uses 1 thread block. Consequently, the computational demand for DNN inferences, namely the number of compute units (CUs), is ever-changing during the execution. As an example, Fig. 3(b) shows the CU usage during VGG execution varies between 6.7% and 86.7%. Therefore, to efficiently exploit the GPU, it is indispensable to leverage a dynamic mechanism to select and execute multiple kernels from different DNN inference tasks at runtime.

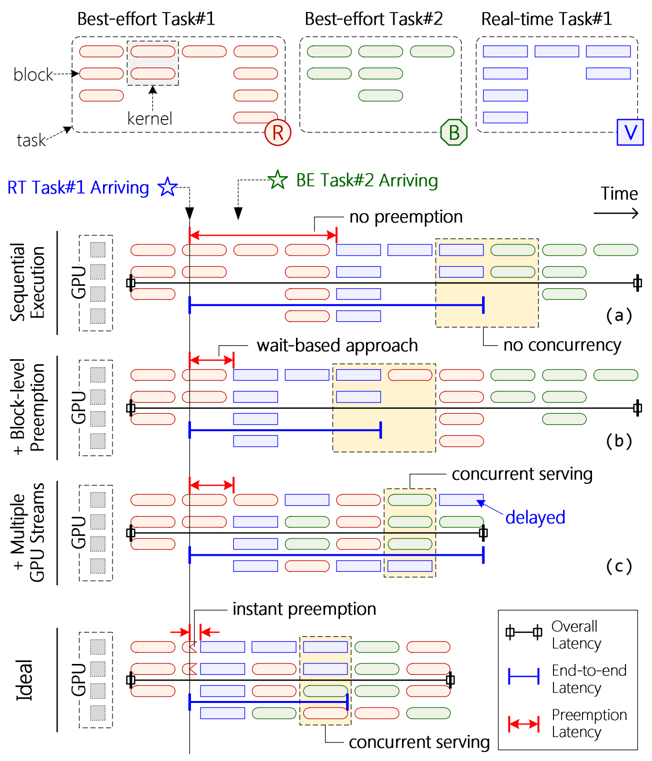

Fig. 4: *An example of GPU task scheduling with different kernel preemption and parallelism schemes for a hybrid workload, which contains two best-effort and one real-time DNN inference tasks. The GPU has four compute units (CUs).*

2.2 State-of-the-art GPU Scheduling

As stated before, DNN inference serving system relies on GPU scheduling to meet two potentially conflicting performance goals: low latency and work conserving. Although GPU scheduling has been widely studied in the HPC community [1, 8, 12, 13, 27, 43, 77, 82, 88], the unique characteristics of DNN inferences and the two performance goals introduce new challenges for GPU scheduling. We review the state-of-the-art schemes of GPU scheduling and discuss the performance issues when serving various DNN inferences through a brief example, as shown in Fig. 4.

Sequential execution. Most existing DNN serving systems, such as Clockwork [28], use sequential execution to avoid interferences among tasks. Thus, each task can achieve optimal execution latency, as shown in Fig. 4(a). However, the end-to-end latency of RT tasks might be significantly extended due to lengthy preemption latency (red dimension line), since it has to wait for the completion of previous tasks (*no preemption*). Further, this scheme has a poor overall throughput, due to sequentially serving inference tasks (i.e., *no concurrency*).

Block-level preemption. To reduce end-to-end latency for real-time tasks, it is necessary to preempt the GPU from running best-effort tasks. However, it is difficult to implement preemptive scheduling on the GPU due to the large context (e.g., a large amount of registers) [56, 70]. Meanwhile, commodity GPUs also lack hardware support for the preemption mechanism.[2] As a compromise, prior work [8, 90] proposes wait-based approaches to implementing block-level preemption for GPU scheduling. The real-time task still needs to passively wait until the completion of running blocks, as shown in Fig. 4(b). Further, the preemption latency will increase with the number of preempted kernels (see Fig. 1(c)). As a compromise, prior work [8, 90] has to limit the number of kernels submitted to the GPU, which is impractical for DNN inferences. Further, a high-frequency real-time task will break the execution of best-effort tasks, even leading to starvation (see Fig. 1(d)).

Multiple GPU streams. To improve overall throughput, modern GPU libraries (e.g., CUDA [52] and ROCm [3]) commonly provide multiple GPU streams (e.g., CUDA Streams [60]) to concurrently execute kernels from independent tasks. The runtime scheduler dispatches kernels from GPU streams on demand to keep all compute units (CUs) busy, as shown in Fig. 4(c). Although leveraging multiple GPU streams can improve throughput (see Fig. 1(a)), the latency of real-time tasks can be significantly degraded by concurrent tasks, e.g., the last kernel of RT Task#1 in Fig. 4(c). Even worse, the latency overhead will increase with the number of concurrent tasks (see Fig. 1(b)).

3 REEF Overview

3.1 System Architecture

The goal of REEF is to provide preemptive GPU scheduling to achieve real time for latency-critical tasks and work conserving for best-effort tasks (see ideal scheduling for the example in Fig. 4). Based on the insight that DNN inference kernels are mostly idempotent and there are a massive number of kernels with varied parallelism and predictable latency, REEF provides two novel designs called reset-based preemption and dynamic kernel padding.

Fig. 5 illustrates an overview of REEF's architecture. REEF consists of (a) an offline part, which compiles and loads user-provided DNN models, and (b) an online part, which schedules and serves DNN inference requests.

DNN model preparation (offline). Typically, DNN models are first compiled and optimized for accelerator back-ends (e.g., GPU) and then loaded into the model pool. Inspired by prior work [12, 36, 77], REEF extends the model compiler (e.g., TVM [15]) with a *code transformer* module, which first validates the idempotence of kernels in DNN models and then transforms the source code to assist GPU scheduling in REEF. Moreover, REEF develops a *kernel profiler* to measure the computational requirements and the execution time for each kernel of the model, which is accurate and practical for DNN models (see §2).

[2] Although NVIDIA claims that their GPUs have been equipped with preemption support since Pascal architecture [51], there is no public available information or a software controllable interface [12, 39, 77].

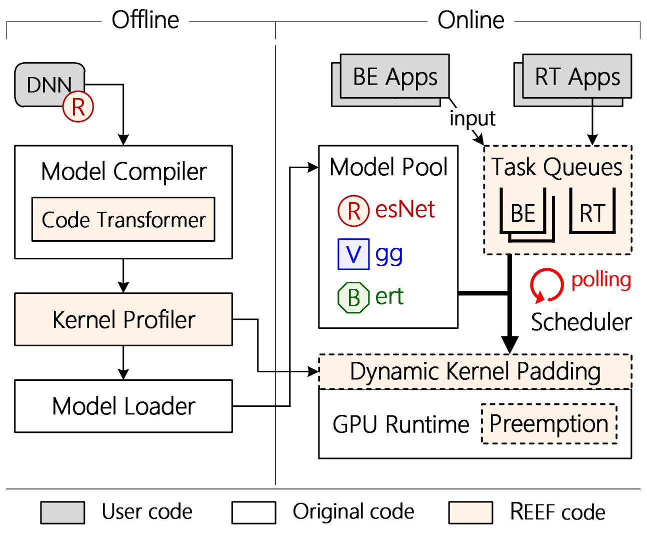

Fig. 5: *Architecture of* REEF. *Modules in boxes with dashed border are on the critical path of serving DNN inference requests. Other modules do not directly impact serving latency and throughput.*

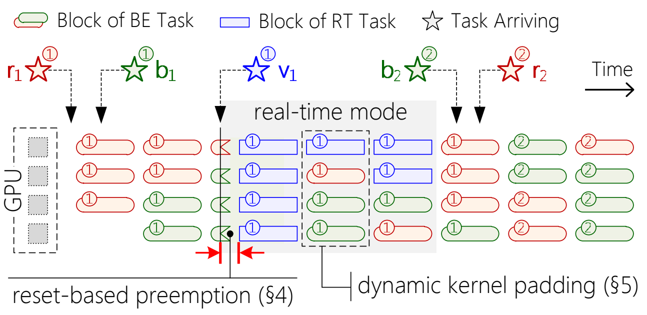

Fig. 6: *An example of timeline in* REEF. *The DNN inference tasks here are the same as that in Fig. 4.*

DNN inference serving (online). REEF extends a state-of-the-art GPU runtime (e.g., ROCm [3]) with four major components for DNN inference serving.

Task Queues. REEF maintains one real-time task queue and several best-effort task queues. Each queue is bound to a GPU stream for launching GPU kernels, where inference requests are served in a FIFO order. For simplicity, REEF executes real-time requests one at a time. Note that any scheduling policy that treats the whole GPU as a single device, such as EDF [10], can be adopted by REEF for real-time requests. Further, REEF offers an RPC-based interface for DNN-based applications to deliver inference requests to task queues.

Scheduler. The scheduler in REEF uses busy polling on task queues and assigns tasks to the associated GPU streams. Corresponding to whether there are real-time tasks, REEF provides two execution modes, namely *real-time* mode and *normal* mode. The scheduler will switch from normal mode to real-time mode when encountering real-time tasks, and switch back to normal mode when the real-time task queue is empty.

Preemption module. In normal mode, REEF concurrently serves best-effort tasks from different task queues using multiple GPU streams [3, 60] provided by GPU runtime. In real-time mode, REEF first uses the preemption module to instantly preempt the GPU from all running best-effort tasks (§4) and then launches the real-time task on the GPU immediately.

Dynamic kernel padding (DKP). In real-time mode, before launching a real-time kernel, the DKP module will select appropriate best-effort kernels and dynamically pad them to the real-time kernel (§5). REEF will execute the padded kernel on the GPU to achieve high throughput. Note that the best-effort kernels will only use GPU resources leftover from the real-time kernel.

3.2 An Illustrative Example

Fig. 6 illustrates the timeline of scheduling five DNN inference tasks in REEF. Upon receiving the first two best-effort requests r_1 and b_1, REEF runs in normal mode, and the kernels of two different tasks are scheduled to two different GPU streams. The GPU runtime will concurrently execute the kernels on the GPU. While r_1 and b_1 execute, a real-time request v_1 arrives. The scheduler immediately switches to real-time mode, and GPU runtime instantly preempts the GPU by killing all running kernels of best-effort tasks (i.e., r_1 and b_1). Meanwhile, the DKP module selects appropriate kernels from restored tasks to dynamically pad the kernels of real-time task v_1. After that, the padded kernel will be executed on the GPU alone. While v_1 is completed, the scheduler switches back to normal mode. All running and later best-effort tasks (i.e., r_1, b_1, b_2, and r_2) will concurrently execute on the GPU through two GPU streams.

4 Reset-based Preemption

The key insight behind our idea, namely reset-based preemption, is that the GPU kernels in DNN models are mostly *idempotent*, which enables *proactive* preemption—killing all running kernels on the GPU immediately and restoring them later. The benefits are two-fold. First, it avoids saving and restoring the large context of the GPU (e.g., a 256 KB register file per CU) [70]. Second, there is no need to wait for all running kernels to complete, which can take hundreds of microseconds.

However, there are still new challenges before making our reset-based preemption come true on commodity GPUs. Except for the kernels running on the GPU, hundreds of launched kernels are buffered in multiple queues maintained by GPU runtime. This is necessary to hide the kernel launch time and fully exploit the massive parallelism of GPU. Whereas, evicting all launched kernels makes it indeed difficult to preempt the GPU in tens of microseconds.

Fig. 7 illustrates the lifetime of launched kernels in the GPU runtime and devices. First, the scheduler launches all kernels of an inference task and specifies a GPU stream for each task. The GPU runtime maintains a linked list, called

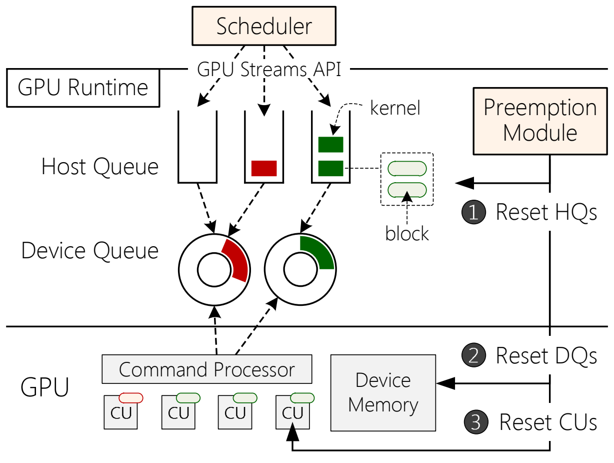

Fig. 7: *Extended GPU runtime in* REEF *for instant preemption.*

host queue, for each GPU stream to buffer launched kernels. Each host queue has a background thread that transmits the buffered kernels asynchronously to a ring buffer, called *device queue*, which is accessed by CPU and GPU simultaneously. The command processor of GPU will poll all device queues to fetch the buffered kernels and eventually dispatches them to *compute units*. Therefore, launched kernels of an inference task may exist in three places, namely host queues (HQs), device queues (DQs), and compute units (CUs). To achieve instant preemption, kernels in all three places must be evicted.

4.1 Evicting Buffered Kernels

The reset-based approach requires proactively evicting all buffered kernels from both host queues and device queues. For host queues, it is straightforward to reset them (❶ in Fig. 7), dequeuing all buffered kernels and reclaiming memory, as they are fully controlled by the GPU runtime. For device queues, however, the GPU runtime cannot evict buffered kernels from device queues, because the command processor of GPU can directly fetch kernels from device queues [23], resulting in data races and unpredictable results. In addition, the CPU also does not provide a way to safely evict kernels from device queues. A potential solution is to notify the GPU to re-register a new device queue [62]. However, it would incur an unacceptable latency overhead (e.g., about 1 ms on our testbed).

Inspired by evictable kernels [12], we propose *lazy eviction* to reset device queues without extending GPU runtime and hardware. The code transformer of REEF injects a piece of code at the beginning of each kernel in advance, which checks the *preemption flag* to realize whether it has been evicted. When the preemption flag is true, the kernel will voluntarily terminate itself. Therefore, when a preemption occurs, the preemption module will immediately set the preemption flag to true in GPU memory (see ❷ in Fig. 7). The kernels buffered in device queues will be fetched and dispatched to the CUs as usual, but will terminate themselves immediately.

Our initial queue eviction mechanism imposes a non-trivial overhead on the preemption process, taking more than 500 µs to preempt a single task (see §7.3). An in-depth analysis shows that the overhead comes mainly from (a) reclaiming memory from the host queue and (b) waiting to fetch kernels from the device queue. Therefore, we propose two optimizations to mitigate overheads.

Asynchronous memory reclamation. The preemption latency is proportional to the host queue length when using synchronous memory reclamation for evicted kernels in the host queues. Therefore, the performance penalty of preempting a DNN inference task would be significant, since it requires buffering hundreds of kernels in the host queue. To instantly evict GPU kernels from the host queue, REEF leverages a background GC thread to reclaim memory asynchronously. Specifically, REEF resets the host queue by simply nullifying the head pointer first and then notifying the GC thread to reclaim memory in the background.

Device queue capacity restriction. Although using lazy eviction can terminate kernels in the device queue immediately at the beginning of execution, the kernels still have to be fetched and dispatched to the CU, which takes around 20 µs per kernel. It is common to buffer hundreds of kernels in a device queue, since it can reduce the frequency of context switches by filling up the device queue with a large number of kernels from host queues at a time. However, it may also increase the preemption delay to even more than 1 ms. Therefore, REEF restricts the capacity of the device queue to achieve microsecond-scale kernel preemption. Tuning the device queue capacity provides a tradeoff between preemption latency and execution time. As the queue capacity decreases, the preemption latency also decreases because fewer kernels need to be evicted, but normal execution time increases because the GPU has more idle time waiting for the runtime to fill device queues with the kernels from host queues. We empirically choose a device queue capacity to 4 on our testbed, since it is sufficient to reset the device queue in 30 µs with negligible overhead on normal execution time (i.e., less than 0.3%). Furthermore, using a smaller device queue also produces slightly higher CPU utilization (e.g., about 15% increase) due to more frequent filling of the device queue.

4.2 Killing Running Kernels

To avoid waiting for the completion of running kernels, the reset-based preemption proactively kills the running kernels in the GPU. Unfortunately, there is neither an API provided by GPU runtime nor a functionality exposed by GPU driver that can kill the running kernels from the host side. We observed that GPU driver has the ability to terminate CPU process and also kill associated GPU kernels, even when the kernel stucks in an infinite loop. It implies that GPU driver can indeed kill an uncompleted kernel. However, this function will also reclaim GPU memory allocated by the process and GPU kernels. Thus, the preempted kernel has to reload DNN model parameters to GPU memory, taking even a few seconds.

To remedy it, REEF retrofits the kernel killing function of GPU driver and exposes it to the preemption module in GPU runtime. The new function will instruct the command processor to kill all running kernels on the CUs but preserve their running state in GPU memory. The preemption module will use it to kill all running kernels (see ❸ in Fig. 7) after evicting host queues and device queues.

4.3 Restoring Preempted Tasks

The best-effort tasks should be restored after being preempted. In general, the task has to be re-executed from the beginning, and is assumed to have no side effects. Fortunately, the *idempotence* characteristic of kernels in the DNN model ensures that the execution of DNN inference task can be restored from any kernel before the interrupted kernel. This implies that the scheduler can safely re-execute the preempted best-effort tasks. However, this may incur severe additional overhead because DNN models commonly have massive kernels (usually hundreds or more). Therefore, it is important to restore the preempted task from the kernel close to where it was interrupted. Unfortunately, it is almost impossible to precisely identify the interrupted kernel, because the kernel running on the CUs is killed directly by the command processor of GPU.

To remedy this problem, REEF adopts an *approximation* approach to ensure that the preempted task is restored from at most a *constant* number (c) of kernels before the interrupted kernel. More specifically, the preemption module first records the last kernel (k_l) transmitted to the device queue when it starts resetting the task queue, and then restores the preempted task from c kernels before k_l, where c denotes the device queue capacity. We observe that the command processor sequentially fetches a kernel from the device queue and runs it on the CUs. This implies that the interrupted kernel will not be earlier than c kernels before the last kernel (k_l) in the device queue. Furthermore, REEF will redundantly execute at most c+1 kernels. Since c is configured to be relatively small (i.e., 4), the restore overhead is negligible (about 30 µs).

4.4 Preemption on closed-source GPUs

Many commodity GPUs (e.g., NVIDIA GPUs) are still closed source. This poses new challenges to our reset-based preemption scheme, which has to treat the GPU runtime as a black box. The primary limitation is that we cannot reset CUs to proactively kill running kernels (❸ in Fig. 7). Apart from that, REEF is also unable to manipulate host queues and device queues directly outside of the GPU runtime. But fortunately, the lazy eviction scheme proposed by REEF for resetting DQs (❷ in Fig. 7) does not require any modification to the GPU runtime.

We propose a restricted version of reset-based preemption, called REEF-N, for closed-source GPUs. REEF-N first wraps each GPU stream, the general abstraction provided by GPU runtime, into a *virtual* host queue (vHQs), which intercepts and buffers all launched kernels. Similar to the (physical) HQ inside the GPU runtime, each vHQ also has a background thread to transmit buffered kernels asynchronously to the GPU runtime. After that, REEF-N treats the whole GPU runtime as several device queues (one for each GPU stream), such that REEF can easily reset vHQs to evict buffered kernels, instead of resetting HQs directly (❶ in Fig. 7). REEF-N still follows the lazy eviction to reset DQs, and then waits for all running kernels to complete. Finally, to simulate DQ capacity restriction, REEF limits the number of outstanding kernels in the GPU runtime; the background thread of vHQ transmits a fixed number of kernels to the GPU runtime in a closed loop.

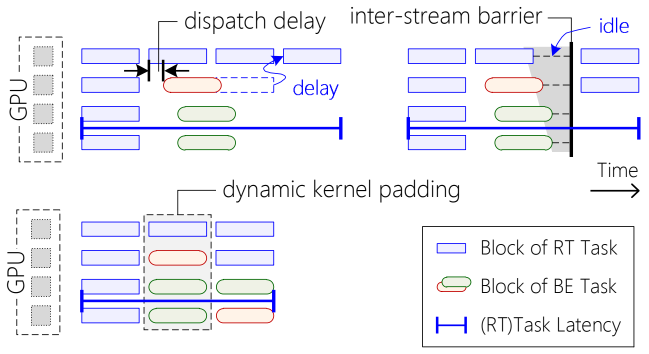

Fig. 8: *An example of serving multiple kernels in parallel with different approaches.*

5 Dynamic Kernel Padding

To achieve high throughput, both real-time and best-effort tasks should be concurrently executed on the GPU to achieve work conserving. However, to avoid interference with real-time tasks, the best-effort tasks should be only served by using GPU resources leftover from the real-time tasks. Regrettably, none of the existing approaches can provide such controlled concurrent execution on the GPU.

First, using different GPU streams to launch real-time and best-effort tasks cannot avoid interfering with each other. As shown in Fig. 8, the dispatch delay between GPU streams (20–40 µs) might postpone the execution of real-time kernels or limit the available resources (e.g., CUs) to them. Using additional inter-stream barriers to synchronize kernel dispatch among CUs will also cause performance overhead.

Second, static kernel fusion [74] can merge multiple kernels from different tasks into a single one at compile time and then launch the fused kernel on the GPU using a single stream. It can avoid interference between real-time tasks and best-effort tasks in advance. However, static kernel fusion has to pre-compile all possible combinations of all kernels in DNN models to enable scheduling at runtime. As mentioned above, DNN inferences have hundreds of kernels in common (see Table 1), which makes it impractical for static kernel fusion. For example, it requires more than 35 GB of GPU memory to store the fused kernels for five DNN models in Table 1—considering only all combinations of no more than three kernels.

```
# device codes
__device__ void dense(in, weight, bias, out): ...

__global__ void dkp(rt_kern, rt_args,
                    be_kerns, be_argss):
1   ncus = rt_kern.ncus  # number of CUs
2   if (cu_id() < ncus) then
3      rt_kern(rt_args)  # run RT/kernel
4   else
5      ncus += be_kerns[i=0].ncus
6      while (cu_id() >= ncus)
7         ncus += be_kerns[++i].ncus
8      be_kerns[i](be_argss[i])  # run BE/kernel

# host codes
void inference(...):
    # set the real-time kernel w/ its args (e.g., dense)
9   rt_kern, rt_args = ...
    # select a set of best-effort kernels w/ their args
10  be_kerns, be_argss = kern_select(rt_kern)
11  dkp <<<..>>> (rt_kern, rt_args, be_kerns, be_argss)
12  ... # launch other dynamic padded kernels
```

Fig. 9: *Pseudocode for dynamic kernel padding in* REEF.

Our approach: dynamic kernel padding. Inspired by kernel fusion, our approach also combines real-time kernels and best-effort kernels into a single one and launches it using a single GPU stream, as shown in Fig. 8. Differently, we construct a template (called *dkp kernel*) at compile time and use *function pointer* to fill and execute kernels at runtime. Further, we dynamically select best-effort kernels to avoid interference with the real-time kernel.

Fig. 9 shows an example of a dkp kernel (`dkp`) for dynamic kernel padding, declared as a *global* function (i.e., kernel entry). Instead of being statically inlined into the dkp kernel, candidate kernel functions (e.g., `dense`) are declared as individual *device* functions, which can be passed as dkp kernel arguments and called by function pointers (line 3 and 8). The dkp kernel partitions the CUs to execute one real-time candidate kernel (`rt_kern`) and a set of best-effort candidate kernels (`be_kerns`) in parallel. It first allocates sufficient CUs for the real-time kernel (line 1–3) and then assigns the leftover CUs to the best-effort kernels (line 5–8). When launching a real-time kernel, the DKP module selects appropriate best-effort kernels to concurrently execute with the real-time kernel (line 10, see also §5.2).

5.1 Efficient Function Pointers

Without specific optimizations, the naive design would significantly decrease the performance of real-time kernels, due to the unique characteristics of *function pointers* on the GPU. We summarize the two key performance issues of the default function pointer mechanism on the GPU.

Limited register allocation. Unlike CPU programs, GPU programs require a diverse yet fixed amount of registers, which is counted at compile time and encoded into the model executable. Such an attribute prohibits the direct use of function pointers in GPU kernels, as the number of registers used by the indirectly called function cannot be determined statically. The default behavior of the GPU compiler is to assign a predefined static upper bound to limit the callee's register usage, which may force the callee to save variables on the stack due to the insufficient registers, leading to poor performance compared to purely using registers [43].

Expensive context saving. Indirect function calls on GPUs are much more expensive than CPU programs, due to the enormous context (e.g., dozens of registers) that needs to be saved and restored before and after the function call. For thousands of threads, there might be MB-sized registers saved and restored, introducing significant overheads. Although the compiler will inline as many functions as possible to avoid this overhead, indirect function calls via function pointers cannot be inlined, which may impose significant performance penalty on dynamic kernel padding.

REEF tackles the two above issues by introducing *global function pointer* as a substitution of the default function pointer mechanism. Since global functions are treated as kernel entries, the compiler neither applies register limitations nor adds context saving/restoring code to them. Thus, declaring candidate kernels as global functions instead of device functions can solve both issues. According to our observation, context saving in candidate kernels is actually unnecessary, as the dkp kernel exits immediately after calling `rt_kern` or `be_kerns[i]` (see Fig. 9). Therefore, the lack of context saving code in candidate kernels does not affect the execution correctness.

However, as the kernel entry, a global function cannot be called by another global function (e.g., dkp kernel). To bypass this restriction, we replace indirect function calls with jump instructions in assembly code, and manually prepare the initial state of candidate kernels by following the conventions [45]. This approach makes no changes to the compiler and only incurs a trivial function call overhead (around 1%).

Dynamic register allocation. The real-time kernel performance is still not ideal after applying the global function pointer technique because of the *over-allocation* problem. To meet the varied register demands of candidate kernels, the dkp kernel has to allocate as many registers as possible (i.e., over-allocation), which may decrease the CU occupancy[3], and thus increase the execution time. An intuitive solution is to overwrite the register count of the dkp kernel just-in-time before it is launched, making it adaptive to selected candidate kernels. Unfortunately, the kernel's register count has been loaded to the GPU memory with the model in the off-line phase (§3), which means overwriting its value requires a CPU-to-GPU memory copy before every kernel execution, severely affecting the execution performance.

REEF addresses the dynamic register allocation problem

[3]The CU occupancy implies how many blocks can be executed on a CU simultaneously. It depends on how many resources (e.g., register) each block demands. Higher CU occupancy can lead to better performance.

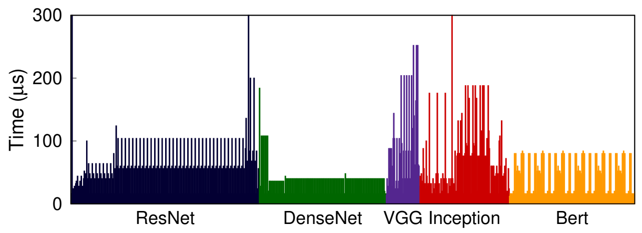

Fig. 10: *The measured execution time of kernels in five DNN models. The details of DNN models can be found in Table 1.*

by introducing a set of *proxy kernels*. Proxy kernels share the same source code as the dkp kernel in Fig. 9, but allocate different number of registers, allowing the scheduler to dynamically pick the proper proxy kernel according to each candidate kernel's register demand. Unfortunately, generating proxy kernels for every possible register count faces the kernel amount explosion problem. For example, on AMD Instinct MI50 GPU with at most 128 scalar registers and 256 vector registers for each thread, it will generate 32,768 proxy kernels to cover all possible register configurations.

To reduce the proxy kernel amount, we generate proxy kernels to cover all possible CU occupancies rather than register counts. Since proxy kernels are introduced to prevent over-allocation from decreasing the CU occupancy, proxy kernels that have different register count yet share the same CU occupancy are actually redundant and can be merged together. More specifically, there are 10 CU occupancy levels on AMD Instinct MI50 GPU we use, corresponding to 10 register count ranges, which allows us to generate only 10 proxy kernels, each allocating the maximum amount of registers allowed in a CU occupancy level. For each candidate kernel, the scheduler picks the proxy kernel with the fewest allocated registers that fulfill the candidate kernel's demand, which achieves the highest CU occupancy possible. This way, the amount of proxy kernels is narrowed down from 32,768 to 10 without affecting the candidate kernel's performance.

Dynamic shared memory. In addition to registers, over-allocation of shared memory may also decrease the CU occupancy of proxy kernels. Fortunately, the kernel is enabled to dynamically allocate shared memory by setting a property (i.e., "dynamic shared memory") when launching the kernel. During model compilation, REEF converts the declaration of variables from *fixed-size* shared memory to *dynamic* shared memory (i.e., adding $extern$ before $\_\_shared\_\_$). Consequently, the amount of shared memory used by proxy kernels can be set at runtime, depending on the maximum demand of candidate kernels.

5.2 Kernel Selection

For dynamic kernel padding, the *kernel selection policy* is important to avoid latency interference with real-time tasks, which selects a set of blocks from candidate best-effort kernels to share the GPU with the arriving real-time kernel. REEF proposes a greedy heuristic to ensure that the best-effort blocks will only use GPU resources (i.e., CUs) leftover from the real-time kernel. Specifically, it first reserves enough CUs for the real-time kernel, and then checks best-effort task queues to select appropriate blocks for the remaining CUs, until there are no free CUs or candidate tasks. The selected best-effort blocks should meet the following two rules.

Rule 1. *The execution time of best-effort kernels must be shorter than that of the real-time kernel*, since the execution time of the dkp kernel is determined by the slowest block. Based on the observation of latency predictability for GPU kernels in DNN models (see §2.1), we develop an offline kernel profiler to measure the computational requirements and the execution time for each kernels of loaded models.

Rule 2. *The CU occupancy of best-effort kernels must be higher than that of the real-time kernel*, since the CU occupancy of the dkp kernel is determined by the minimum of kernels. Note that the CU occupancy of kernels can be directly obtained from the source code of DNN models.

The kernel selection policy fully meets the design goal of treating the real-time tasks as first-class citizens on the GPU. It is not only efficient, selecting best-effort kernels in less than 1 µs, but also effective, limiting the latency overhead of real-time kernels to less than 1% on average, see §7.4 for details. However, the policy is also conservative, so the constraint may limit room for improvement in overall throughput. For example, when the execution time of best-effort kernels is often longer than that of real-time kernels (e.g., VGG and DenseNet in Fig. 10), the throughput improvement of dynamic kernel padding may be trivial, even if the real-time tasks only use a few CUs.

6 Implementation

We first implemented and deployed REEF on AMD GPUs because of its open-source platform and ISA [26, 54], which can fully demonstrate the efficacy of reset-based preemption and dynamic kernel padding. REEF was implemented by extending Apache TVM [73] and AMD ROCm [3], with about 5,500 lines of C++ code. Beyond that, to further show the feasibility of REEF on closed-source GPUs, we also ported REEF-N, a restricted version of reset-based preemption, on NVIDIA GPUs with CUDA [52].

Model compiler. REEF extends Apache TVM [15], a machine learning compiler framework, with a code transformer, which mainly adds two modifications to the source code of DNN inference: (1) a *preemption flag*, which is injected into kernel arguments to lazily evict the kernel; (2) a set of *proxy kernels*, which is constructed for the padded kernels.

GPU runtime. For AMD GPUs, REEF builds the preemption module on HIP [63] of ROCm, a portable GPU runtime and programming library. similar to NVIDIA CUDA [52]. Specifically, REEF adds three new APIs to GPU runtime: (1) `hip_reset_hq`, which resets host queues and moves commands to the GC thread; (2) `hip_set_stream_cap`, which limits the capacity of the device queue used by a GPU stream; (3) `hip_reset_kern`, which resets the compute units by using hardware mechanisms via the GPU driver in Linux [61].

For NVIDIA GPUs, REEF-N intercepts three CUDA APIs related to kernel launch and stream management, and adds the following operations: (1) `cuStreamCreate`, which creates a vHQ and links it to the created CUDA stream; (2) `cuKernelLaunch`, which buffers the launched kernel in the vHQ and transmits it to GPU runtime (i.e., CUDA [52]) in the background; (3) `cuStreamSynchronize`, which waits for GPU runtime to complete all launched kernel of the CUDA stream. Finally, REEF-N provides a new API `cuResetHQ` to reset vHQ by dequeuing all buffered kernels.

7 Evaluation

7.1 Experimental Setup

Testbed. The experiments were mainly conducted on a GPU server that consists of one Intel Core i7-10700 CPU (total 8 cores), 16 GB of DRAM, and one AMD Radeon Instinct MI50 GPU (60 CUs and 16GB of memory). The software environment of the server was configured with ROCm 4.3.0 [3], Apache TVM [73] 0.8.0, and Ubuntu 18.04. The hardware platform resembles the computational resources of autonomous vehicles [4, 71]. We further evaluate REEF-N on a closed-source GPU (NVIDIA V100 GPU) to demonstrate the generality of our approach, using the same server with CUDA 10.2 [52] installed.

Workloads. Inspired by YCSB [17, 18], we build a new DNN inference serving benchmark (DISB) that contains a suite of tools and five workloads: (A) low load, (B) high RT load, (C) high BE load, (D) multi-RT load, and (E) random load, summarized in Table 2. The real-time (RT) clients in DISB A–D uniformly send inference requests at a given frequency, which simulates real-time DNN applications in autonomous driving (e.g., obstacle recognition with cameras [7]), while the clients in DISB E send 20 requests per second with a Poisson arrival distribution, which simulates event-driven real-time DNN applications (e.g., speech recognition [32, 75]). Note that serving 220 RT requests per second sequentially for VGG model would saturate our testbed (see Fig. 1(d)). On the other hand, the closed-loop best-effort (BE) client continuously issues inference requests, which simulates a contention load on the GPU (e.g., driver monitoring).

Five representative DNN models are deployed in DISB, including ResNet-152 [30] (RNET), DenseNet-201 [35] (DNET), VGG-19 [68] (VGG), Inception v3 [69] (IN3), and DistilBert [66] (BERT), all generated by Apache TVM [15]. Each client always submits inference requests for a certain DNN model. Specifically, VGG is used by DISB A–C for their RT clients, and RNET is used by DISB A and B for their BE clients. Workloads with 5 RT/BE clients deploy all five

Table 2: *DISB workload description.* #/model *denotes the number of clients and their DNN models.* [U/P] *denotes an arrival distribution (i.e., Uniform or Poisson).*

DISB	A	B	C	D	E
Num. of RT clients	1/VGG	1/VGG	1/VGG	5/ALL	5/ALL
Frequency (reqs/s)	100 [U]	220 [U]	100 [U]	20 [U]	20 [P]
Num. of BE clients	1/RNET	1/RNET	5/ALL	5/ALL	5/ALL

DNN models in their clients separately, which simulates multiple DNN applications in a single scenario (e.g., autonomous vehicles [7, 41]).

Furthermore, we use a real-world trace from an open autonomous driving platform (i.e., Apollo [7]) as the real-time workload, which provides a realistic arrival distribution of real-time tasks in autonomous driving. The trace was collected from the logs of the perception module [5] when running Apollo with SVL simulator [42, 65], and we selected the closest DNN models in terms of execution time from the above five models for the inference requests. Meanwhile, the same best-effort workload as DISB C–E is used, where five clients continuously issue different DNN inference requests.

Currently, each workload in DISB represents a particular mix of real-time and best-effort DNN inference tasks, the number of clients, and request frequency, which focuses on a particular point in the performance space. Users can further extend DISB with new workloads, or even some production traces from specific applications, to model more different scenarios.

Comparing targets. We compare REEF with typical scheduling approaches. **SEQ** sequentially runs each DNN inference task on the GPU with passive task preemption, which is adopted by Clockwork [28]. Specifically, when there are multiple tasks waiting in the queue, it prioritizes real-time tasks, but still needs to wait for the completion of launched best-effort tasks. **GPUStreams** runs both real-time and best-effort tasks simultaneously on the same GPU through multiple GPU streams, which is adopted by TensorRT [50]. As a reference, we further provide **RT-Only**, which represents the optimal end-to-end latency for real-time tasks, as it dedicates the GPU to real-time tasks.[4]

7.2 Overall Performance

We first compare the end-to-end latency of real-time tasks and the overall throughput of REEF with other approaches using DISB workloads and a real-world trace, as shown in Fig. 11.

Single BE Client (DISB A and B). For workloads with a single BE client, the performance impact of using SEQ or GPUStreams is relatively low, since GPU contention from best-effort tasks is not severe, either in terms of wait time (SEQ) or concurrent interference (GPUStreams). For DISB

[4]In this case, additional GPUs are dedicated to best-effort tasks, which also result in extra cost and energy consumption, as well as low GPU utilization.

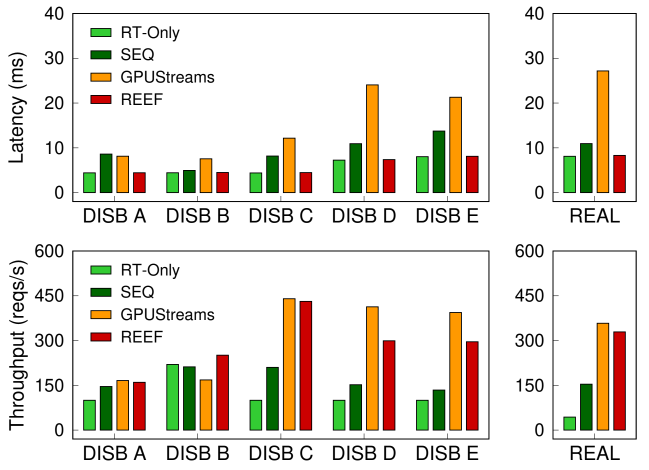

Fig. 11: *Comparison of (a) end-to-end real-time task latency, and (b) overall throughput (including both real-time and best-effort tasks) using different scheduling approaches.*

A, compared with RT-Only, SEQ and GPUStreams improve overall throughput by 1.46× and 1.66×, but also amplify real-time task latency by 1.95× and 1.84×, respectively. In contrast, REEF incurs negligible (0.5%) overhead on real-time task latency, but improves overall throughput by 1.60×, comparable to GPUStreams.

For DISB B, due to running real-time tasks more frequently, SEQ suffers 1.12× slowdown on real-time task latency, slightly better than DISB A, as it only has to wait for fewer best-effort tasks. However, its throughput only achieves 96% of RT-Only, since real-time tasks saturate the GPU and best-effort tasks have little chance to run. For similar reasons, the overall throughput of GPUStreams also drops to 76% of RT-Only, while its real-time task latency is still 1.70× higher than RT-only. Conversely, REEF can still limit the overhead on real-time task latency to 1% (about 60 µs) and provides a 1.14× speedup on overall throughput, thanks to our reset-based kernel preemption and dynamic kernel padding.

Multiple BE Clients (DISB C, D, and E). With the increase of best-effort workloads, the overall throughput of all approaches improve to varying degrees over RT-Only by sharing the GPU between two types of tasks. However, they have very different performance in terms of real-time task latency. Both SEQ and GPUStream make the same tradeoff between real-time task latency and overall throughput, differing only in the magnitude of the performance impact. For three workloads, SEQ improves overall throughput by 1.34× to 2.10×, but also amplifies real-time task latency by 1.51× to 1.86×. For GPUStreams, the above numbers become 3.94× to 8.19× and 2.65× to 3.31×.

Differently, REEF improves overall throughput as much as possible, based on the premise that real-time tasks should not be affected in any way. As a result, REEF offers almost the same real-time task latency as RT-Only in all workloads, with less than 1.5% overhead (0.1 ms). For overall throughput,

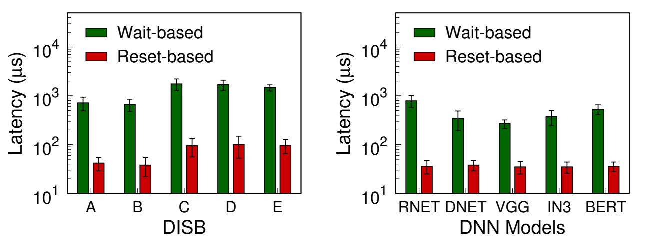

Fig. 12: *Comparison of preemption latency between reset-based and wait-based approaches (a) on DISB workloads, and (b) when preempting one DNN inference task of different DNN models.*

REEF provides a close result of GPUStreams on DISB C, since VGG is easy to be padded with most DNN models (see §5.2). On DISB D and E, the throughput of REEF is about 25% lower than that of GPUStreams, due to using a mix of five DNN models for real-time tasks, while DKP does not always work well on a few combinations of real-time and best-effort tasks (see §7.4 for details). However, REEF still outperforms RT-Only by 3.00× and 2.96×, respectively.

Real-world workload from Apollo (REAL). For the real-world workload, compared to RT-Only, SEQ and GPUStreams increase overall throughput by 3.6× and 8.3×, while amplifying the latency of real-time tasks by 1.35× and 3.35×, respectively. Due to the low load of real-time tasks in the real-world trace (about 43 reqs/s), REEF stays in normal mode to execute best-effort tasks concurrently most of the time, similar to GPUStreams. Therefore, compared to RT-Only, REEF achieves 7.7× throughput improvement with less than 2% latency overhead for real-time tasks, thanks to our reset-based preemption, which can preempt the GPU within tens of microseconds after the real-time task arrives.

7.3 DNN Inference Preemption

The vanilla wait-based preemption approach proposed in prior work [12] is not practical for DNN inference serving, since it only allows executing tasks one by one. Therefore, we extended it to allow concurrent inference serving by removing the limit on the amount of launched kernels and also implementing *lazy eviction*. This version is used as the baseline to demonstrate the efficiency of our reset-based preemption.

Preemption latency. Fig. 12(a) compares the preemption latency of two approaches. The reset-based preemption outperforms the wait-based approach by more than an order of magnitude for all DISB workloads, from 15.3× (DISB E) to 18.5× (DISB C). The main reason is that the wait-based approach has to passively wait for the completion of running kernels in CUs and the eviction of massive kernels in host and device queues, while the reset-based approach is able to proactively kill all kernels (usually much less) in these three places. As expected, both approaches take more time to handle multiple concurrent BE clients (DISB C, D, and E) than a singel BE client (DISB A and B).

Furthermore, we evaluate the preemption latency for di-

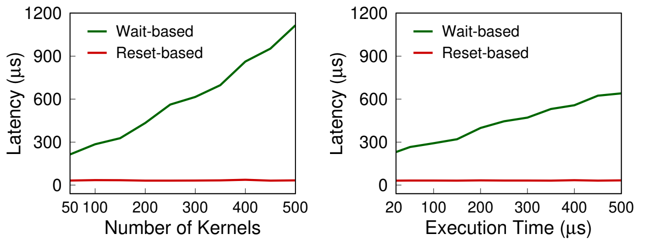

Fig. 13: *Comparison of preemption latency with the increase of (a) launched kernels and (b) kernel execution time.*

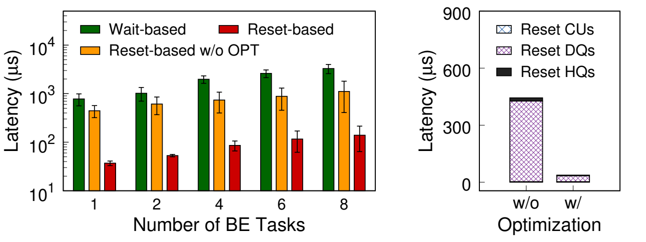

Fig. 14: *(a) Comparison of preemption latency with the increase of BE clients, and (b) the latency breakdown of the reset-based approach w/o and w/ optimizations.*

verse DNN models, where we use a single BE client to send inference requests for a given model and send a real-time request after a random time interval to preempt the GPU. As shown in Fig. 12(b), the wait-based preemption latency highly depends on the type of models, from 268 µs (VGG) to 790 µs (RNET), due to the difference in the number of kernels and the execution time (see Table 1). In contrast, the reset-based approach is not sensitive to DNN models and can preempt the GPU in the range of 35 µs to 38 µs for all five models.

To further investigate the impact of different model properties on the preemption latency, we simulate DNN models with different number of launched kernels and kernel execution times. By default, we set the number of launched kernels and the kernel execution time to 100 and 100 µs, respectively. As shown in Fig. 13, the preemption latency of wait-based approach raises linearly, while our reset-based preemption approach remains stable at very low latency (less than 40 µs). For wait-based approach, the preemption latency is significantly positively correlated with as number of launched kernels and the kernel execution time, since it has to wait for the eviction of launched kernels and the completion of running kernels. In contrast, the reset-based approach proactively resets the host and device queues in GPU runtime, as well as the CUs, where the cost is independent of model properties.

Optimizations. We propose two optimizations on the reset-based preemption approach, namely asynchronous memory reclamation and queue capacity restriction. To demonstrate the effect of optimizations, Fig. 14 shows the preemption latency with the increase of BE clients (RNET), and the latency breakdown for a single BE client. By enabling two optimizations, the preemption latency significantly drops by up to 92% (from 87%), as shown in Fig. 14(a). As a reference, even with-

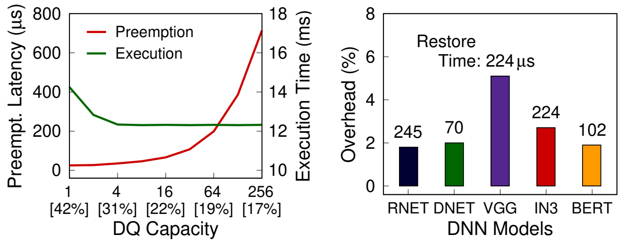

Fig. 15: *(a) The preemption latency and execution time with the increase of device queue capacity, where [%] shows the CPU utilization during normal execution, and (b) the restore overhead for different DNN models, where labels show the restore time (in µs).*

out optimization, the reset-based approach still outperforms wait-based approach by up to 3.0× (from 1.7×).

Since the two optimizations are used when resetting host and device queues, respectively, Fig. 14(b) breaks down the preemption latency to show the contribution of two optimizations separately. For a single BE client, using asynchronous memory reclamation reduces the latency of resetting host queue from 17 µs to 3 µs. Meanwhile, using queue capacity restriction further reduces the latency of resetting device queue from 424 µs to 31 µs. Note that using command processor to reset CUs is extremely fast (less than 3 µs).

Queue capacity. We restrict the device queue capacity to mitigate the overhead incurred by lazily evicting the remaining kernels in the queue (see §4.1 for details). However, reducing queue capacity also increases normal execution time and CPU utilization. Fig. 15(a) shows the preemption latency and normal execution time when serving RNET inferences as the queue capacity increases. When the device queue capacity increases from 1 to 4, the execution time reduces from 14.3 ms to 12.3 ms. However, when the capacity further increases, the change in execution time becomes trivial (less than 0.3%). Conversely, the preemption latency increases linearly with the queue capacity. Therefore, as a reasonable tradeoff between preemption latency and normal execution time, REEF adopts a default capacity of 4 for the device queue on our testbed, which has almost zero overhead for normal execution and provides acceptable preemption performance (about 30 µs). Finally, using a smaller device queue also results in higher CPU utilization. For instance, reducing the queue capacity from 256 to 4 increases CPU utilization from 17% to 31%.

Task restore. We further evaluate the execution time overhead of preempted tasks due to task restore. We use a single BE client to send inference requests; for each task, we randomly preempt and restore it. As shown in Fig. 15(b), the restore time for all DNN models is low, ranging from 70 µs to 245 µs, which mainly depends on the kernel execution time of DNN models (see Fig. 10). Note that REEF redundantly executes at most five kernels for restoring preempted tasks, thanks to the queue capacity restriction. Further, the execution time overhead is about 2% for all DNN models, except for VGG (5.1%), as it has the fewest kernels (55), and its kernel

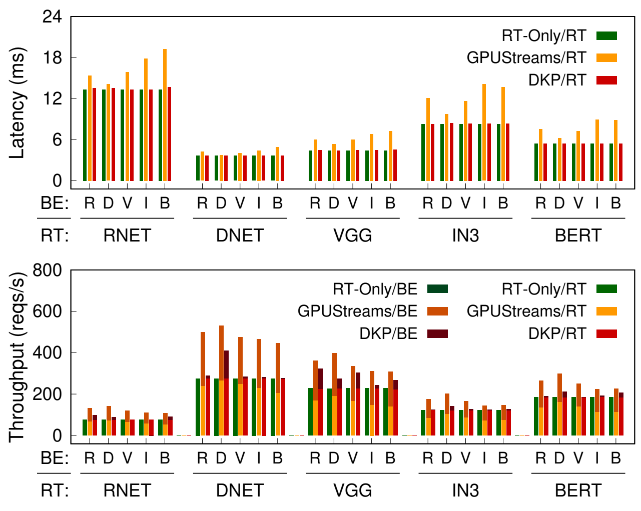

Fig. 16: *Comparison of (a) end-to-end latency of RT tasks and (b) overall throughput using different concurrent execution schemes.*

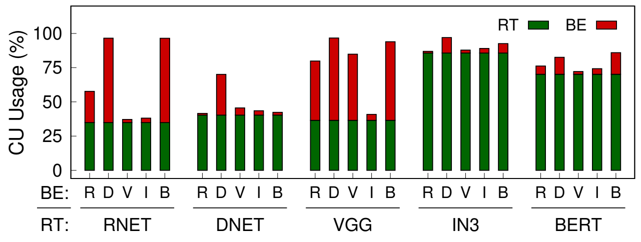

Fig. 17: *The average CU usage for running real-time and best-effort kernels with different combinations of DNN models using dynamic kernel padding.*

execution time is longer (see Fig. 10).

7.4 Dynamic Kernel Padding

To study the efficacy of dynamic kernel padding, we use a high contention workload, where one RT client and one BE client simultaneously send requests at a high-enough frequency to keep the GPU busy. RT-Only serves only real-time tasks to ensure optimal (real-time) task latency, while GPUStreams serves both types of requests concurrently to achieve the highest overall throughput. Differently, dynamic kernel padding also serves only real-time tasks but pads best-effort tasks to avoid starvation and improve overall throughput.

Performance. Fig. 16 reports the experimental results for one-to-one combinations among five DNN models using above workload. As expected, GPUStreams significantly amplifies real-time task latency by an average of 1.35×, ranging from 1.04× to 1.70×, due to severe interference from concurrent best-effort tasks. However, REEF is able to provide almost optimal latency to real-time tasks, with an average overhead of just 1% (up to 3%).

For overall throughput, we separately report the throughput of real-time tasks and the normalized throughput of best-effort

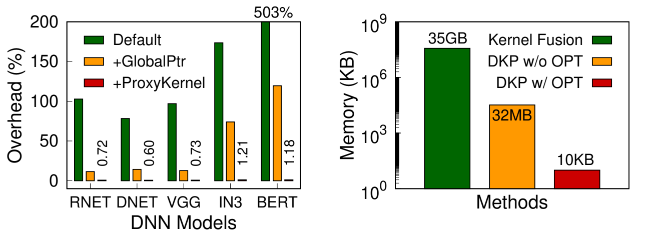

Fig. 18: *Comparison of (a) execution time overhead and (b) memory overhead for padded kernels using different optimizations.*

tasks.[5] For RT-Only, the GPU is busy serving real-time tasks, so the throughput of best-effort tasks is zero (even if RT-Only is willing to serve them). Although GPUStreams increases overall throughput by an average of 1.52×, the throughput of real-time tasks drops by 24.4% on average, due to severe interference in concurrent execution. Conversely, REEF first guarantees throughput for real-time tasks and then leverages dynamic kernel padding to increase overall throughput. The performance improvement mainly depends on two conditions. First, the execution of real-time tasks on the GPU leaves room for improvement. As shown in Fig. 17, the real-time kernels in IN3 and BERT use an average of 85% and 70% of CUs, respectively. Therefore, dynamic kernel padding hardly improves such cases, increasing just 6% on average. Note that GPUStreams can still improve overall throughput of them, but also greatly sacrifices the performance of real-time tasks. Second, the execution time of best-effort kernels must be shorter than that of the padded real-time kernels. This explains why REEF can achieve large improvement (1.41×) by padding VGG with RNET, but not vice versa, which is also confirmed by the increase of CU usage (BE) in Fig. 17

Optimizations. To investigate the impact of optimizations on both performance and memory usage, we first evaluate the overhead using different implementations of the function pointer on the GPU. We measured such overhead by launching real-time kernels through the dkp kernel without padding any best-effort kernels. As shown in Fig. 18(a), the default function pointer implementation (Default) incurs execution time overhead from 78% up to 503% for real-time tasks with different DNN models. By using the global function pointer (GlobalPtr), the overhead is significantly reduced to 46.4% on average (from 11.5% to 120%), as it eliminates the limit on the number of registers for device function pointers and avoids additional register saving and restoring during the function call. Finally, the overhead drops to 0.8% on average (1.21% at most) by using proxy kernel (ProxyKernel), which can dynamically allocate registers to each kernel and maximize CU occupancy. The minimal overhead comes from the logic branch of CU partition and the initial state preparation for global function pointers.

We further evaluate the impact of optimizations on reduc-

[5]The throughput of best-effort tasks is normalized to that of real-time tasks, following the formula: $throughput_{\text{BE}} \times (latency_{\text{BE}} \ / \ latency_{\text{RT}})$.

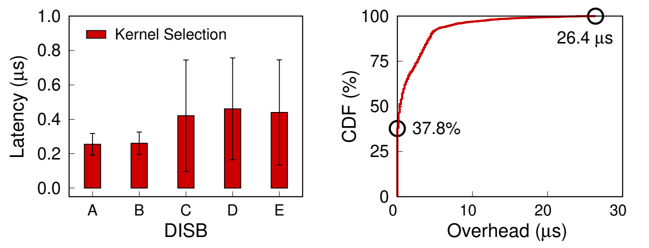

Fig. 19: *(a) The execution time of kernel selection for DISB A-E and (b) the CDF of execution time overhead for real-time kernels using dynamic kernel padding.*

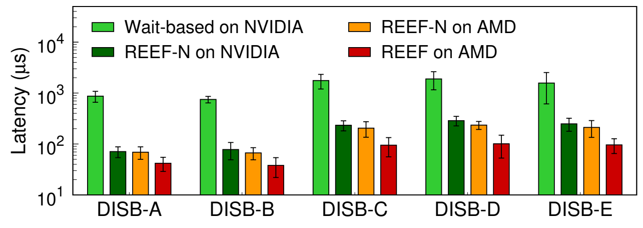

Fig. 20: *Comparison of preemption latency on NVIDIA and AMD GPUs using different preemption schemes with DISB workloads.*

ing GPU memory usage. As shown in Fig. 18(b), using static kernel fusion (Kernel Fusion) requires over 35 GB of GPU memory to store the fused kernels for five DNN models—all combinations of no more than three kernels, which even exceeds the memory capacity of most commodity GPUs. REEF proposes proxy kernels (DKP w/o OPT) to reduce GPU memory usage to about 32 MB. Finally, generating proxy kernels to cover all possible CU occupancies (DKP w/ OPT), instead of all possible register configurations, can dramatically reduce GPU memory usage to only 10 KB.

Kernel selection. Fig. 19(a) shows the average time of kernel selection for DISB A-E during dynamic kernel padding. For workloads with a single BE client (DISB A and B), REEF takes about 0.2 µs to select best-effort kernels for the given real-time kernel. The selection time increases to 0.4 µs for workloads with multiple BE clients (DISB C, D, and E) due to more candidates. In general, the cost of kernel selection is quite trivial and can be easily hidden by kernel execution.

To further study the accuracy of kernel selection, we evaluate the execution time overhead for the real-time kernel due to padding best-effort kernels on all DISB workloads. As shown in Fig. 19(b), over 37% of real-time kernels are not negatively impacted by concurrent execution with best-effort kernels, and the overhead of more than 90% real-time kernels is still less than 4 µs. The increase of execution time is mainly due to the contention on GPU memory and shared L2 cache.

7.5 Closed-source GPUs

Finally, we evaluate REEF-N, a restricted version of reset-based preemption using DISB workloads on both NVIDIA and AMD GPUs, and compare it to the wait-based approach and REEF, respectively. As shown in Fig. 20, even if REEF-N does not reset CUs to proactively kill running kernels, the preemption latency just ranges from 71µs to 288µs, which still outperforms the wait-based approach by up to 12.3× (from 6.3×) on the NVIDIA GPU. By comparing REEF-N and REEF on the AMD GPU, we observe that killing running kernels proactively further contributes to an average speedup of 2.0× in preemption latency, especially for preempting concurrent tasks (e.g., 2.3× for DISB C). In addition, the performance of REEF-N is close on two GPUs.

8 Discussion

Assumption of idempotence. The reset-based preemption in REEF is based on the assumption that each kernel in DNN inference should be idempotent. Currently, all DNN inference kernels we encountered, a total of 320 kernels from 11 models [72], are shown to be idempotent. However, readers might be interested in whether our approach still works with kernels without the idempotence assumption. Strictly speaking, the reset-based preemption demands that the kernel always produces the same output for the same input no matter it has been retried or not. Therefore, a transactionization approach [40] can be used to transform non-idempotent kernels into idempotent ones if necessary. Furthermore, since only best-effort kernels may be preempted in REEF, this transformation only sacrifices the performance of transformed kernels (i.e., best-effort kernels) to ensure that real-time kernels can be instantly executed upon arrival with no performance penalty. We leave the incorporation of this technique to future work until we actually encounter non-idempotent DNN kernels.

Restrictions on kernel selection. The current kernel selection policy is effective but conservative, since the primary goal of REEF is to avoid performance interference with real-time tasks. An obvious limitation is the constraint that the execution time of best-effort kernels must be shorter than that of the padded real-time kernel, which limits room for improvement in overall throughput. We found that the GPU kernel can be tailored towards shorter execution time per block by using more thread blocks during model compilation. For example, Apache TVM automatically tunes the number of thread blocks for overall performance, but also allows developers to customize it [38]. Currently, the overall throughput improvement of REEF is largely attributed to enabling instant kernel preemption, which allows the idle GPU to perform best-effort tasks. Thus, we leave it to future work to overcome the restriction on kernel selection. Furthermore, the policy does not consider the contention for GPU memory between real-time and best-effort kernels, since it is still sufficient for running multiple DNN inference tasks. We also leave it to future work.

Future GPU APIs and runtime. We leverage several subtle hacks on the GPU runtime to enable µs-scale reset-based preemption on commodity GPUs. Our work also informs the design of future GPU APIs and runtime. First, given that commodity GPUs are generally capable of resetting compute units (CUs), a separate GPU API to precisely reset CUs is feasible and would be useful to kill and restore all running kernels. Second, we propose a new GPU API that instructs the command processor to discard fetched kernels and stop fetching more kernels from the device queue (DQs). Based on it, DQs can be proactively reset with a hardware-software co-design, replacing our software-only solution (i.e., lazy eviction). Finally, the GPU runtime could provide a high-level API for developers to reset the GPU stream, by discarding kernels buffered in internal data structures (e.g., host queues) and resetting the GPU via two new APIs. We believe that these extensions can greatly simplify implementation, even fully implementing reset-based preemption on closed-source GPUs, and further improve performance, for example instantly preempting the GPU in 10 µs.

9 Related Work

DNN inference serving systems. Prior model serving systems [21, 25, 29, 53, 79] mainly focus on meeting service-level objectives (SLO), typically in the tens of milliseconds [22, 31, 81], and improving overall throughput of datacenter applications. Clockwork [28] leverages the latency predictability of DNN inference to achieve low tail latency. It runs inferences sequentially on dedicated GPUs to provide predictable performance. Clipper [20] and Nexus [67] enables batching inferences on the same model to improve GPU utilization and inference throughput. Abacus [22] enables simultaneous DNN inferences by accurately predicting the latency of the overlapped operators. INFaaS [64] can automatically select the right variant with different optimizations for each inference to meet diverse SLOs. However, the latency SLOs for datacenter applications are much more relaxed than those for real-time systems, for example 2× of their solo-run latencies [22]. Therefore, using non-preemptive scheduling or batching scheme is effective for datacenter applications, but not for real-time scenarios (e.g., autonomous vehicles). Furthermore, the design of REEF is orthogonal to the above distributed serving systems. Two key mechanisms in REEF can also be integrated into them to improve per-GPU throughput and preserve low latency for real-time inferences.

GPU kernel preemption. Apart from the software preemption techniques, prior work also has proposed hardware enhancement to support preemptive GPU scheduling [44, 56, 70]. An intuitive solution is to support context switch on GPUs [70]. However, it is far more expensive on GPU than CPU due to the large context (e.g., a large amount of registers). Zhen et al. [44] proposed lightweight context switching to avoid unnecessary register saving. Tanasić et al. [70] extended the hardware to passively preempt a streaming multiprocessor (SM) of GPU by stopping issuing new thread blocks. Chimera [56] further proposed SM flushing to instantly preempt an SM when detecting idempotent execution. Differently, our approach retrofits existing hardware mechanism and requires no modification on the GPU to implement instant preemption.

GPU multitasking. There have been many efforts to concurrently execute multiple GPU kernels for high throughput [27, 43, 55, 57, 74, 76]. For DNN computation, Rammer [47] takes a holistic approach to exploit both inter- and intra-kernel parallelisms at compile time, which uses static kernel fusion [74] to enforce the CU assignments of the concurrent kernels. However, static kernel fusion requires the fused kernels to be known at compile time, which is not applicable for dynamic task scheduling in REEF. REEF proposes dynamic kernel padding to allow making scheduling decisions at runtime. Prior work has also proposed approaches to model and predict the slowdown of concurrent kernel execution [13, 14, 86, 88]. DASE [34] models the memory contention of concurrent kernels. Themis [87] uses a neural network to predict the performance interference. The prediction can help make scheduling decisions to match the latency requirements of real-time kernels. However, the prediction cannot always be accurate, and the slowdown actually happens. Differently, dynamic kernel padding in REEF enforces concurrent kernels to use only GPU resources leftover from the real-time kernel. Currently, REEF mainly focuses on GPU computational resources (i.e., CUs) and assumes that other resources are sufficient (e.g., GPU memory and bandwidth). We leave it as future work.

10 Conclusion

This paper presented REEF, the first DNN inference serving system for commodity GPUs. It enables microsecond-scale kernel preemption and controlled concurrent execution in GPU scheduling to achieve real time and work conserving. First, REEF can launch a real-time kernel on the GPU by proactively killing and restoring best-effort kernels at microsecond-scale. Second, REEF can dynamically pad the real-time kernel with appropriate best-effort kernels to fully exploit the GPU with negligible overhead. In addition, we built a new benchmark (DISB) for DNN inference serving that contains diverse workloads and a real-world trace. Evaluation using DISB and microbenchmarks confirmed the efficacy and efficiency of REEF on AMD and NVIDIA GPUs.

11 Acknowledgment

We sincerely thank our shepherd Dejan Kostić and the anonymous reviewers for their insightful comments and feedback, and Xiaoniu Song for sharing his experience in preparing the Artifact Evaluation. This work was supported in part by the National Natural Science Foundation of China (No. 61925206, 62132014), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 19511121100), and Shanghai AI Laboratory. Corresponding author: Rong Chen (`rongchen@sjtu.edu.cn`).

References

[1] Jacob Adriaens, Katherine Compton, Nam Sung Kim, and M. Schulte. The Case for GPGPU Spatial Multitasking. *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, 2012.

[2] Miguel Alcon, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. Timing of Autonomous Driving Software: Problem Analysis and Prospects for Future Solutions. *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 267–280, 2020.

[3] AMD ROCm. AMD ROCm Platform Documentation. https://rocmdocs.amd.com/, 2022.

[4] Apollo Auto. Apollo: Architecture/Hardware Connection. https://github.com/ApolloAuto/apollo, 2022.

[5] Apollo Auto. Apollo Perception Module. https://github.com/ApolloAuto/apollo/tree/master/modules/perception, 2022.

[6] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'20, pages 499–514, November 2020.

[7] Baidu. Apollo. https://apollo.auto/, 2022.

[8] C. Basaran and K. Kang. Supporting Preemptive Task Executions and Memory Copies in GPGPUs. In *24th Euromicro Conference on Real-Time Systems*, ECRTS'12, pages 287–296, 2012.

[9] Karsten Behrendt, Libor Novak, and Rami Botros. A Deep Learning Approach to Traffic Lights: Detection, Tracking, and Classification. *IEEE International Conference on Robotics and Automation*, pages 1370–1377, 2017.

[10] N. Capodieci, R. Cavicchioli, M. Bertogna, and Aingara Paramakuru. Deadline-Based Scheduling for GPU with Preemption Support. *IEEE Real-Time Systems Symposium*, pages 119–130, 2018.

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. *IEEE International Symposium on Workload Characterization*, pages 44–54, 2009.

[12] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. EffiSha: A Software Framework for Enabling Efficient Preemptive Scheduling of GPU. *22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017.

[13] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. *Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[14] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. *Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[15] T. Chen, T. Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, L. Ceze, Carlos Guestrin, and A. Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'18, 2018.

[16] Green Car Congress. New ultrafast camera for self-driving vehicles and drones. https://www.greencarcongress.com/2017/02/20170217-ntu.html, 2017.

[17] Brian F. Cooper. YCSB Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads, 2022.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *1st ACM Symposium on Cloud Computing*, SoCC'10, pages 143—-154, 2010.

[19] Alexander Craik, Yongtian He, and José Luis Contreras-Vidal. Deep Learning for Electroencephalogram (EEG) Classification Tasks: A Review. *Journal of neural engineering*, 16(3), 2019.

[20] D. Crankshaw, Xin Wang, Giulio Zhou, M. Franklin, Joseph E. Gonzalez, and I. Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'17, 2017.

[21] Weihao Cui, Mengze Wei, Quan Chen, Xiaoxin Tang, Jingwen Leng, Li Li, and Ming Guo. Ebird: Elastic Batch for Improving Responsiveness and Throughput of Deep Learning Services. *IEEE 37th International Conference on Computer Design*, pages 497–505, 2019.

[22] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Enable Simultaneous DNN Services Based on Deterministic Operator Overlap and Precise Latency Prediction. *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[23] ROCm documentation. GCN Native ISA LLVM Code Generator: Kernel Dispatch. https://rocmdocs.amd.com/en/latest/ROCm_Compiler_SDK/ROCm-Native-ISA.html, 2022.

[24] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. A Guide to Deep Learning in Healthcare. *Nature medicine*, 25(1):24–29, 2019.

[25] Jiarui Fang, Yang Yu, Chen liang Zhao, and Jie Zhou. TurboTransformers: An Efficient GPU Serving System for Transformer Models. *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

[26] AMD GPUOpen. AMD GPU ISA documentation. https://gpuopen.com/documentation/amd-isa-documentation, 2021.

[27] Chris Gregg, Jonathan Dorn, K. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *4th USENIX Workshop on Hot Topics in Parallelism*, HotPar'12, 2012.

[28] A. Gujarati, Reza Karimi, Safya Alzayat, 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, 2020.

[29] Johann Hauswald, Yiping Kang, Michael Laurenzano, Quan Chen, Cheng Li, Trevor N. Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. DjiNN and Tonic: DNN as A Service and Its Implications for Future Warehouse Scale Computers. *ACM/IEEE 42nd Annual International Symposium on Computer Architecture*, pages 27–40, 2015.

[30] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[31] Jeremy Hermann and Mike Del Balso. Meet Michelangelo: Uber's Machine Learning Platform. https://eng.uber.com/michelangelo-machine-learning-platform/, 2017.

[32] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdelrahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[33] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. GRNN: Low-Latency and Scalable RNN Inference on GPUs. *14th European Conference on Computer Systems*, 2019.

[34] Qingda Hu, J. Shu, Jie Fan, and Youyou Lu. Run-Time Performance Estimation and Fairness-Oriented Scheduling Policy for Concurrent GPGPU Applications. *45th International Conference on Parallel Processing*, pages 57–66, 2016.

[35] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely Connected Convolutional Networks. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2261–2269, 2017.

[36] Saksham Jain, Iljoo Baek, Shige Wang, and R. Rajkumar. Fractional gpus: Software-based compute and memory bandwidth reservation for gpus. *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 29–41, 2019.

[37] Won-Seok Jang, Hansaem Jeong, Kyungtae Kang, Nikil D. Dutt, and Jong-Chan Kim. R-TOD: Real-Time Object Detector with Minimized End-to-End Delay for Autonomous Driving. *IEEE Real-Time Systems Symposium*, pages 191–204, 2020.

[38] Ziheng Jiang. Schedule Primitives in TVM. https://tvm.apache.org/docs/how_to/work_with_schedules/schedule_primitives.html.

[39] Hyeonsu Lee, Hyunjune Kim, Cheolgi Kim, Hwansoo Han, and Euiseong Seo. Idempotence-Based Preemptive GPU Kernel Scheduling for Embedded Systems. *IEEE Transactions on Computers*, 70:332–346, 2021.

[40] Hyeonsu Lee, Jaehun Roh, and Euiseong Seo. A GPU Kernel Transactionization Scheme for Preemptive Priority Scheduling. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS'18, pages 202–213, 2018.

[41] TIMOTHY B. LEE. Tesla's autonomy event:Impressive progress with an unrealistic timeline. https://arstechnica.com/cars/2019/04/teslas-autonomy-event-impressive-progress-with-an-unrealistic-timeline/, 2019.

[42] LG Electronics Inc. Running Apollo 5.0 with SVL Simulator. https://www.svlsimulator.com/docs/system-under-test/apollo5-0-instructions/, 2022.

[43] Yun Liang, Huynh Phung Huynh, Kyle Rupnow, R. Goh, and Deming Chen. Efficient GPU Spatial-Temporal Multitasking. *IEEE Transactions on Parallel and Distributed Systems*, 26:748–760, 2015.

[44] Zhen Lin, L. Nyland, and Huiyang Zhou. Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching. *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 898–908, 2016.

[45] LLVM. User Guide for AMDGPU Backend. https://llvm.org/docs/AMDGPUUsage.html, 2021.

[46] Justin Luitjens. CUDA Streams—Best Practices and Common Pitfalls. http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf.

[47] Lingxiao Ma, Z. Xie, Zhi Yang, J. Xue, Youshan Miao, Wei Cui, W. Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'20, pages 881–897, 2020.

[48] Pavlo Molchanov, Shalini Gupta, Kihwan Kim, and Kari Pulli. Multi-sensor System for Driver's Hand-gesture Recognition. *11th IEEE International Conference and Workshops on Automatic Face and Gesture Recognition*, 1:1–8, 2015.

[49] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. Accelerating Deep Learning Workloads Through Efficient Multi-model Execution. In *NeurIPS Workshop on Systems for Machine Learning*, page 20, 2018.

[50] NVIDIA. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[51] NVIDIA. NVIDIA Tesla P100. http://www.nvidia.com/object/pascal-architecture-whitepaper.html, 2016.

[52] NVIDIA. CUDA Toolkit: Develop, Optimize and Deploy GPU-Accelerated Apps. https://developer.nvidia.com/cuda-toolkit, 2021.

[53] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

[54] Nathan Otterness and James H. Anderson. AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads. In *32nd Euromicro Conference on Real-Time Systems*, ECRTS'20, 2020.

[55] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU Concurrency with Elastic Kernels. In *Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, 2013.

[56] J. Park, Yongjun Park, and S. Mahlke. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. *Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[57] J. Park, Yongjun Park, and S. Mahlke. Dynamic Resource Management for Efficient Utilization of Multitasking GPUs. *Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[58] Reid Pinkham, Andrew Berkovich, and Zhengya Zhang. Near-Sensor Distributed DNN Processing for Augmented and Virtual Reality. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2021.

[59] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[60] Steve Rennich. CUDA C/C++ Streams and Concurrency. https://developer.download.nvidia.cn/CUDA/training/StreamsAndConcurrencyWebinar.pdf.

[61] ROCm Core Technology. AMD GPU kernel driver with KFD. https://github.com/RadeonOpenCompute/ROCK-Kernel-Driver, 2022.

[62] ROCm Core Technology. AMD GPU kernel driver with KFD: unmap_queues_cpsch. https://github.com/RadeonOpenCompute/ROCK-Kernel-Driver/blob/master/drivers/gpu/drm/amd/amdkfd/kfd_device_queue_manager.c, 2022.

[63] ROCm Developer Tools. Hip: C++ heterogeneous-compute interface for portability. https://github.com/ROCm-Developer-Tools/HIP, 2022.

[64] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated Model-less Inference Serving. In *USENIX Annual Technical Conference*, ATC'21, pages 397–411, 2021.

[65] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Mārtiņš Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, Eugene Agafonov, Tae Hyung Kim, Eric Sterner, Keunhae Ushiroda, Michael Reyes, Dmitry Zelenkovsky, and Seonman Kim. LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving. In *IEEE 23rd International Conference on Intelligent Transportation Systems Conference*, ITSC'20, pages 1–6, 2020.

[66] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, A Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. *CoRR*, abs/1910.01108, 2019.

[67] 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. *27th ACM Symposium on Operating Systems Principles*, 2019.

[68] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-scale Image Recognition. *CoRR*, abs/1409.1556, 2014.

[69] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[70] I. Tanasić, Isaac Gelado, Javier Cabezas, A. Ramírez, N. Navarro, and M. Valero. Enabling Preemptive Multiprogramming on GPUs. *ACM/IEEE 41st International Symposium on Computer Architecture*, pages 193–204, 2014.

[71] TESLARATI. AMD confirms Tesla's new Model S and Model X will boast RDNA 2 GPUs. https://www.teslarati.com/tesla-model-s-model-x-mcu3-specs-amd-gpu-confirmed-video/, 2021.

[72] Apache TVM. A test suite of DNN models. https://github.com/apache/tvm/tree/v0.8/python/tvm/relay/testing, 2021.

[73] Apache TVM. Apache TVM: An End to End Machine Learning Compiler Framework for CPUs, GPUs and accelerators. https://tvm.apache.org/, 2021.

[74] Guibin Wang, Yisong Lin, and Wei Yi. Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU. *IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350, 2010.

[75] Yuxuan Wang, RJ Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, Samy Bengio, et al. Tacotron: A Fully End-to-end Text-to-speech Synthesis Model. *arXiv preprint arXiv:1703.10135*, 2017.

[76] Zhenning Wang, J. Yang, R. Melhem, B. Childers, Youtao Zhang, and M. Guo. Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-grained Sharing. *IEEE International Symposium on High Performance Computer Architecture*, pages 358–369, 2016.

[77] Bo Wu, Xu Liu, Xiaobo Zhou, and C. Jiang. Flep: Enabling flexible and efficient preemption on gpus. *Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[78] Yecheng Xiang and Hyoseung Kim. Pipelined Data-Parallel CPU/GPU Scheduling for Multi-DNN Real-Time Inference. *IEEE Real-Time Systems Symposium*, pages 392–405, 2019.

[79] Feng Yan, Yuxiong He, Olatunji Ruwase, and Evgenia Smirni. Efficient Deep Neural Network Serving: Fast and Furious. *IEEE Transactions on Network and Service Management*, 15:112–126, 2018.

[80] Ming Yang, Shige Wang, Joshua Bakita, Thanh Vu, F. Donelson Smith, James H. Anderson, and Jan-Michael Frahm. Re-Thinking CNN Frameworks for Time-Sensitive Autonomous-Driving Applications: Addressing an Industrial Challenge. *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 305–317, 2019.

[81] Wai Chee Yau. How Zendesk Serves TensorFlow Models in Production. `https://zendesk.engineering/how-zendesk-serves-tensorflow-models-in-production-751ee22f0f4b`, 2017.

[82] T. Yeh, Matthew D. Sinclair, Bradford M. Beckmann, and Timothy G. Rogers. Deadline-Aware Offloading for High-Throughput Accelerators. *IEEE International Symposium on High-Performance Computer Architecture*, pages 479–492, 2021.

[83] Juheon Yi and Youngki Lee. Heimdall: mobile gpu coordination platform for augmented reality applications. In *26th Annual International Conference on Mobile Computing and Networking*, MobiCom'20, pages 1–14, 2020.

[84] Sebastian Zepf, Javier Hernandez, Alexander Schmitt, Wolfgang Minker, and Rosalind W. Picard. Driver Emotion Recognition for Intelligent Vehicles. *ACM Computing Surveys*, 53:1–30, 2020.

[85] Hengyu Zhao, Yubo Zhang, Pingfan Meng, Hui Shi, Erran L. Li, Tiancheng Lou, and Jishen Zhao. Towards Safety-Aware Computing System Design in Autonomous Vehicles. *ArXiv*, abs/1905.08453, 2019.

[86] Wenyi Zhao, Quan Chen, and M. Guo. KSM: Online Application-Level Performance Slowdown Prediction for Spatial Multitasking GPGPU. *IEEE Computer Architecture Letters*, 17:187–191, 2018.

[87] Wenyi Zhao, Quan Chen, H. Lin, Jianfeng Zhang, Jingwen Leng, C. Li, Wenli Zheng, Linlin Li, and M. Guo. Themis: Predicting and Reining in Application-Level Slowdown on Spatial Multitasking GPUs. *IEEE International Parallel and Distributed Processing Symposium*, pages 653–663, 2019.

[88] Xia Zhao, Magnus Jahre, and L. Eeckhout. HSM: A Hybrid Slowdown Model for Multitasking GPUs. *Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[89] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor : Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'20, 2020.

[90] H. Zhou, G. Tong, and Cong Liu. GPES: A Preemptive Execution System for GPGPU Computing. *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 87–97, 2015.

[91] Husheng Zhou, Soroush Bateni, and Cong Liu. S3DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads. *2018 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 190–201, 2018.

A Artifact Appendix

This artifact provides the source code of REEF, a detailed readme, and scripts to reproduce the main experimental results from the OSDI 2022 paper—"Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences" by M. Han, H. Zhang, R. Chen, and H. Chen. REEF is the first GPU-accelerated DNN inference serving system that enables microsecond-scale kernel preemption and controlled concurrent execution in GPU scheduling. We provide instructions to build the software package and run experiments. Our artifact obtained the "Artifacts Available", "Artifacts Functional" and "Artifacts Reproduced" badges from the Artifact Evaluation process of OSDI 2022. The DOI of our artifact is `https://doi.org/10.5281/zenodo.6586106`.

Artifact repository. All project source code, including full instructions on how to build and run the main experiments on REEF and benchmarks is available in the following git repository: `https://github.com/SJTU-IPADS/reef-artifacts/tree/osdi22-ae`.