



# Real-time, Work-conserving GPU Scheduling for Concurrent DNN Inference

**MINGCONG HAN**, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China

**RONG CHEN**, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China

**WEIHANG SHEN**, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China

**HANZE ZHANG**, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China

**JINRONG YANG**, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China

**HAIBO CHEN**, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China and China and Key Laboratory of System Software, Chinese Academy of Sciences, Beijing, China

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

This article presents REEF, the first GPU-accelerated DNN inference serving system that achieves low-latency and work-conserving for concurrent real-time and best-effort tasks. REEF accomplishes this by enabling 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 are mostly idempotent, REEF devises a reset-based preemption scheme that launches a real-time kernel on the GPU by proactively killing and restoring

This article extends our prior conference paper “Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences” presented at the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI’22) [38]. Our significant extensions include: a complete implementation and evaluation of REEF on NVIDIA GPUs; an improved preemption mechanism for non-idempotent kernels through our novel multi-kernel idempotence concept; a multi-version kernel selection technique to optimize kernel padding and improve throughput; and a comprehensive evaluation comparing REEF against state-of-the-art systems using modern workloads.

This work was supported in part by the National Natural Science Foundation of China (No. 62432010, 62272291, 62132014). Authors’ Contact Information: Mingcong Han, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China; e-mail: mingconghan@sjtu.edu.cn; Rong Chen (corresponding author), Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China; e-mail: rongchen@sjtu.edu.cn; Weihang Shen, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China; e-mail: shenwhang@sjtu.edu.cn; Hanze Zhang, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China; e-mail: hanzezhang@sjtu.edu.cn; Jinrong Yang, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China; e-mail: coyangjr@sjtu.edu.cn; Haibo Chen, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China, China and Key Laboratory of System Software and Chinese Academy of Sciences, Beijing, China; e-mail: haibochen@sjtu.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 0734-2071/2025/11-ART1

<https://doi.org/10.1145/3768622>

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 both NVIDIA and AMD GPUs shows that REEF only incurs less than 5% overhead in end-to-end latency for real-time tasks but increases the overall throughput by up to 1.53 $\times$ , compared to scheduling tasks sequentially. To demonstrate the practical benefits of our approach, we compare REEF with Triton, a widely-adopted production-level serving system. Our evaluation shows that REEF outperforms Triton by 1.12 $\times$  to 5.20 $\times$  in end-to-end latency for real-time tasks, while maintaining comparable throughput.

CCS Concepts: • **Software and its engineering** → **Scheduling**; • **Computing methodologies** → **Concurrent computing methodologies**;

Additional Key Words and Phrases: GPU scheduling, kernel preemption, kernel padding, DNN inference

#### ACM Reference Format:

Mingcong Han, Rong Chen, Weihang Shen, Hanze Zhang, Jinrong Yang, and Haibo Chen. 2025. Real-time, Work-conserving GPU Scheduling for Concurrent DNN Inference. *ACM Trans. Comput. Syst.* 44, 1, Article 1 (November 2025), 42 pages. <https://doi.org/10.1145/3768622>

## 1 Introduction

**Deep Neural Network (DNN)** inference has been widely adopted by modern intelligent applications, such as autonomous driving [2, 46, 54, 108], virtual reality [78, 112], speech/image recognition [42, 102], and healthcare [23, 30], 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 [17, 43, 62, 120].

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 [12, 108, 121], it is hard to fully exploit the massive parallelism of the GPU [62]. Hence, it is a common practice to share a GPU among multiple applications with different timing constraints in emerging intelligent systems [54, 105, 107], which can greatly improve overall throughput, as shown in Figure 1(a). For example, autonomous vehicles use DNNs to recognize obstacles and traffic lights [11, 36, 81], which are latency-critical tasks (called *real-time* tasks in this article). Meanwhile, other tasks with no hard real-time requirement [105] (called *best-effort* tasks in this article), such as monitoring human driver's emotion and fatigue, are also served within the GPU using DNNs [23, 65, 113].

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 [71] and ROCm [3]) commonly provide multiple GPU streams (e.g., CUDA Streams [83]) to concurrently execute multiple tasks on the same GPU. However, as shown in Figure 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 [116].

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 [96]. Prior work [14, 104, 122] 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

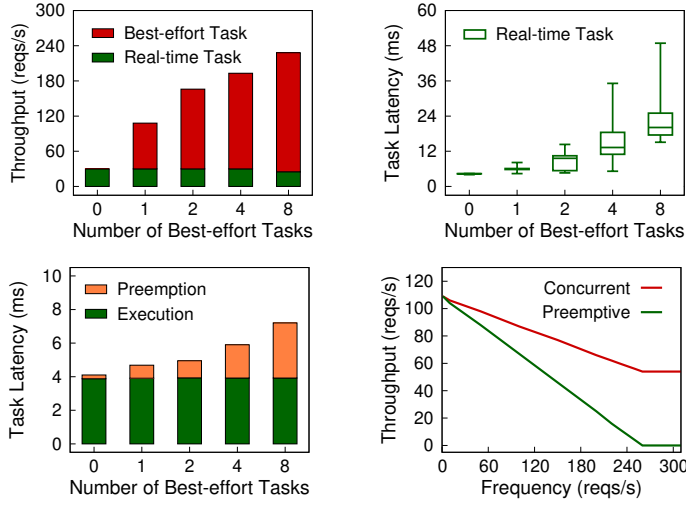


Fig. 1. (a) Overall throughput of DNN inferences (both real-time and best-effort tasks); (b) End-to-end latency of real-time tasks using concurrent GPU scheduling (i.e., multiple GPU streams [61, 66, 83]); (c) End-to-end latency of real-time tasks using preemptive GPU scheduling (i.e., wait-based preemption [14, 104, 122]); (d) Throughput of best-effort tasks as real-time task frequency increases. **Workload:** VGG [94] (real-time) and ResNet [40] (best-effort). **Testbed:** One AMD Radeon Instinct MI50 GPU with 16 GB of memory (see Section 6 for details).

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 Figure 1(c). Further, when the real-time inference requests arrive at a high frequency (e.g., camera (120 reqs/s) [19] or multiple sensors [54]), the best-effort tasks may even get starved, as shown in Figure 1(d).

This article 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 without saving contexts for an instant preemption, and restored by simply re-executing the kernel. 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. To handle the small fraction of non-idempotent kernels, REEF introduces the concept of *multi-kernel idempotence*, which groups consecutive kernels into atomic units. When a non-idempotent kernel is preempted, REEF restores its state by re-executing the entire group that contains the kernel. With such a preemption mechanism, 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. Furthermore, 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 initially implemented REEF on AMD GPUs (e.g., Radeon Instinct MI50) and subsequently ported it to NVIDIA GPUs (e.g., GV100). REEF extends Apache TVM [98], a deep learning compiler, to generate customized GPU kernels for DNN inferences, implementing reset-based preemption and dynamic kernel padding mechanisms. Our reset-based preemption scheme has influenced XSched [91], a preemptive scheduling framework for diverse XPU, enabling efficient preemption of both pending and running kernels.<sup>1</sup> 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 [8] (an open autonomous driving platform). The evaluation conducted on AMD GPUs shows that REEF only incurs less than 5% end-to-end latency overhead for real-time tasks while achieves up to a 1.53× throughput improvement compared to scheduling tasks sequentially. 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 on both AMD and NVIDIA GPUs. To demonstrate the feasibility of our approaches, we conducted evaluations of REEF on an NVIDIA GPU using inference engine with closed-source GPU kernels (e.g., TensorRT [68]). REEF outperforms Triton [22], a production-level serving system, by 1.12× to 5.20× in end-to-end latency for real-time tasks, while maintaining comparable throughput.

**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 (Section 2).
- A novel reset-based preemption scheme that launches a real-time kernel on the GPU in a few microseconds, regardless of the number of preempted kernels (Section 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 (Section 5).
- A prototype implementation for both AMD and NVIDIA GPUs and an evaluation that demonstrates the advantage and efficacy of REEF over state-of-the-art approaches (Section 6).

The source code of REEF is publicly available at <https://github.com/SJTU-IPADS/reef>. The DISB framework can be obtained separately from <https://github.com/SJTU-IPADS/dish>.

## 2 Background and Motivation

### 2.1 Characterizing GPU-Accelerated DNN Inference

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 [24, 35, 87]. To serve inference requests on GPUs, the pre-trained DNN model (e.g., ResNet [40]) is loaded into GPU memory ahead of time. Figure 2 outlines the implementation of GPU-accelerated DNN inference. For each arriving request, all kernels of the DNN 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 [11, 81], and *best-effort* (BE) tasks, such as emotion and fatigue monitoring [23, 65, 113]. 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

<sup>1</sup>XSched is publicly available at <https://github.com/XpuOS/xsched>

```

# device codes
__global__ void conv_relu(in, weight, out):
1  sum = 0;
2  for i in range(0,3)
3      for j in range(0,3)
4          sum += in[..] * weight[..]
5  out[..] = ReLU(sum)

__global__ void dense(in, weight, bias, out):
6  sum = 0;
7  for i in range(0,512)
8      sum += in[..] * weight[..]
9  out[..] = sum + bias[..]

# host codes
void inference(...):
10 memcpH2D(in, in_host, in_sz) # copy in to GPU
11 conv_relu <<<dim(32), ...>>> (in, .., buf_conv)
12 ... # launch other kernels
13 pooling <<<dim(64), ...>>> (.., buf_pool)
14 dense <<<dim(10), ...>>> (buf_pool, .., buf_dense)
15 softmax <<<dim(1), ...>>> (buf_dense, .., out)
16 memcpD2H(out_host, out, out_sz) # copy out to CPU

```

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

Table 1. Number of Idempotent GPU Kernels and Total GPU Kernels for DNN Models Evaluated in Section 6 Across Different Inference frameworks (TVM [98], TensorRT [68], and PyTorch [77])

Model	#Idem Kernels / #Total Kernels			Execution Time
	TVM	TensorRT	PyTorch	
ResNet-152 [40] (RNET)	207/207	67/103	145/211	9.1
VGG-19 [94] (VGG)	71/ 71	30/ 30	45/ 81	3.7
Yolov3 [82] (YOLO)	144/144	153/175	569/572	10.0
Bert-Base [28] (BERT)	393/393	161/173	165/238	8.1
GPT-2 [80] (GPT2)	440/440	136/148	258/294	13.2

Their execution times (in milliseconds) are measured on an AMD Radeon Instinct MI50 GPU using TVM.

periodically at various frequencies by input sensors (e.g., camera and LiDAR [8, 54]). 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 most GPU kernels in DNN models are *idempotent* as they primarily perform dense linear algebra computations without side effects. Through our analysis of five DNN models across three major inference frameworks, we found that all kernels generated by TVM [98] are idempotent, while 1182 out of 1396 kernels from PyTorch [77] and 547 out of 629 kernels from TensorRT [68] are idempotent, as detailed in Table 1. The idempotence property ensures that a kernel will always produce identical output given the same input, regardless of whether it has been retried or not. This characteristic enables efficient preemption of DNN inference tasks by simply killing the running kernel and restoring execution by re-starting it. For the small fraction of non-idempotent kernels, we note that each kernel in a DNN model only depends on outputs from previous kernels and static arguments (e.g., weights) as inputs. For instance, the dense kernel in Figure 2 takes the output of the pooling kernel and a

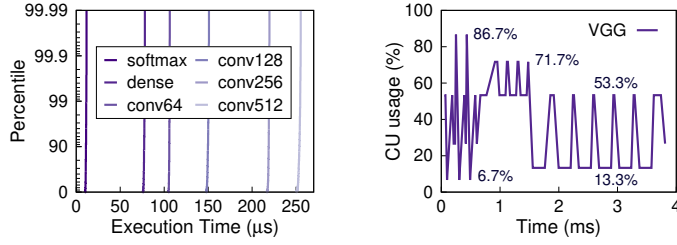


Fig. 3. (a) The CDF of execution time (in  $\mu\text{s}$ ) for several typical kernels in VGG, and (b) the timeline of CU usage during VGG execution on a GPU with 60 compute units (CUs). The execution time of GPU kernels in VGG ranges from 10  $\mu\text{s}$  to 365  $\mu\text{s}$  (see Figure 13).

weight matrix as inputs. This dependency structure allows us to safely restore a preempted DNN inference task by re-executing it from the previous kernel, even if the preemption occurred during a non-idempotent kernel's execution.

**Massive kernels.** Unlike traditional GPU applications that only contain a few kernels (e.g., at most 14 kernels in Rodinia [13]), 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 advance to hide the lengthy kernel launching time. Furthermore, 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 Figure 3(a)). This is also confirmed in recent literature [7, 35, 62].

**Varied parallelism.** The GPU kernels in DNN inferences usually exhibit completely different parallelism due to varied input scales. For example, as shown in Figure 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, Figure 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.

## 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, 10, 14, 15, 34, 58, 104, 111, 119], 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 Figure 4.

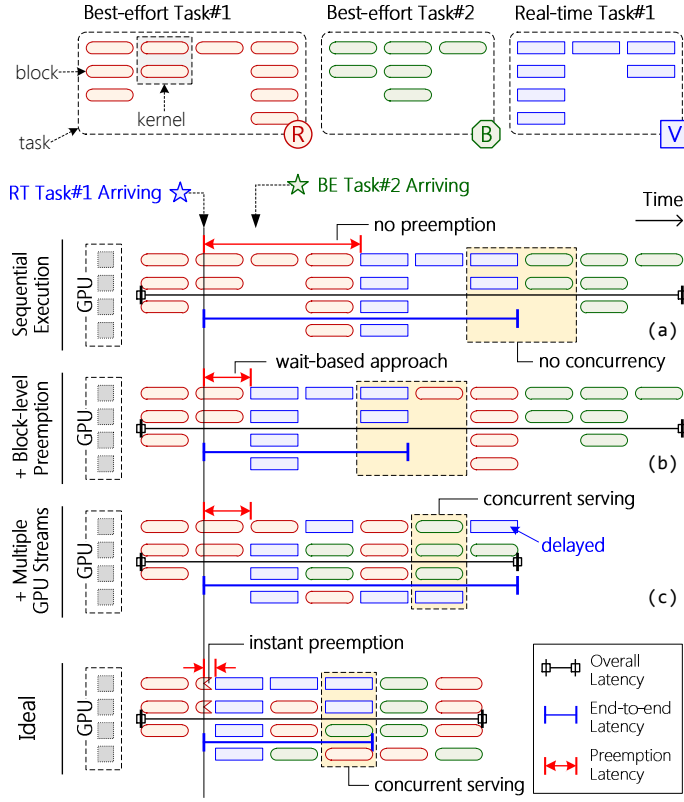


Fig. 4. An example of GPU task scheduling using different kernel preemption and parallelism schemes for a hybrid workload, which contains two best-effort and one real-time DNN inference tasks. The GPU has four compute units (CUs).

**Sequential execution.** Most existing DNN serving systems, such as Clockwork [35], use sequential execution to avoid interferences among tasks. Thus, each task can achieve optimal execution latency, as shown in Figure 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) [75, 96]. Meanwhile, commodity GPUs also lack hardware support for the preemption mechanism.<sup>2</sup> As a compromise, prior work [10, 122] 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 Figure 4(b). Further, the preemption latency will increase with the number of preempted kernels (see Figure 1(c)). As a compromise, prior work [10, 122] has to limit the number of kernels submitted to the GPU, which is impractical for DNN inferences. Further, a

<sup>2</sup>Although NVIDIA claims that their GPUs have been equipped with preemption support since Pascal architecture [70], there is no publicly available information or a software controllable interface [14, 53, 104].

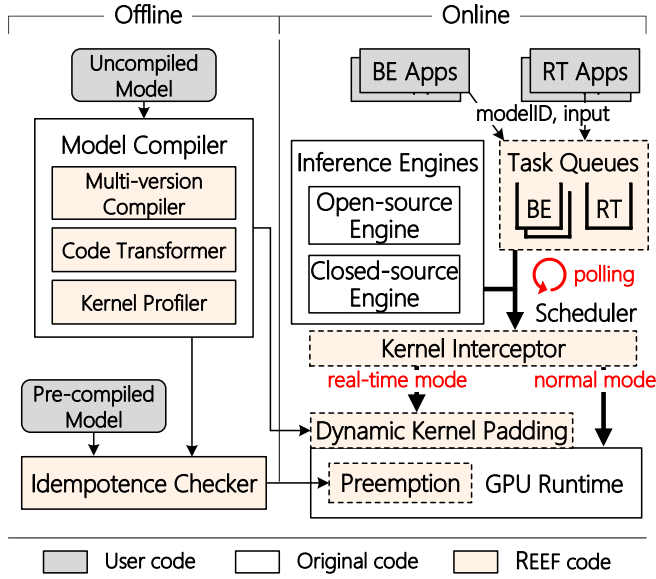


Fig. 5. *Architecture of REEF*. Modules in boxes with dashed border are on the critical path of serving DNN inference requests. Other modules do not directly affect serving latency and throughput.

high-frequency real-time task will break the execution of best-effort tasks, even leading to starvation (see Figure 1(d)).

**Multiple GPU streams.** To improve overall throughput, modern GPU libraries (e.g., CUDA [71] and ROCm [3]) commonly provide multiple GPU streams (e.g., CUDA Streams [83]) 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 Figure 4(c). Although leveraging multiple GPU streams can improve throughput (see Figure 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 Figure 4(c). Even worse, the latency overhead will increase with the number of concurrent tasks (see Figure 1(b)).

### 3 REEF Overview

#### 3.1 System Architecture

The goal of REEF is to provide preemptive GPU scheduling that achieves real-time guarantees for latency-critical tasks while remaining work-conserving for best-effort tasks (see ideal scheduling example in Figure 4). Leveraging the insight that DNN inference kernels are mostly idempotent and consists of numerous kernels with varied parallelism and predictable latency, REEF introduces two novel designs: reset-based preemption and dynamic kernel padding.

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

**DNN model preparation (offline).** REEF supports both uncompiled DNN models (e.g., in ONNX [6] format) and pre-compiled DNN models (e.g., in TensorRT [68] format) with closed-source GPU kernels. At the offline stage, REEF performs two tasks: (a) compiles the uncompiled models to generate GPU kernels, and (b) checks the idempotence of all kernels for both types of models.

*Model compiler.* REEF compiles models using a customized model compiler (TVM [17]) with three extensions: (a) a multi-version compiler that generates multiple GPU kernel variants for each DNN layer, each with different performance characteristics (e.g., execution time, register usage); (b) a code transformer module that converts GPU kernels to be compatible with our dynamic kernel padding technique; and (c) a kernel profiler that measures computational requirements and execution time for each kernel.

*Idempotence checker.* Reset-based preemption requires knowledge of kernel idempotence (whether kernels can be safely restarted). REEF pre-executes all kernels (both pre-compiled and those compiled using our model compiler) and traces their execution to check idempotence. This information is then used by the preemption module at runtime to ensure safe kernel preemption and restoration.

**DNN inference serving (online).** REEF provides an RPC-based interface for DNN-based applications to submit inference requests. REEF supports serving DNN inference requests using either open-source inference engines (e.g., TVM Runtime [98]) or closed-source inference engines (e.g., TensorRT [68]). REEF consists of four major components:

*Task queues.* REEF maintains one real-time task queue and several best-effort task queues. Each queue is bound to a GPU stream for executing inference requests, and requests are processed in 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 [12], can be adopted by REEF for real-time requests.

*Scheduler.* The scheduler in REEF uses busy polling on task queues to retrieve requests and then invokes the associated inference engine to execute these tasks on their designated 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.

*Kernel interceptor.* REEF intercepts GPU kernels launched by inference engines through intercepting the GPU runtime API calls, enabling it to schedule tasks at the granularity of GPU kernels without directly manipulating the inference engines themselves. This approach makes REEF more flexible and easier to extend to support additional inference engines, even closed-source ones like TensorRT. The interceptor identifies which request a kernel belongs to by checking the GPU stream it is launched with.

*Preemption module.* In normal mode, REEF concurrently serves best-effort tasks from different task queues using multiple GPU streams [3, 83] provided by GPU runtime. In real-time mode, REEF first uses the preemption module to instantly preempt all running best-effort tasks (Section 4) and then immediately launches the real-time task on the GPU. The preemption module works for both open-source and closed-source inference engines, as well as pre-compiled and uncompiled models.

*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 (Section 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. Notably, DKP is only applicable to kernels that are compiled with our extended model compiler.

### 3.2 An Illustrative Example

Figure 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

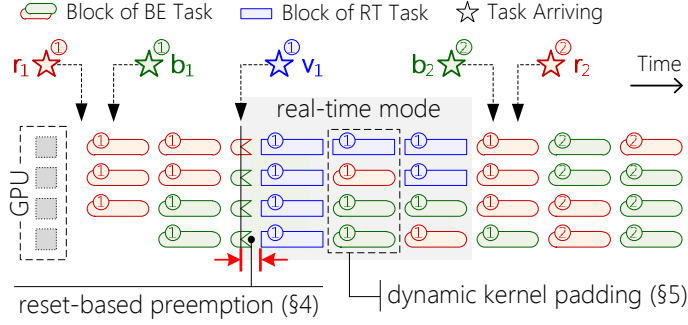


Fig. 6. An example timeline in REEF. The DNN inference tasks are identical to those in Figure 4.

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) [96]. Second, there is no need to wait for all running kernels to complete, which can take hundreds of microseconds or even milliseconds.

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 asynchronously launched kernels are buffered in the 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.

##### 4.1 Abstracting Queues in GPU Runtime

To understand the kernel lifetime and locations of launched kernels, we first propose an abstraction of the queues that buffer the kernels in GPU runtime, and later explain how this abstraction maps to the commodity GPU runtime in Section 4.5 for AMD GPUs and Section 4.6 for NVIDIA GPUs.

As illustrated in Figure 7, each inference task is associated with a GPU stream, and all kernels of the task are launched to its corresponding GPU stream by the scheduler through the GPU stream API. The launched kernels are first buffered in a *host queue* (HQ). Each GPU stream has its own host queue. The primary function of the host queue is to ensure that each GPU stream API call is asynchronous, allowing it to return immediately after being called. Therefore, the host queue is often implemented as a linked list with unlimited capacity. Each host queue has a background thread that transmits the buffered kernels asynchronously to a **device queue** (DQ). The DQ can be accessed by both the CPU and the GPU simultaneously, with the CPU responsible for pushing new kernels to the back of the queue, while the GPU continuously pops kernels from the front

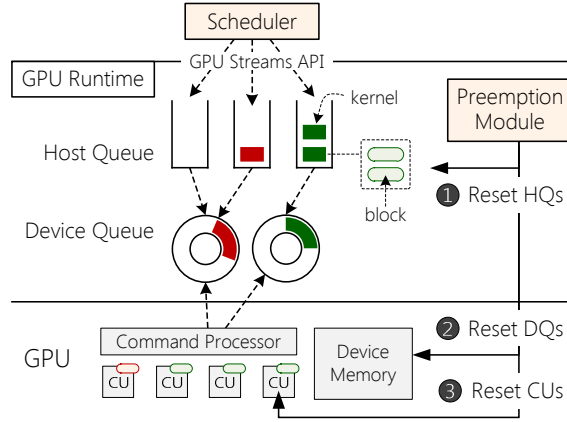


Fig. 7. Queue abstraction for the GPU runtime and preemption module in REEF.

of the queue. The DQ is often implemented as a ring buffer with a fixed capacity to simplify the communication and synchronization between the CPU and the GPU. The kernels in the DQ are fetched by the GPU's command processor and eventually dispatched to the *compute units* (CUs) for execution. Therefore, launched kernels of an inference task may exist in three places, namely host queues (HQs), DQs (DQs), and compute units (CUs). To achieve instant preemption, kernels in all three places must be evicted. Figure 8 shows the pseudocode for the preemption module in REEF.

#### 4.2 Evicting Buffered Kernels

The reset-based approach requires proactively evicting all buffered kernels from both host queues and DQs. For host queues, it is straightforward to reset them (❶ in Figure 7), dequeuing all buffered kernels and reclaiming memory, as they are fully controlled by the CPU code. For DQs, however, the GPU runtime cannot evict buffered kernels from DQs, because the command processor of GPU can directly fetch kernels from DQs [29], resulting in data races and unpredictable results. In addition, the CPU also does not provide a way to safely evict kernels from DQs. A potential solution is to notify the GPU to re-register a new DQ [85]. However, it would incur an unacceptable latency overhead (e.g., about 1 ms on our testbed).

Inspired by evictable kernels [14], we propose *lazy eviction* to reset DQs without extending GPU runtime and hardware. The code transformer of REEF injects a piece of code at the beginning of each kernel in advance (lines 4 in Figure 8), which checks a *preemption flag* to realize whether it has been evicted. When the preemption flag is true, the kernel will voluntarily terminate itself (lines 1–2 in Figure 8). Therefore, when a preemption occurs, the preemption module will immediately set the preemption flag to true in GPU memory (see ❷ in Figure 7). The kernels buffered in DQs 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  $\mu$ s to preempt a single task (see Section 6.4). 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 DQ. 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 **garbage collection (GC)** thread to reclaim memory

---

```

# device codes
__device__ void preemptor(p_args):
1  if (*p_args.flag == true)
2      terminate() # lazy eviction
3      *p_args.global_idx = p_args.curr_idx

__global__ void dense(in, weight, bias, out, p_args):
4  preemptor(p_args)
5  ... # run kernel body

# host codes
void inference(model, p_args, start_idx, stream):
6  p_args.curr_idx = 0
7  ... # launch other kernels
8  p_args.curr_idx += 1 # increment execution index
   # skip this kernel if it has been executed
9  if (p_args.curr_idx >= start_idx) then
   # launch the kernels with preemption arguments
10     dense<<<..., stream>>> (... , p_args)
11 ... # launch other kernels

void preempt(p_args, stream):
12 reset_hq(stream) # ❶ reset HQs
13 set_flag(p_args.flag, true) # ❷ reset DQs
14 reset_cu(stream) # ❸ reset CUs

void restore(model, p_args, stream):
   # find the last executed kernel
15 last_idx = load_idx(p_args.global_idx)
16 set_flag(p_args.flag, false) # reset the preemption flag
   # find the entry kernel with multi-kernel idempotence
17 start_idx = find_idem_entry(model, last_idx)
   # re-launch the non-executed kernels
18 inference(model, p_args, start_idx, stream)

```

---

Fig. 8. Pseudocode for reset-based preemption in REEF.

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.

**DQ capacity restriction.** Although using lazy eviction can terminate kernels in the DQ immediately at the beginning of execution, the kernels still have to be fetched and dispatched to the CU, which takes around 20  $\mu$ s per kernel. It is common to buffer hundreds of kernels in a DQ, since it can reduce the frequency of CPU thread context switches by filling up the DQ 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 DQ to achieve microsecond-scale kernel preemption. Tuning the DQ 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 DQs with the kernels from host queues. We empirically choose a DQ capacity to 4 on our testbed, since it is sufficient to reset the DQ in 30  $\mu$ s with negligible overhead on normal execution time (i.e., less than 0.3%). Furthermore, using a smaller DQ also produces slightly higher CPU utilization (e.g., about 15% increase) due to more frequent filling of the DQ.

### 4.3 Killing Running Kernels

To avoid waiting for the completion of running kernels, the reset-based preemption proactively kills the running kernels in the GPU (see ❸ in Figure 7). We observed that modern GPUs have

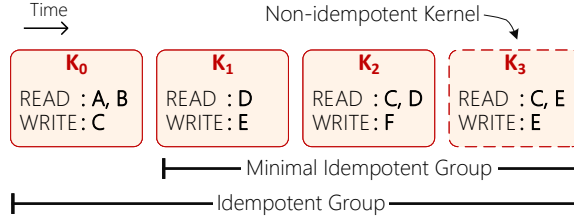


Fig. 9. An example of multi-kernel idempotence.

the ability to terminate CPU process and also kill associated GPU kernels, even when the kernel sticks in an infinite loop. It implies that GPU driver can indeed interrupt and kill an uncompleted kernel. Therefore, REEF retrofits such mechanism and exposes it to the preemption module in GPU runtime.

#### 4.4 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 by only re-executing the interrupted kernel. However, there are still some challenges to benefit from the idempotence property. First, because the kernel running on the CUs is killed by the command processor of GPU, which is asynchronous to the CPU, the scheduler cannot directly identify which kernel in the DQ was interrupted. Second, even if most kernels in DNN models are idempotent, there are still some non-idempotent kernels that cannot be restored by simply re-executing them. For example, we observed some kernels in TensorRT and PyTorch are non-idempotent (shown in Table 1). The existence of non-idempotent kernels is primarily due to in-place tensor operations, where a kernel both reads from and writes to the same memory location. For example, when a kernel performs an in-place addition (e.g., adding a constant to a tensor), it needs to read the original tensor value and then write the updated value back to the same memory location. This read-after-write pattern makes the kernel non-idempotent because re-executing it would produce different results depending on whether the memory location has been modified by previous executions.

**Kernel execution index.** To address the first challenge, REEF assigns a unique *execution index* to each kernel when it is launched (line 8 in Figure 8), and transforms the kernel code to update a global execution index variable using its execution index at the beginning of each kernel (line 3 in Figure 8). When restoring a preempted task, the preemption module reads the global execution index variable to determine the kernel that was interrupted (line 15 in Figure 8). If the interrupted kernel is idempotent, the preemption module restores the task by re-executing the interrupted kernel and launching the subsequent kernels.

**Multi-kernel idempotence.** To address the second challenge, REEF extends the application of idempotence from a single kernel to the use of multi-kernel idempotence. An *idempotent group* consists of consecutively executed kernels that are idempotent as a whole, meaning that no matter at which kernel the execution is interrupted, it can be restarted from the first kernel of the group to achieve the same execution result as a single uninterrupted execution. An idempotent group does not require each kernel to have single-kernel idempotence. In other words, even if a single kernel is non-idempotent, it is still possible to find an idempotent group, including the non-idempotent one, that satisfies the multi-kernel idempotence property. In the worst case, this idempotent group includes all kernels in the DNN model. Figure 9 illustrates an example of multi-kernel idempotence

with four consecutive executed kernels. Although K3 is non-idempotent, the entire group (K0, K1, K2, K3) is still idempotent. If K3 is killed during the preemption, the task can still be restored by re-executing from K0, which is the first kernel of the idempotent group to which K3 belongs.

To reduce restoration overhead, REEF must identify the kernel group with minimal number of kernels that satisfies multi-kernel idempotence for each non-idempotent kernel, thereby minimizing the number of kernels that need re-execution. For example, in Figure 9, both the group (K0, K1, K2, K3) and (K1, K2, K3) satisfy multi-kernel idempotence, but the former is smaller and takes less time to restore.

Checking multi-kernel idempotence. REEF performs this identification in two offline steps REEF currently assumes that the idempotence property of GPU kernels can be determined statically through offline trace analysis and remains constant at runtime [53, 55, 75]. However, REEF can also integrate recently proposed approaches and tools [37] for dynamic idempotence validation, thereby relaxing the static assumption. First, it executes the DNN model to generate a trace of the memory addresses accessed by each GPU operations, including GPU kernels, memory copy, and memory set. Then, it searches for the minimal group that satisfies multi-kernel idempotence for each kernel. The search algorithm begins with the target kernel and iteratively adds predecessor kernels until multi-kernel idempotence is achieved. The condition for multi-kernel idempotence is that all bytes on GPU memory accessed by the group must be free from clobber-anti-dependencies [27]. Specifically, for a kernel group to be idempotent, each memory location it accesses must satisfy one of two conditions: either (1) it is only read from (never written to) during the group's execution, or (2) if it is written to, the first access to that location within the group must be a write operation. In other words, the input of a group of kernels must not be overwritten.

To illustrate how REEF checks multi-kernel idempotence, consider the example in Figure 9 with four consecutive kernels (K0–K3) and six memory locations (A–F). Note that each memory location represents a single byte in memory. We next show how the algorithm finds the minimal idempotent group for K3. The algorithm starts with K3, which reads C, E and writes E. K3 is non-idempotent because it both reads and writes E, meaning the input E might be overwritten during its execution. The algorithm then adds K3's predecessor K2 to form the group (K2, K3). This group is still non-idempotent because the group still treats E as both input and output. The algorithm continues by adding K1, forming the group (K1, K2, K3). This group is idempotent because its inputs (C, D) are distinct from its outputs (E, F). The algorithm stops here as it has found the minimal idempotent group for K3.

We validated the idempotence property of the kernels across five DNN models, as detailed in Table 1. Our analysis encompassed both TVM-generated kernels and those from TensorRT [68] and PyTorch [77]. The results revealed that all TVM-generated kernels are idempotent, while TensorRT and PyTorch show the presence of non-idempotent kernels: 82 out of 629 kernels in TensorRT and 214 out of 1,396 kernels in PyTorch are non-idempotent, leaving 547 and 1,182 idempotent kernels respectively. For the non-idempotent kernels, we applied our iterative algorithm to identify their minimal idempotent groups. As shown in Figure 10, these groups are predominantly small, with most containing 1–4 kernels. This small group size ensures that restoring non-idempotent kernels requires re-executing only a few additional kernels (less than 5 for most cases), resulting in minimal restoration overhead (see Section 6.4 and Figure 22). Interestingly, many groups contain only one kernel because they included other GPU operations (e.g., memory set) that writes to the memory locations used by the non-idempotent kernel. The largest identified group contains 9 kernels.

The results of this offline analysis are saved in a metadata file that maps each kernel to its idempotence property and group information. When REEF loads a DNN model, the preemption module reads this metadata file to build an in-memory lookup table for runtime use.

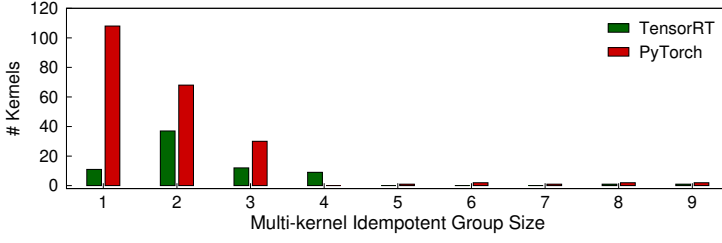


Fig. 10. The number of kernels in the minimal idempotent group containing non-idempotent kernels for DNN models in Table 1.

**Restoring idempotent kernel group.** After identifying the minimal idempotent group for each kernel, REEF can efficiently restore preempted tasks by re-executing the appropriate kernel group. When restoring a preempted task, the preemption module first reads the global execution index to determine which kernel was interrupted. If the interrupted kernel is idempotent, REEF simply re-executes that kernel and continues with subsequent kernels. For non-idempotent kernels, REEF looks up its pre-computed minimal idempotent group and re-executes all kernels in that group, starting from the first kernel in the group.

#### 4.5 Reset-Based Preemption on Open-Source GPUs

Our queue abstractions were originally inspired by the design of ROCm [3], an open-source GPU runtime for AMD GPUs. ROCm has exactly the data structure of host queue and DQ as described in Section 4.1. Therefore, REEF implements the preemption module on AMD GPUs by directly modifying its implementation of host queue and DQ in ROCm, and adding three new APIs to HIP [86].

**Queue eviction.** REEF adds two new APIs to the HIP for managing host queues and DQs: (1) `hip_reset_hq`, which resets a host queue and moves the untransmitted kernels to the GC thread; (2) `hip_set_stream_cap`, which limits the capacity of the DQ used by a GPU stream.

**Kernel killing.** REEF retrofits the GPU reset function of Linux AMD GPU driver [84], which instructs the GPU command processor to kill all running kernels on the CUs. We expose this function to the preemption module via `ioctl` system call, and add a new API `hip_reset_kern` to the HIP to interrupt and kill all running kernels. Our experiments show that the kernel killing function can terminate all running kernels in less than  $3\mu\text{s}$  on our testbed.

#### 4.6 Reset-Based Preemption on Closed-Source GPUs

The closed-source GPU runtime, such as CUDA [71] of NVIDIA GPUs, poses two main challenges for implementing reset-based preemption. First, the CUDA runtime does not expose the queues to the user. Even worse, prior research also discovered that CUDA kernels are launched to the DQs, without buffering in a host queue [12]. As a result, our techniques for manipulating the queues cannot be directly applied to NVIDIA GPUs outside of the CUDA runtime. Second, the CUDA runtime also does not provide an API to reset the CUs, making the kernel killing mechanism also difficult to implement. We have implemented REEF for NVIDIA GPUs, which tackles these challenges with the following techniques.

**Queue eviction.** To address the first challenge, REEFnv first wraps each CUDA stream, into a *virtual* host queue (vHQs), which intercepts and buffers all launched kernels. Similar to the (physical) HQ, each vHQ also has a background thread to transmit buffered kernels asynchronously to the CUDA runtime. After that, REEFnv treats the whole GPU runtime as several DQs (one for each GPU stream), such that REEFnv can easily reset vHQs to evict buffered kernels, instead of resetting HQs

directly (● in Figure 7). REEFnv still follows the lazy eviction to reset DQs. To simulate DQ capacity restriction, REEFnv limits the number of outstanding kernels in the CUDA runtime; the background thread of vHQ transmits a fixed number of kernels to the CUDA runtime in a closed loop.

**Special GPU commands handling.** In addition to GPU kernels, some inference engines will also launch other GPU commands when submitting inference tasks, such as memcpy, memset, and event recording. When submitting these commands, REEFnv will block them from being submitted to the DQ until all commands in the DQ have been executed. This ensures that these special commands are executed only once.

**Kernel killing.** To address the second challenge, REEFnv leverages the GPU trap mechanism which is originally used for debugging. The interface for triggering GPU traps is exposed by the recently open-source NVIDIA GPU driver [69] via the `ioctl` system call and is compatible with the closed-source driver. The trap mechanism of the GPU is similar to that of ARM CPUs. When the GPU trap is triggered, all running GPU threads are interrupted and redirected to a trap handler, which is a piece of code in the GPU instruction memory. The handler pushes several general-purpose registers to the stack in GPU local memory and starts to check trap reason and handle the exception. REEFnv utilizes the CUDA export table API to get the address of the trap handler and adds several instructions to its binary. The modified trap handler additionally checks the preemption flag to identify whether the trap is intentionally triggered by the preemption module of REEFnv. If the preemption flag is true, the trap handler will then kill the running kernel by executing `EXIT` instruction on all threads. In this case, all running threads are terminated, without saving either its registers or shared memory. If the preemption flag is false, the trap handler will find nothing to be handled and simply return back to the normal execution flow. Note that inactive thread blocks of the running kernel are then evicted by the preemptor in Figure 8 since they have not been executed yet. Our experiments show that the kernel killing latency is about 15  $\mu$ s on our testbed (GV100). We have confirmed that the approach maintains compatibility across multiple CUDA versions (10.4 through 12.1), GPU driver versions (530.x.x to 570.x.x), and GPU architectures (from Volta to Ampere).

**Binary instrumentation.** Many popular DNN frameworks (e.g., PyTorch [77]) use GPU kernels from closed-source libraries (e.g., cuDNN [18]), or even the entire inference engine is closed-source (e.g., TensorRT [68]), making the source code transformation for preemption flag checking and execution index updating infeasible. To enable REEFnv on these closed-source inference engines, REEFnv further implements dynamic binary instrumentation to inject the device codes (preemptor in Figure 8) into the binary code of the kernels. The preemptor code is compiled offline, and is dynamically injected into the binary code of the kernels at runtime. The injection is implemented using NVBit [99], a dynamic binary instrumentation tool for NVIDIA GPUs. However, the naive use of NVBit introduces a significant overhead to the kernel execution because NVBit also injects instructions to save and restore the register states before and after the preemptor code. These register saving and restoring instructions are unnecessary for our purpose, since the preemptor code is injected at the beginning of the kernel. To mitigate such overhead, REEFnv removes the register saving and restoring instructions injected by NVBit, and only keeps the preemptor code. Our experiments show that the integration of NVBit makes reset-based preemption work with closed-source inference engines (i.e., TensorRT [68]).

## 5 Dynamic Kernel Padding

To achieve high throughput, REEF allows both real-time and best-effort tasks to be concurrently executed on the GPU. 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.

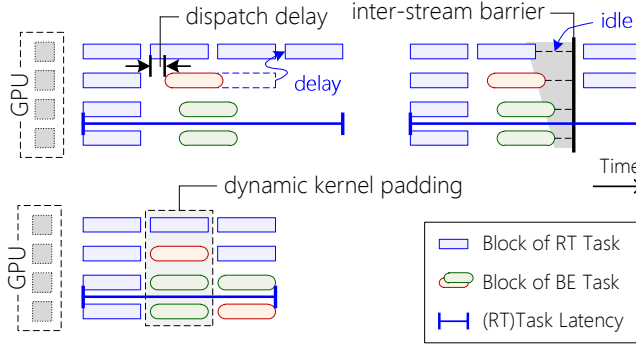


Fig. 11. An example of executing multiple kernels in parallel using different approaches.

First, using different GPU streams to launch real-time and best-effort tasks cannot avoid interfering with each other. As shown in Figure 11, the dispatch delay between GPU streams (20–40  $\mu$ s) 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 [100] 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.

**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 Figure 11. 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.

Figure 12 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 (lines 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 (lines 1–3) and then assigns the leftover CUs to the best-effort kernels (lines 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 Section 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

---

```

# 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)
    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. 12. Pseudocode for dynamic kernel padding in REEF.

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 pre-defined 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 [58].

*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 Figure 12). 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) in modern GPU programming frameworks. 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 [60]. 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 pre-allocate as many registers as possible (i.e.,

over-allocation), which may decrease the CU occupancy<sup>3</sup>, 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 (Section 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 by introducing a set of *proxy kernels*. Proxy kernels share the same source code as the dkp kernel in Figure 12, 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 block execution time of best-effort kernels must be shorter than the kernel execution time of the real-time kernel<sup>4</sup>, since the execution time of the dkp kernel is determined by the slowest block.*

<sup>3</sup>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. In general, fewer resources usage leads to higher CU usage, and higher CU occupancy can lead to better performance.

<sup>4</sup>The block execution time means the execution time of a single block of the kernel, while the kernel execution time means the execution time of all blocks.

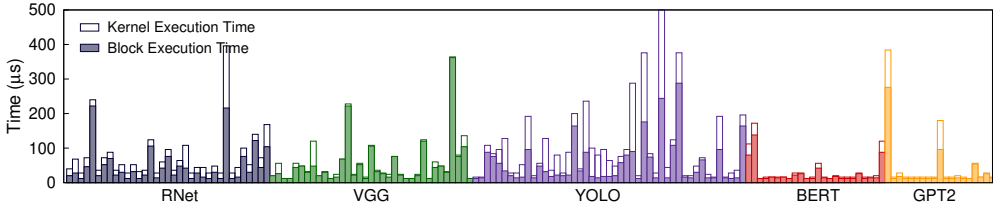


Fig. 13. Measured execution time of kernels in DNN models. The solid bars represent the execution time of a single block, while the hollow bars represent the execution time of all blocks of the kernel. For simplicity, we omit the results of the repeatedly executed kernels as they have the same execution time as the first execution. For model details, refer to Table 1.

Based on the observation of latency predictability for GPU kernels in DNN models (see Section 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 statically obtained.

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 microseconds, but also effective, limiting the latency overhead of real-time kernels to less than 3% on average, see Section 6.5 for details. However, the policy is also conservative, so the constraint may limit the 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 BERT in Figure 13), the throughput improvement of dynamic kernel padding may be trivial, even if the real-time tasks only use a few CUs.

### 5.3 Multi-Version Kernel Selection

An ideal best-effort kernel should have a short block execution time and a high CU occupancy, increasing the probability of being padded to the real-time kernel. However, current DNN compilers primarily emphasize generating kernels with the shortest kernel execution time [17, 48, 62, 92, 120, 123]. This optimization goal, while effective for standalone kernel execution, may not be optimal for best-effort tasks in the context of dynamic kernel padding. The reason is that the kernel implementation with the shortest kernel execution time often trades off other important characteristics that are crucial for best-effort tasks in dynamic kernel padding. First, an efficient kernel implementation with a short kernel execution time usually demands more resources, such as registers and shared memory, resulting in a lower CU occupancy. Second, an efficient kernel implementation also tends to involve more computations within a block to better utilize the on-chip shared memory, thereby leading to a longer block execution time. Consequently, the default implementation of best-effort kernels generated by the DNN compiler may miss the opportunity to be padded to the real-time kernel.

To demonstrate this, we generated 1,000 kernel implementations for a convolution operator from VGG model and a matrix multiplication operator from BERT using TVM [98], and measured their execution time and CU occupancy on both AMD MI50 and NVIDIA GV100 GPUs. As shown in Figure 14, on AMD MI50, the kernel for the convolution operator with the shortest kernel execution time (54.2  $\mu$ s) has a block execution time of 30.8  $\mu$ s, which is not the shortest block execution time (18.0  $\mu$ s). Moreover, Figure 14 reveals that kernels with shorter execution times typically exhibit lower CU occupancy. These trends are consistent across both NVIDIA GV100 GPU and the matrix multiplication operator. This observation highlights that the default kernel implementation, which

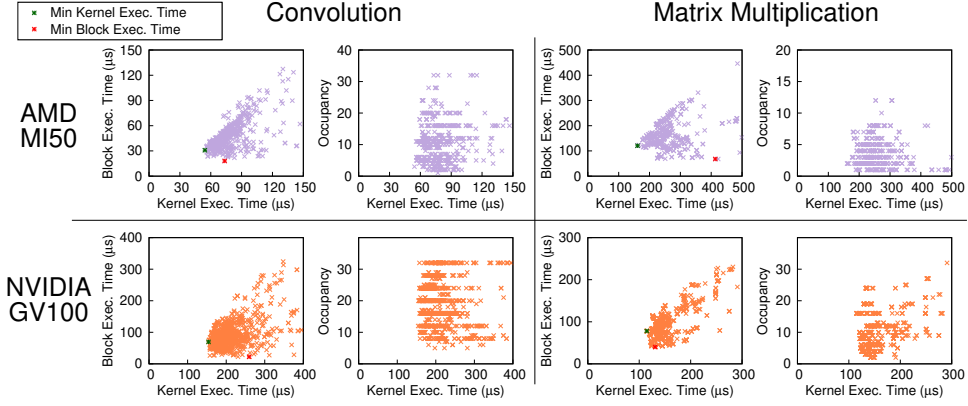


Fig. 14. Block execution time and occupancy of each generated GPU kernel for a convolution operator from VGG and a matrix multiplication operator from BERT.

prioritizes the shortest execution time for real-time tasks, may not be optimal for best-effort tasks in the context of dynamic kernel padding.

Consequently, to address this issue, we further propose *multi-version kernel selection*. This approach enables a DNN operator to incorporate multiple kernel implementations with varying resource consumption and performance. By doing so, the scheduler gains more flexibility in choosing an appropriate kernel implementation for the best-effort task that is subsequently padded with the real-time kernel.

**Multi-version kernel compilation.** To enable multi-version kernel selection, we retrofit the *tuning* functionality of modern DNN compilers [17, 48, 62, 120]. This allows us to generate multiple kernel implementations for each DNN operator, achieving a trade-off between resource usage and performance. The DNN compiler initially defines a template equipped with a set of tunable parameters for each DNN operator. Subsequently, it generates multiple kernel implementations by tuning these parameters. These parameters directly control how the operator is computed (for example, how many loops each block executes, how many data are cached in shared memory, whether loops are unrolled, etc.), thereby affecting the block execution time and resource usage. By default, the DNN compiler tunes the parameters and selects the kernel implementation with the shortest kernel execution time. However, in our approach, we preserve all kernel implementations for each DNN operator. We subsequently utilize these implementations as candidate kernels for the best-effort tasks in dynamic kernel padding.

However, sampling all the kernel implementations for each DNN operator may introduce a large number of candidate kernels, which may significantly increase the kernel selection time. For example, a convolution operator from VGG model can have over 1,000 candidate kernels. The kernel selection time increases linearly with the number of candidate kernels. When the number of candidate kernels is 100, the kernel selection time reaches approximately 10  $\mu$ s, which is even longer than the execution time of some kernels, and may not be able to be hidden by the execution time.

**Offline kernel pruning.** To address this issue, we propose *offline kernel pruning*, which prunes the candidate kernels to reduce the kernel selection time. The key idea is to eliminate a candidate kernel if there is another candidate kernel that is superior to it in all aspects. Specifically, we establish a *kernel dominance relation* to facilitate this pruning process as follows. A kernel  $k_1$  is said to dominate another kernel  $k_2$  if  $k_1$  displays (1) a shorter kernel execution time, (2) a shorter block

execution time, (3) a lower register usage, (4) a lower shared memory usage, (5) a smaller block size, and (6) a smaller block number compared to  $k_2$ . Consequently, we discard a candidate kernel if there exists another candidate kernel that dominates it. Through the utilization of offline kernel pruning, the number of candidate kernels for each DNN operator remarkably decreases from over 1,000 to roughly 10. This significant reduction effectively diminishes the kernel selection time.

The overhead of multi-version kernel compilation and pruning is minimal. First, the compilation overhead is negligible since sampling kernel candidates is already part of the model compiler's workflow—we simply preserve these kernels instead of discarding them after selection. Second, kernel pruning is a lightweight process that only involves comparing kernel characteristics (e.g., execution time, resource usage, etc.) and can be completed within 1 second for each model.

**Multi-version kernel selection.** The selection policy with multi-version kernels is similar to the original policy in Section 5.2, with the exception that more than one candidate kernel are available for each best-effort task. Specifically, every time the scheduler launches a real-time kernel, it iterates through all the candidate kernels for the best-effort tasks, and looks for the kernels that meet the two rules in Section 5.2. When multiple kernels satisfy the two rules, the scheduler selects the one with the shortest kernel execution time to maximize the throughput of the best-effort tasks.

#### 5.4 Dynamic Kernel Padding on Open-Source GPUs

We discuss the implementation details of dynamic kernel padding on open-source AMD GPUs.

**Dynamic register allocation.** Dynamic register allocation for AMD GPUs is implemented by modifying the assembly file of proxy kernels. AMD GPU device code assembly files contain attributes specifying the number of registers (SGPR and VGPR) used for each kernel. We directly modify these attributes before the assembly code is compiled into binary to generate proxy kernels with different register counts.

**Handling private memory.** Private memory is generally used for stack operations. The size of private memory should be statically set for proxy kernels. However, we observe that kernels using private memory take approximately 10  $\mu$ s longer for dispatching, likely due to stack allocation and initialization. Setting non-zero private memory size for all proxy kernels would affect the performance of candidate kernels that do not use private memory. Fortunately, most kernels do not use stack because the large amount of available registers is sufficient for intermediate data. Therefore, we implement a simple but effective approach. Every proxy kernel has two versions: one that includes private memory and another that does not. When a candidate kernel requires private memory, it is launched using the version with private memory. In contrast, if a candidate kernel does not require private memory, it is launched through the version without private memory. This approach ensures the correctness of candidate kernels that use private memory, while avoiding unnecessary performance overhead for those that do not.

#### 5.5 Dynamic Kernel Padding on Closed-Source GPUs

The dynamic kernel padding is also applicable to closed-source NVIDIA GPUs. We discuss the implementation details here.

**Dynamic register allocation.** For NVIDIA GPUs, the register allocation is also static, which means the register count of a kernel cannot be modified at runtime. To address this issue, we propose a similar approach to REEF by introducing a set of *proxy kernels* with different register counts, allowing the scheduler to dynamically pick the proper proxy kernel. However, the main technical challenge is that the register count of a kernel cannot be explicitly set in the source code, which makes it difficult to generate proxy kernels. To mitigate this issue, we first compile a set of proxy kernels into ELF files (i.e., binary files), and then directly manipulate the ELF file of the

Table 2. DISB Workload Description

DISB	A	B	C	D	E
Num. of RT clients	1/VGG	1/VGG	1/VGG	5/ALL	5/ALL
Avg. throughput per RT client	50% [U]	100%	50% [U]	10% [U]	10% [P]
Num. of BE clients	1/RNET	1/RNET	5/ALL	5/ALL	5/ALL

“#/model” denotes the number of clients and their DNN models. “[U/P]” denotes the arrival distribution (i.e., Uniform or Poisson).

kernel binary to modify the register count of each proxy kernel, which is a feasible approach to generate proxy kernels for NVIDIA GPUs.

**Handling constant memory.** NVIDIA GPUs offer a distinct memory segment known as constant memory, characterized by its read-only nature and caching capabilities. The NVIDIA GPU compiler, namely `nvcc`, has a unique feature of utilizing constant memory to hold literal constants, which is not found in AMD GPUs. Kernels access this memory through relative addressing, involving both the relative bank number and the offset within the bank. Notably, the constant memory address space may be shared between the real-time kernel and the best-effort kernel when they are executed via dynamic kernel padding, potentially leading to contention issues. Fortunately, only a few kernels (2 kernels in Table 1) are observed to use constant memory, so we disable dynamic kernel padding for the kernels that use constant memory, which is a simple yet effective approach to address the contention issue.

## 6 Evaluation

### 6.1 Implementation

We have implemented REEF on both AMD and NVIDIA GPUs with about 8.7k lines of C++ code. For AMD GPUs, REEF extends AMD ROCm [3] to support the reset-based preemption technique. REEF uses TVM [17] to generate DNN inference kernels, and extends it to support the dynamic kernel padding mechanisms. To demonstrate the generality and the extensibility of REEF, we also adapted REEF to support serving inference requests using TensorRT [68] and PyTorch [77].

### 6.2 Experimental Setup

**Testbed.** The experiments were mainly conducted on a GPU server that consists of one Intel Core i7-10700 CPU (total 8 cores), 32 GB of DRAM, and one AMD Radeon Instinct MI50 GPU (60 CUs and 16 GB of memory). The software environment of the server was configured with ROCm 4.3.0 [3], Apache TVM [98] 0.14.0, and Ubuntu 20.04. The hardware platform resembles the computational resources of autonomous vehicles [4, 97]. We further evaluate REEFnv on a NVIDIA GV100 GPU (80 SMs and 32 GB of memory) to demonstrate the generality of our approach, using the same server with CUDA 11.4 [71], PyTorch 1.12.0 [77], and TensorRT 8.5.1 [68] installed.

**Workloads.** Inspired by YCSB [20, 21], 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 [8]), while the clients in DISB E send requests with a Poisson arrival distribution, which simulates event-driven real-time DNN applications (e.g., speech recognition [42, 102]). 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). The request frequency of each RT client is set based on a relative throughput

target, e.g., 50% for DISB A and C, meaning that each RT client sends requests at a frequency of 50% of the maximum throughput that can be achieved by the GPU. For DISB B, the RT client sends requests at the maximum throughput, and for DISB D and E, each RT client sends requests at 10% of the maximum throughput.

Five representative DNN models are deployed, including ResNet-152 [40] (RNET), VGG-19 [94] (VGG), Yolov3 [82] (YOLO), Bert-Base [28] (BERT), and GPT-2 [80] (GPT2). For BERT and GPT2, we use a 128-token sequence length input, and for GPT2, the model only executes one forward pass (i.e., prefill) for each request. Each DISB 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 DNN models in their clients separately, which simulates multiple DNN applications in a single scenario (e.g., autonomous vehicles [8, 54]).

Furthermore, we use a real-world trace from an open autonomous driving platform (i.e., Apollo [8]) 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 [56, 88], 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 [35]. 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 Triton [22]. 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.<sup>5</sup>

**Metrics.** We use two key metrics to evaluate the performance of different approaches: (1) normalized end-to-end latency for real-time requests and (2) normalized throughput for all clients. For real-time requests, we normalize each request's end-to-end latency by its model's standalone execution time. The standalone execution time is pre-measured offline by taking the average execution time over 1000 runs. For throughput, we normalize each client's throughput to the maximum throughput achievable by its model when running alone on the GPU. These normalized metrics allow us to fairly compare performance across different DNN models and scheduling approaches.

### 6.3 Overall Performance

We first compare the normalized end-to-end latency of real-time tasks and the normalized overall throughput of all clients using DISB workloads and a real-world trace. The experiments are conducted on both AMD MI50 and NVIDIA GV100 GPUs, and the results are shown in Figures 15 and 16.

**Single-RT Single-BE Client (DISB A and B).** For workloads with a single real-time and best-effort client, SEQ significantly increases real-time task latency by 2.87× and 3.66× on AMD MI50 GPU,

<sup>5</sup>In this case, additional GPUs are dedicated to best-effort tasks, which also results in extra cost and energy consumption, as well as low GPU utilization.

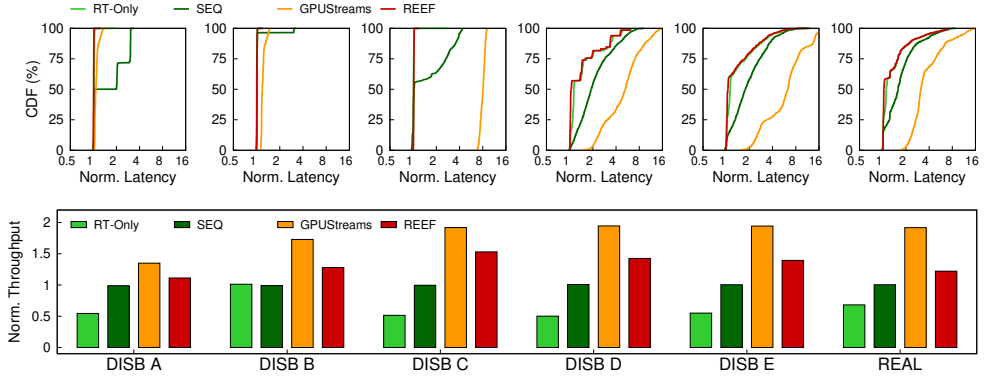


Fig. 15. Comparison of (a) normalized end-to-end latency for RT tasks, and (b) normalized overall throughput of all tasks using various scheduling approaches. **Testbed:** One AMD MI50 GPU.

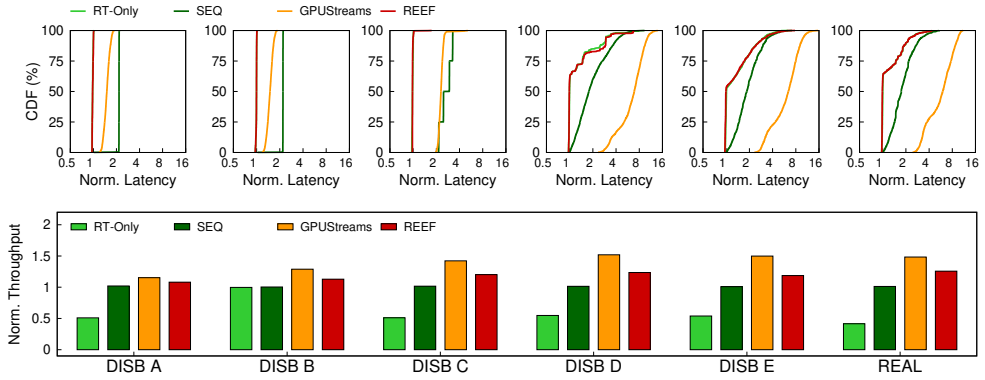


Fig. 16. Comparison of (a) normalized end-to-end latency for RT tasks, and (b) normalized overall throughput of all tasks using various scheduling approaches. **Testbed:** One NVIDIA GV100 GPU.

and  $2.16\times$  and  $2.19\times$  on NVIDIA GV100 GPU. This substantial latency increase stems from SEQ's inability to preempt running best-effort tasks, forcing real-time tasks to wait for their completion. The waiting time is entirely dependent on the remaining execution time of the best-effort task. In contrast, GPUStreams shows more moderate latency increases of  $1.1\times$  and  $1.21\times$  on AMD MI50, and  $1.46\times$  and  $1.51\times$  on NVIDIA GV100. While GPUStreams can execute real-time tasks without waiting for best-effort tasks to complete, the concurrent execution through multiple streams introduces GPU resource contention, leading to interference with real-time task performance. The degree of performance degradation varies based on the level of resource competition from best-effort tasks — less severe on AMD GPUs due to lower CU utilization by best-effort tasks, and more pronounced on NVIDIA GPUs. Notably, REEF maintains near-optimal real-time task performance with only a 0.5% latency increase.

From a throughput perspective, SEQ's single-stream execution limits its throughput to  $1.0\times$  regardless of workload. GPUStreams achieves higher throughput of  $1.34\times$  and  $1.72\times$  on AMD MI50, and  $1.15\times$  and  $1.28\times$  on NVIDIA GV100. REEF leverages dynamic kernel padding to execute best-effort tasks alongside real-time tasks, achieving throughput improvements of  $1.12\times$  and  $1.28\times$  on AMD MI50, and  $1.08\times$  and  $1.12\times$  on NVIDIA GV100. Notably, even in DISB B where real-time tasks saturate 100% of GPU resources, REEF successfully prevents best-effort task starvation.

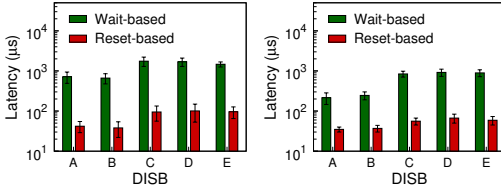


Fig. 17. Comparison of preemption latency between reset-based and wait-based approaches using DISB workloads on (a) AMD MI50 and (b) NVIDIA GV100 GPUs.

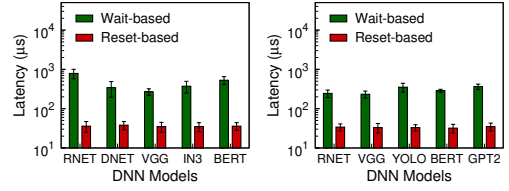


Fig. 18. Comparison of preemption latency when preempting one DNN inference task of different DNN models on (a) AMD MI50 and (b) NVIDIA GV100 GPUs.

**Single-RT Multi-BE Clients (DISB C).** With the number of best-effort clients increased to five, GPUStreams shows significantly higher real-time task latency increases of  $8.1\times$  and  $4.2\times$  on AMD and NVIDIA GPUs respectively. This substantial degradation occurs because more concurrent best-effort tasks interfere with real-time task execution. SEQ's performance degradation is less severe than that of GPUStreams since it is only affected by one running best-effort task at a time, independent of the total number of clients. In contrast, REEF maintains stable performance with only a 2% latency increase. Throughput improvements are more pronounced with increased concurrency: GPUStreams achieves  $1.91\times$  and  $1.42\times$  on AMD and NVIDIA GPUs, while REEF reaches  $1.53\times$  and  $1.20\times$  respectively.

**Multi-RT Multi-BE Clients (DISB D, E and real-world trace).** When multiple real-time tasks are present, REEF faces a challenge due to different task periods and a single real-time queue, potentially causing some real-time tasks to wait for preceding tasks to complete. Despite this, REEF maintains performance close to RT-Only, with the highest average latency increase of only 4.2% on AMD GPU (DISB E). Throughput improvements remain significant, ranging from  $1.22\times$  to  $1.42\times$  on AMD GPU and  $1.18\times$  to  $1.25\times$  on NVIDIA GPU, demonstrating REEF's effectiveness in handling complex multi-client scenarios while maintaining real-time task performance.

#### 6.4 DNN Inference Preemption

The vanilla wait-based preemption approach proposed in prior work [14] 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 number 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.** Figure 17 compares the preemption latency of two approaches.

On AMD MI50 GPU, the reset-based preemption outperforms the wait-based approach by more than an order of magnitude for all DISB workloads, from  $19.3\times$  (DISB B) to  $21.8\times$  (DISB C). While on NVIDIA GV100 GPU, the reset-based preemption outperforms the wait-based approach by from  $6.1\times$  (DISB A) to  $15.0\times$  (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 DQs, 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 single BE client (DISB A and B).

Furthermore, we evaluate the preemption latency for diverse 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 Figure 18, the wait-based preemption latency highly depends on the type of models, from  $310\mu s$  (VGG) to  $882\mu s$  (RNET) on AMD MI50 GPU and from

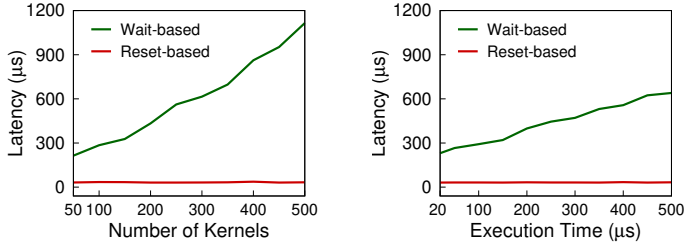


Fig. 19. Comparison of preemption latency with the increase of (a) launched kernels and (b) kernel execution time. **Testbed:** One AMD MI50 GPU.

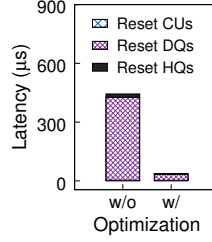
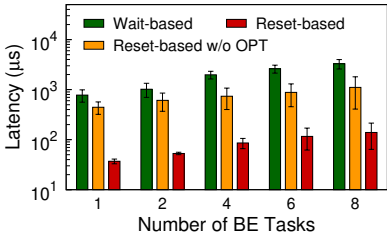


Fig. 20. (a) Comparison of preemption latency with the increase of BE clients, and (b) latency breakdown of the reset-based approach without (w/o) and with (w/) optimizations. **Testbed:** One AMD MI50 GPU.

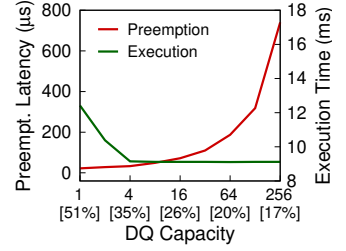


Fig. 21. Preemption latency and execution time with the increase of DQ capacity. [%] denotes CPU utilization during normal execution.

233  $\mu$ s (RNET) to 365  $\mu$ s (GPT2) on NVIDIA GV100 GPU, 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 30  $\mu$ s to 38  $\mu$ s on both GPUs for all five models.

To further investigate the impact of different model properties on the preemption latency on AMD MI50 GPU, we simulate DNN models with different numbers 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  $\mu$ s, respectively. As shown in Figure 19, the preemption latency of the wait-based approach raises linearly, while our reset-based preemption approach remains stable at very low latency (less than 40  $\mu$ s). For the 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 DQs 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, Figure 20 shows the preemption latency with the increase of BE clients (RNET), and the latency breakdown for a single BE client, evaluated on AMD MI50 GPU. By enabling two optimizations, the preemption latency significantly drops by up to 92% (from 87%), as shown in Figure 20(a). As a reference, even without optimization, the reset-based approach still outperforms wait-based approach by up to 3.0 $\times$  (from 1.7 $\times$ ).

Since the two optimizations are used when resetting host and DQs, respectively, Figure 20(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  $\mu$ s to 3  $\mu$ s. Meanwhile, using queue capacity restriction further reduces the latency

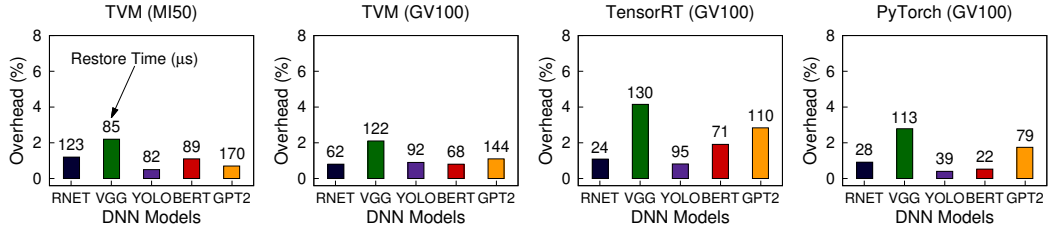


Fig. 22. Restore overhead for different DNN models where labels show restore time (in  $\mu\text{s}$ ). Experiments are conducted on an AMD MI50 GPU using TVM as the inference engine, and an NVIDIA GV100 GPU using TVM, TensorRT, and PyTorch as the inference engines.

of resetting DQ from  $424\mu\text{s}$  to  $31\mu\text{s}$ . Note that using command processor to reset CUs is extremely fast (less than  $3\mu\text{s}$ ).

**Queue capacity.** We restrict the DQ capacity to mitigate the overhead incurred by lazily evicting the remaining kernels in the queue (see Section 4.2 for details). However, reducing the queue capacity also increases normal execution time and CPU utilization. Figure 21(a) shows the preemption latency and normal execution time when serving RNET inferences as the queue capacity increases, evaluated on AMD MI50 GPU. When the DQ capacity increases from 1 to 4, the execution time reduces from  $14.3\text{ ms}$  to  $12.3\text{ ms}$ . 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 DQ on our testbed, which has almost zero overhead for normal execution and provides acceptable preemption performance (about  $30\mu\text{s}$ ). Finally, using a smaller DQ 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 measure this overhead by having a single best-effort client send inference requests, where each task is randomly preempted and then restored. As shown in Figure 22, the restore time remains low across all DNN models, ranging from  $22\mu\text{s}$  to  $170\mu\text{s}$ . This variation primarily correlates with the kernel execution time of each DNN model (see Figure 13). For DNN models compiled and executed using TVM, REEF achieves efficient restoration by executing at most one redundant kernel, since all TVM kernels are idempotent. However, for models executed using TensorRT and PyTorch, the presence of non-idempotent kernels requires REEF to re-execute an idempotent kernel group (see Section 4.4) when restoring a preempted non-idempotent kernel. Despite this additional complexity, the restoration overhead remains minimal—the number of re-executed kernels is capped at nine, with most groups containing only one to three kernels. As a result, the total overhead stays below 5% across all cases.

## 6.5 Dynamic Kernel Padding

To study the efficacy of dynamic kernel padding, we evaluate its performance under high contention scenarios. Our evaluation setup consists of one real-time client and one best-effort client that simultaneously send requests at a high-enough frequency to keep the GPU busy. We compare REEF with GPUStreams, which serves both types of requests concurrently using different GPU streams to achieve the highest overall throughput. In contrast, REEF prioritizes real-time tasks while opportunistically padding best-effort tasks to avoid starvation and improve overall throughput. We evaluate two variants of dynamic kernel padding: the baseline version without multi-version

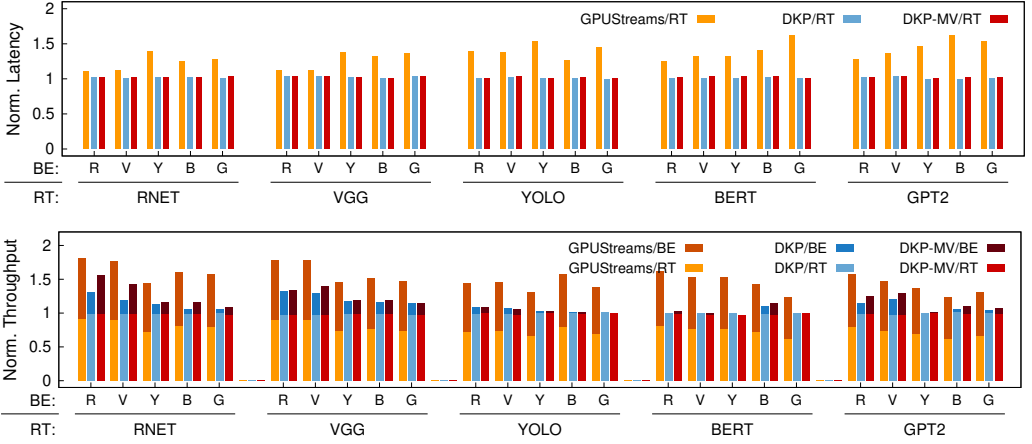


Fig. 23. Comparison of (a) normalized end-to-end latency of RT tasks and (b) normalized overall throughput of both tasks using different concurrent execution schemes. **Testbed:** One AMD MI50 GPU.

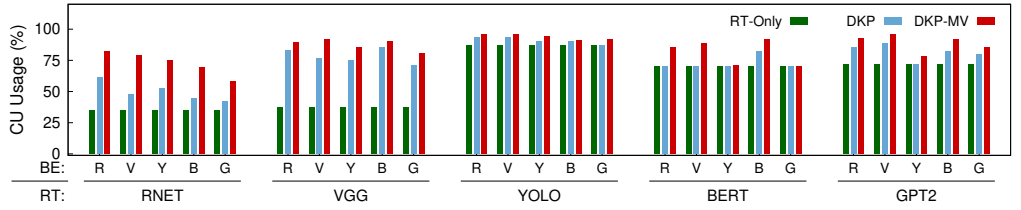


Fig. 24. Average compute unit (CU) usage for running RT and BE kernels with various DNN model combinations using dynamic kernel padding.

kernel selection (DKP), where the real-time and best-effort kernels are the same for the same DNN model, and an enhanced version with multi-version kernel selection (DKP-MV).

**Performance.** Figure 23 reports the experimental results for one-to-one combinations among five DNN models using above workload on AMD MI50 GPU. We use the same metrics as mentioned in Section 6.2, i.e., normalized latency for real-time tasks and normalized throughput for all tasks. As expected, GPUStreams significantly amplifies real-time task latency by an average of 1.34 $\times$ , ranging from 1.10 $\times$  to 1.62 $\times$ , 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 1.0% (up to 2.9%) for DKP (without multi-version kernel selection).

For overall throughput, we separately report the throughput of real-time tasks and the normalized throughput of best-effort tasks. Although GPUStreams increases overall throughput by an average of 1.50 $\times$ , the throughput of real-time tasks drops by 24.7% 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 by an average of 1.12 $\times$  (up to 1.34 $\times$ ). The performance improvement of using dynamic kernel padding mainly depends on two conditions. First, the CU usage of both real-time and best-effort tasks significantly impacts the throughput improvement. When real-time tasks use fewer CUs, more resources are available for best-effort tasks. For instance, when using VGG as the best-effort task, DKP achieves a 19.8% throughput improvement with RNET (34% CU usage, see Figure 24) as the real-time task, but only 8.1% with YOLO (87% CU usage). Similarly, best-effort tasks with lower CU usage can achieve

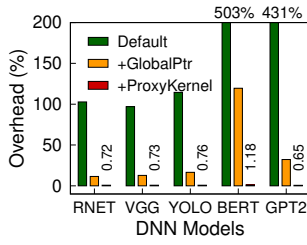


Fig. 25. Comparison of execution time overhead for padded kernels using various optimizations.

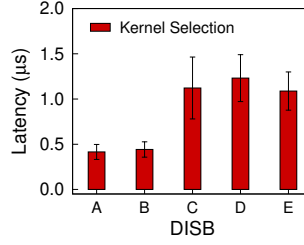


Fig. 26. Execution time of kernel selection for DISB A–E.

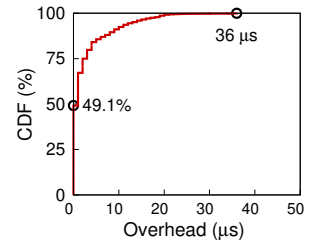


Fig. 27. CDF of execution time overhead for RT kernels using dynamic kernel padding.

higher throughput improvements when executing alongside the same real-time task. This factor also affects GPUstreams' performance in a similar way. Second, the relative relationship between kernel execution times of real-time and best-effort tasks determines the opportunity for kernel padding. When real-time tasks have relatively longer kernel execution times compared to best-effort tasks, there are more opportunities to pad them together. For example, VGG's kernels have relatively long execution times (see Figure 13), so when used as a real-time task, it allows for throughput improvements across all best-effort tasks. In contrast, BERT's kernels have shorter execution times, and throughput improvements are only possible when BERT itself is used as a best-effort task.

**Multi-version kernel selection.** By leveraging multi-version kernel selection, REEF can achieve an average throughput of  $1.18\times$  (up to  $1.57\times$ ). The performance improvement of multi-version kernel selection is more significant when both real-time and best-effort models use fewer CUs. For instance, when using RNET as the real-time task, the average overall throughput increases from  $1.16\times$  with DKP to  $1.29\times$  with DKP-MV. This improvement is also reflected in the CU usage (shown in Figure 24), which increases from an average of 49% with DKP to 72% with DKP-MV. However, multi-version kernel selection shows limited benefits for models with inherently high CU usage, as their performance is constrained by limited resources rather than kernel execution time. Notably, while DKP results in zero throughput for best-effort tasks in four cases (YOLO-GPT2, BERT-RNET, BERT-VGG, and GPT2-YOLO), DKP-MV successfully prevents best-effort task starvation in these scenarios.

**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 Figure 25, the default function pointer implementation (Default) incurs execution time overhead from 97% up to 503% for real-time tasks with different DNN models. By using the global function pointer (GlobalPtr), the overhead is significantly reduced to an average of 38.5% (ranging from 11.5% to 119%), 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 (1.18% 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.

**Kernel selection.** Figure 26 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  $0.4\mu s$  to select best-effort kernels for the given real-time kernel. The selection time increases to  $1.2\mu s$  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.

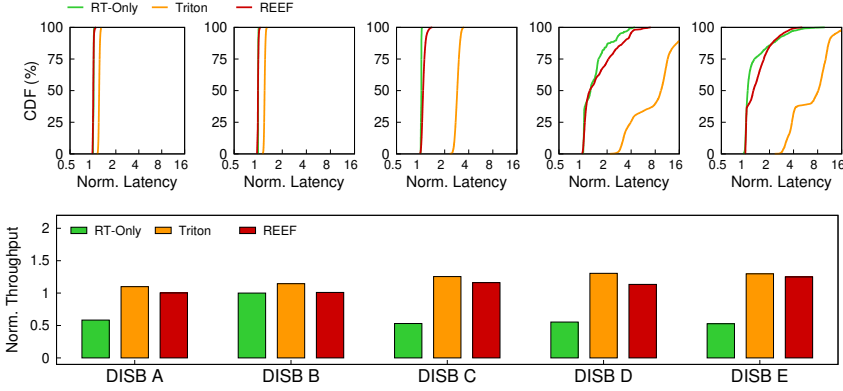


Fig. 28. Comparison of (a) normalized end-to-end latency for RT tasks, and (b) normalized overall throughput of all tasks using Triton [22] and REEF. Both Triton and REEF use the TensorRT [68] to execute DNN models. **Testbed:** One NVIDIA GV100 GPU.

**Overhead.** 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 Figure 27, over 49.1% 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 9  $\mu$ s. The increase in execution time is mainly due to the contention on GPU memory and shared L2 cache. Notably, while such overhead on a real-time kernel may delay the start of the next one, it does not affect the execution time of subsequent kernels. So if a best-effort kernel runs longer than expected, it will only affect the real-time kernel it is padded with.

## 6.6 Comparison with State-of-the-Art Systems

We further compare the performance of REEF with two state-of-the-art systems: Triton [22], a production-level inference serving system, and Orion [95], a fine-grained GPU sharing system. Both these systems use inference engines with closed-source GPU kernels, i.e., TensorRT [68] for Triton and PyTorch [77] for Orion. REEF does not support dynamic kernel padding in this comparison, as it requires modifying the source code of GPU kernels, which is not allowed for closed-source inference engines. Fortunately, our reset-based preemption implementation on NVIDIA GPU leverages dynamic binary instrumentation to transform the binary code of the GPU kernel at runtime, thereby successfully working with the closed-source inference engines. We still compare the performance of REEF with Triton and Orion in terms of normalized end-to-end latency for real-time tasks and normalized overall throughput of all tasks using the DISB workloads.

**Comparison with Triton.** Triton is a production-level inference serving system, and has been widely used in the industry. Triton uses multiple GPU streams to serve multiple clients concurrently. To make a fair comparison, we enable the priority feature of Triton’s TensorRT backend, and set the models used by real-time clients to MAX priority, while the models used by best-effort clients were set to MIN priority. The priority feature of TensorRT backend is implemented by setting the CUDA stream priority. The experimental results are shown in Figure 28.

Across all workloads, REEF consistently achieves lower real-time task latency compared to Triton, ranging from 1.12 $\times$  lower in DISB-A to 5.20 $\times$  lower in DISB-E. Specifically, in DISB-A and B, Triton’s real-time tasks experience relatively moderate interference, with an average latency increase of only 1.21 $\times$ . However, in DISB-C, the increased number of best-effort clients leads to a more significant latency increase of 2.8 $\times$  for real-time tasks. In contrast, REEF’s preemptive scheduling approach

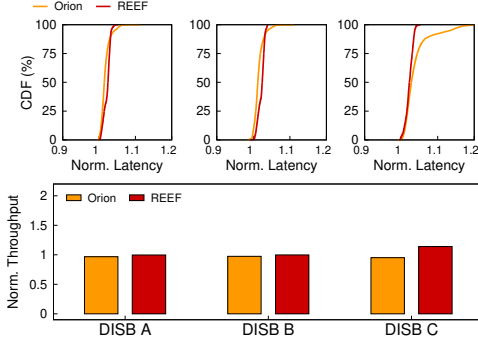


Fig. 29. Comparison of (a) normalized end-to-end latency for RT tasks, and (b) normalized overall throughput of all tasks using Orion [95] and REEF. Both Orion and REEF use the PyTorch [77] to execute DNN models. **Testbed:** One NVIDIA GV100 GPU.

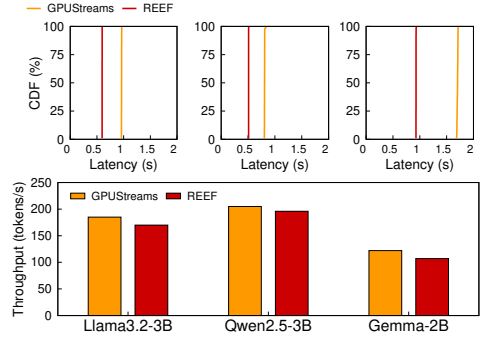


Fig. 30. Comparison of (a) end-to-end latency for RT tasks, and (b) overall throughput using GPUStreams and REEF under DISB-A workload with LLMs. The LLMs are executed using MLC-LLM [64] as the inference engine. **Testbed:** One NVIDIA GV100 GPU.

maintains stable real-time task performance, with a maximum latency increase of only 9% (average 5%) in DISB-C. In terms of throughput, Triton achieves higher average throughput of  $1.21\times$ , while REEF reaches  $1.11\times$  average throughput.

**Comparison with Orion.** Orion is a state-of-the-art fine-grained GPU sharing system designed for ML workloads, with the ability to control the interference between co-executing high-priority tasks and low-priority tasks. Because Orion only supports scheduling one high-priority task and multiple low-priority tasks, we compare the performance of REEF with Orion using only DISB A, B and C workloads by setting the real-time client to high-priority and the best-effort clients to low-priority. The experimental results are shown in Figure 29.

In DISB-A and DISB-B, Orion achieves comparable performance to REEF. This is because Orion employs a one-kernel-at-a-time scheduling approach, where it launches only one kernel at a time (prioritizing high-priority kernels), resulting in low latency for real-time tasks. However, this scheduling approach has two significant limitations. First, it cannot preempt executing best-effort kernels, leading to higher tail latency. For instance, Orion’s real-time tasks experience  $P_{99}$  latency of  $1.05\times$  in DISB-A and B, and  $1.17\times$  in DISB-C, while REEF consistently maintains  $P_{99}$  latency below  $1.05\times$ . Second, it incurs high scheduling overhead (similar to having a DQ size of 1), which reduces system throughput. Specifically, Orion achieves only  $0.97\times$  throughput in DISB-C, while REEF reaches  $1.14\times$ .

## 6.7 REEF with Large Language Models (LLMs)

REEF is not limited to the models in Table 1. We demonstrate that REEF can support more recent models, such as LLMs. To evaluate REEF’s performance with LLMs, we conduct experiments using the DISB-A workload, where one real-time client and one best-effort client submit inference requests simultaneously. The real-time client submits requests at 50% of the maximum throughput. We test three popular LLM models: Llama3.2-3B [63], Qwen2.5-3B-Instruct [79], and Gemma-2B-IT [33], where both RT and BE clients use the same model in each experiment. We set the batch size to 1 and fix the decode length to 100 tokens per request. All LLM models are executed using MLC-LLM [64] as the inference engine.

Figure 30 shows the end-to-end latency for real-time tasks and the total throughput (tokens/s) for both clients across the three LLM models. The results demonstrate that REEF consistently maintains

low latency for real-time tasks, achieving  $1.56\times$  to  $1.81\times$  lower latency compared to GPUStreams. This significant latency reduction comes with only a minimal throughput penalty—REEF’s throughput is at most 5% lower than that of GPUStreams.

## 7 Discussion

**Limitations of reset-based preemption.** The re-execution mechanism for restoring killed kernels imposes several constraints: (1) kernels must be stateless with respect to external systems (no network access or GPU device interactions), (2) kernels must be deterministic, and (3) incompatibility with advanced GPU features like dynamic parallelism, in-kernel malloc. Since DNN kernels typically perform algebraic computations without these behaviors, REEF works well for them. As for CUDA Graph, a feature increasingly used in DNN inference, it is theoretically possible for REEF to partition a large graph into smaller sub-graphs (ensuring each idempotent kernel group is contained within a single sub-graph) and re-execute at the sub-graph granularity upon preemption. However, this partitioning would diminish the performance benefits of using CUDA Graphs. An ideal solution would be to extend the CUDA Graph API to support execution starting from an arbitrary node. For general GPU applications, these constraints may limit reset-based preemption’s applicability. A potential solution is to pre-check applications and fallback to wait-based preemption when incompatibilities are detected.

**Limitations of dynamic kernel padding.** The kernel selection policy in dynamic kernel padding currently relies on offline profiling to estimate execution times and resource utilization. This approach has two significant limitations: (1) it may not accurately predict runtime behavior under dynamic conditions, particularly with varying input sizes, and (2) it does not account for memory bandwidth contention between real-time and best-effort kernels, potentially degrading performance in memory-intensive scenarios. A more precise policy can potentially mitigate these limitations, such as profiling the memory bandwidth usage of each kernel [45, 95].

**Compatibility with newer GPUs.** REEF leverages several hardware-related features to implement and optimize two key techniques. For example, in the NVIDIA GPU implementation of reset-based preemption, we utilize trap mechanisms and EXIT instructions to implement GPU reset, while in the dynamic kernel padding implementation, we use JUMP instructions to optimize function pointers. These mechanisms are not architecture-specific, as they are common hardware features. For instance, we tested the reset-based preemption mechanism on NVIDIA RTX 3080 (Ampere architecture), and it works correctly. EXIT and JUMP instructions are also supported on recent generations of NVIDIA GPUs [72]. Therefore, REEF can theoretically be compatible with modern GPU hardware.

**Workloads beyond GPU memory capacity.** Currently, REEF assumes that the GPU memory capacity is sufficient for all models. However, as models continue to grow in size, GPU memory may not be able to accommodate all model parameters. Recent work has explored optimizing performance through faster swapping between CPU and GPU memory [7, 47, 49]. REEF can be integrated with these approaches to further optimize real-time performance. Additionally, the key insight of our reset-based preemption (i.e., re-execution instead of context saving) could potentially be leveraged to optimize memory swapping itself. For instance, instead of swapping intermediate results of best-effort tasks back to CPU memory, we could recompute these results through re-execution. We leave this as a future work.

**Future GPU APIs and runtime.** We leverage several subtle hacks on the GPU runtime to enable  $\mu$ s-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 DQ (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  $\mu$ s.

**Future low-latency GPU inference systems.** For low-latency GPU inference systems, regardless of their scheduling objectives (such as meeting SLOs like Paella [67], or ensuring real-time tasks are not affected like REEF), their core mechanism is how to control interference between concurrent requests, and how to maximize throughput under the premise of controlling interference. REEF proposes reset-based preemption and dynamic kernel padding two techniques that control interference from temporal and spatial dimensions respectively. However, there is still significant room for performance improvement. For example, for dynamic kernel padding, compared to using multiple GPU streams, there is still a performance gap in terms of throughput. Therefore, we believe that future low-latency GPU inference systems need more precise control over GPU computing resource allocation mechanisms, dynamically adjusting each task's resource usage at a finer granularity (even at the nanosecond level). This requires not only new techniques on the scheduling system side, but also new mechanisms on the GPU hardware side.

## 8 Related Work

**DNN inference serving systems.** Prior model serving systems [25, 31, 39, 57, 67, 73, 93, 106, 114] mainly focus on meeting **service-level objectives (SLO)**, typically in the tens of milliseconds [26, 41, 110], and improving overall throughput of datacenter applications. Clockwork [35] leverages the latency predictability of DNN inference to achieve low tail latency. It runs inferences sequentially on dedicated GPUs to provide predictable performance. Clipper [24] and Nexus [90] enables batching inferences on the same model to improve GPU utilization and inference throughput. Abacus [26] enables simultaneous DNN inferences by accurately predicting the latency of the overlapped operators. INFaaS [87] can automatically select the right variant with different optimizations for each inference to meet diverse SLOs. AlphaServe [57] leverages model parallelism to reduce the serving latency of bursty requests. Shepherd [114] aggregates inference request streams into moderately-sized groups to improve the predictability of the request arrival pattern. USHER [93] merges multiple models with similar weights to minimize the contention in GPU cache when concurrent inferences are running on the same GPU. Paella [67] uses a similar kernel transformation approach to REEF to bypass the built-in GPU scheduler for low-latency inference. However, the latency SLOs for datacenter applications are much more relaxed than those for real-time systems, for example  $2\times$  of their solo-run latencies [26, 93]. 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.

There are also some efforts to achieve low-latency inference under circumstances where the models are not pre-loaded in GPU memory. PipeSwitch [7] pipelines the model loading and the kernel execution, based on the observation that DNN models have a layer-by-layer structure. DeepPlan [47] further leverages the direct-host-access facility provided by commodity GPUs and the GPU-GPU communication to reduce the model loading latency. DeepUM [49] transparently

prefetches the CPU memory based on the historical access pattern of the model. INFless [109] leverages both the long-term and short-term application idle times to decide when to pre-load the models. ServerlessLLM [32] optimizes the structure of the model checkpoint to fully utilize the bandwidth of multi-tier storage. In contrast, REEF assumes that the models are pre-pinned in GPU memory, and is orthogonal to the above efforts. The two mechanisms in REEF can also be integrated into the above systems to improve the performance for the time period when the models are already loaded in GPU memory.

**GPU kernel preemption.** Prior work has proposed hardware enhancement to support preemptive GPU scheduling [59, 75, 96, 115]. An intuitive solution is to support context switch on GPUs [96]. 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. [59] proposed lightweight context switching to avoid unnecessary register saving. Tanasić et al. [96] extended the hardware to passively preempt a streaming multiprocessor (SM) of GPU by stopping issuing new thread blocks. Chimera [75] 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.

Apart from the hardware enhancement, prior work has also proposed software techniques to support preemptive GPU scheduling [14, 50–53, 89, 104, 122]. Most prior work employs wait-based preemption, albeit through various approaches. For example, GPES [122] slices a GPU kernel into multiple sub-kernels and schedule tasks at the boundary of sub-kernels. EffiSha [14] and FLEP [104] leverage kernel transformation to preempt the running kernel at the end of each block. Lee et al. [53] proposes an idempotence-based preemption mechanism that can instantly preempt the running kernel without waiting for its completion similar to our reset-based preemption mechanism. The preemption mechanism in REEF is different from these work mainly in two aspects. First, REEF not only preempts the currently running kernel but also evicts the kernels in the software queues. This capability is crucial for DNN inference tasks, which often involve numerous asynchronously launched kernels. In contrast, prior work primarily concentrates on scheduling single-kernel applications. Second, while both approaches leverage the idempotence property of GPU kernels, they differ in their handling of non-idempotent kernels. Lee et al. employ a transaction scheme that involves copying the input data of non-idempotent kernels prior to their execution. This method can result in significant performance overhead during normal execution, even when preemption is not invoked. In contrast, REEF utilizes a novel multi-kernel idempotence property to recover non-idempotent kernels, incurring no overhead during normal execution.

**GPU multitasking.** There have been many efforts to concurrently execute multiple GPU kernels for high throughput [9, 34, 58, 74, 76, 100, 101, 103]. For DNN computation, Rammer [62] takes a holistic approach to exploit both inter- and intra-kernel parallelisms at compile time, which uses static kernel fusion [100] 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 [15, 16, 117, 119]. DASE [44] models the memory contention of concurrent kernels. Themis [118] 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.

## 9 Conclusion

This article presented REEF, the first DNN inference serving system for commodity GPUs with microsecond-scale kernel preemption and controlled concurrent execution in GPU scheduling. REEF achieves both low-latency and work-conserving for concurrent real-time and best-effort tasks. 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 comprehensive microbenchmarks confirmed the efficacy and efficiency of REEF on both AMD and NVIDIA GPUs.

## References

- [1] Jacob T. Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. 2012. The case for GPGPU spatial multitasking. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA'12)*. IEEE Computer Society, USA, 1–12. DOI : <https://doi.org/10.1109/HPCA.2012.6168946>
- [2] Miguel Alcon, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. 2020. Timing of autonomous driving software: problem analysis and prospects for future solutions. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 267–280. DOI : <https://doi.org/10.1109/RTAS48715.2020.000-1>
- [3] AMD ROCm. 2022. AMD ROCm Platform Documentation. Retrieved from <https://rocm-docs.amd.com/>
- [4] Apollo Auto. 2022. Apollo: Architecture/Hardware Connection. Retrieved from <https://github.com/ApolloAuto/apollo>
- [5] Apollo Auto. 2022. Apollo Perception Module. Retrieved from <https://github.com/ApolloAuto/apollo/tree/master/modules/perception>
- [6] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. ONNX: Open Neural Network Exchange. Retrieved from <https://github.com/onnx/onnx>
- [7] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 499–514. Retrieved from <https://www.usenix.org/conference/osdi20/presentation/bai>
- [8] Baidu. 2022. Apollo. Retrieved from <https://apollo.auto/>
- [9] Joshua Bakita and James H. Anderson. 2023. Hardware compute partitioning on NVIDIA GPUs\*. *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS) (2023)*, 54–66. Retrieved from <https://api.semanticscholar.org/CorpusID:259235797>
- [10] C. Basaran and K. Kang. 2012. Supporting preemptive task executions and memory copies in GPGPUs. In *24th Euromicro Conference on Real-Time Systems (ECRTS'12)*. 287–296.
- [11] Karsten Behrendt, Libor Novak, and Rami Botros. 2017. A deep learning approach to traffic lights: Detection, tracking, and classification. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 1370–1377. DOI : <https://doi.org/10.1109/ICRA.2017.7989163>
- [12] Nicola Capodici, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. 2018. Deadline-Based Scheduling for GPU with Preemption Support. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. 119–130. DOI : <https://doi.org/10.1109/RTSS.2018.00021>
- [13] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. DOI : <https://doi.org/10.1109/IISWC.2009.5306797>
- [14] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. EffiSha: A software framework for enabling efficient preemptive scheduling of GPU. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'17)*. Association for Computing Machinery, Austin, Texas, USA, 3–16. DOI : <https://doi.org/10.1145/3018743.3018748>
- [15] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. Association for Computing Machinery, Xi'an, China, 17–32. DOI : <https://doi.org/10.1145/3037697.3037700>
- [16] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. Association for Computing Machinery, Atlanta, Georgia, USA, 681–696. DOI : <https://doi.org/10.1145/2872362.2872368>

- [17] 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. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*.
- [18] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan M. Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. arXiv:1410.0759. Retrieved from <https://arxiv.org/abs/1410.0759>
- [19] Green Car Congress. 2017. New ultrafast camera for self-driving vehicles and drones. Retrieved from <https://www.greencarcongress.com/2017/02/20170217-ntu.html>
- [20] Brian F. Cooper. 2022. YCSB Core Workloads. Retrieved from <https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads>
- [21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *1st ACM Symposium on Cloud Computing (SoCC'10)*. 143–154. DOI: <https://doi.org/10.1145/1807128.1807152>
- [22] NVIDIA Corporation. Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution. Retrieved from <https://github.com/triton-inference-server>. (n. d.).
- [23] Alexander Craik, Yongtian He, and José Luis Contreras-Vidal. 2019. Deep learning for electroencephalogram (EEG) classification tasks: A review. *Journal of Neural Engineering* 16, 3 (June 2019), 031001. DOI: <https://doi.org/10.1088/1741-2552/ab0ab5>
- [24] D. Crankshaw, Xin Wang, Giulio Zhou, M. Franklin, Joseph E. Gonzalez, and I. Stoica. 2017. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*.
- [25] Weihao Cui, Mengze Wei, Quan Chen, Xiaoxin Tang, Jingwen Leng, Li Li, and Mingyi Guo. 2019. Ebird: Elastic batch for improving responsiveness and throughput of deep learning services. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*. 497–505. DOI: <https://doi.org/10.1109/ICCD46524.2019.00075>
- [26] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Mingyi Guo. 2021. Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'21)*. Association for Computing Machinery, St. Louis, Missouri. DOI: <https://doi.org/10.1145/3458817.3476143>
- [27] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static analysis and compiler design for idempotent processing. In *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. 475–486. DOI: <https://doi.org/10.1145/2254064.2254120>
- [28] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805. Retrieved from <https://arxiv.org/abs/1810.04805>
- [29] ROCm documentation. 2022. GCN Native ISA LLVM Code Generator: Kernel Dispatch. Retrieved from [https://rocmdocs.amd.com/en/latest/ROCm\\_Compiler\\_SDK/ROCm-Native-ISA.html](https://rocmdocs.amd.com/en/latest/ROCm_Compiler_SDK/ROCm-Native-ISA.html)
- [30] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. 2019. A guide to deep learning in healthcare. *Nature medicine* 25, 1 (2019), 24–29.
- [31] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: an efficient GPU serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'21)*. Association for Computing Machinery, Virtual Event, Republic of Korea, 389–402. DOI: <https://doi.org/10.1145/3437801.3441578>
- [32] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. {ServerlessLLM}:{Low-Latency} serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*. 135–153.
- [33] Google. 2024. Gemma-2B-IT. Retrieved from <https://huggingface.co/google/gemma-2b-it>
- [34] Chris Gregg, Jonathan Dorn, K. Hazelwood, and K. Skadron. 2012. Fine-grained resource sharing for concurrent GPGPU kernels. In *4th USENIX Workshop on Hot Topics in Parallelism (HotPar'12)*.
- [35] A. Gujarati, Reza Karimi, Safya Alzayat, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*.
- [36] Mingcong Han, Weihang Shen Rong Chen, Binyu Zang, and Haibo Chen. 2025. Holistic Heterogeneous Scheduling for Autonomous Applications using Fine-grained, Multi-XPU Abstraction. (2025). arXiv:2508.09503. Retrieved from <https://arxiv.org/abs/2508.09503>
- [37] Mingcong Han, Weihang Shen, Guanwen Peng, Rong Chen, and Haibo Chen. 2024. Microsecond-scale Dynamic Validation of Idempotency for GPU Kernels. (2024). arXiv:2410.23661. Retrieved from <https://arxiv.org/abs/2410.23661>

- [38] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. Retrieved from <https://www.usenix.org/conference/osdi22/presentation/han>
- [39] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. 2015. DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*, Association for Computing Machinery, Portland, Oregon, 27–40. DOI : <https://doi.org/10.1145/2749469.2749472>
- [40] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (CVPR'16)*. 770–778. DOI : <https://doi.org/10.1109/CVPR.2016.90>
- [41] Jeremy Hermann and Mike Del Balso. 2017. Meet Michelangelo: Uber's Machine Learning Platform. Retrieved from <https://eng.uber.com/michelangelo-machine-learning-platform/>
- [42] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, et al. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine* 29, 6 (2012), 82–97.
- [43] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. 2019. GRNN: Low-latency and scalable RNN inference on GPUs. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys'19)*. Association for Computing Machinery, Dresden, Germany. DOI : <https://doi.org/10.1145/3302424.3303949>
- [44] Qingda Hu, Jiwu Shu, Jie Fan, and Youyou Lu. 2016. Run-time performance estimation and fairness-oriented scheduling policy for concurrent GPGPU applications. In *2016 45th International Conference on Parallel Processing (ICPP)*. 57–66. DOI : <https://doi.org/10.1109/ICPP.2016.14>
- [45] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. 2019. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 29–41. DOI : <https://doi.org/10.1109/RTAS.2019.00011>
- [46] Wonseok Jang, Hansaem Jeong, Kyungtae Kang, Nikil Dutt, and Jong-Chan Kim. 2020. R-TOD: Real-time object detector with minimized end-to-end delay for autonomous driving. In *2020 IEEE Real-Time Systems Symposium (RTSS)*. 191–204. DOI : <https://doi.org/10.1109/RTSS49844.2020.00027>
- [47] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. 2023. Fast and efficient model serving using multi-GPUs with direct-host-access. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys'23)*. Association for Computing Machinery, New York, NY, USA, 249–265. DOI : <https://doi.org/10.1145/3552326.3567508>
- [48] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. Association for Computing Machinery, Huntsville, Ontario, Canada, 47–62. DOI : <https://doi.org/10.1145/3341301.3359630>
- [49] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. 2023. DeepUM: Tensor migration and prefetching in unified memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 207–221. DOI : <https://doi.org/10.1145/3575693.3575736>
- [50] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Raj Rajkumar. 2011. RGEM: A Responsive GPGPU Execution Model for Runtime Engines. *2011 IEEE 32nd Real-Time Systems Symposium* (2011), 57–66. Retrieved from <https://api.semanticscholar.org/CorpusID:8258940>
- [51] Shinpei Kato, Karthik Lakshmanan, Ragunathan Raj Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX Annual Technical Conference*. Retrieved from <https://api.semanticscholar.org/CorpusID:18344830>
- [52] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott A. Brandt. 2012. Gdev: First-class GPU resource management in the operating system. In *USENIX Annual Technical Conference*. Retrieved from <https://api.semanticscholar.org/CorpusID:3200101>
- [53] Hyeonsu Lee, Hyunjun Kim, Cheolgi Kim, Hwansoo Han, and Euseong Seo. 2021. Idempotence-based preemptive GPU kernel scheduling for embedded systems. *IEEE Transactions on Computers* 70, 3 (2021), 332–346. DOI : <https://doi.org/10.1109/TC.2020.2988251>
- [54] TIMOTHY B. LEE. 2019. Tesla's autonomy event: Impressive progress with an unrealistic timeline. Retrieved from <https://arstechnica.com/cars/2019/04/teslas-autonomy-event-impressive-progress-with-an-unrealistic-timeline/>
- [55] Jingwen Leng, Alper Buyuktosunoglu, Ramon Bertran Monfort, Pradip Bose, Quan Chen, Minyi Guo, and Vijay Janapa Reddi. 2020. Asymmetric resilience: Exploiting task-level idempotency for transient error recovery in accelerator-based systems. *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA) (2020)*, 44–57.

- [56] LG Electronics Inc. 2022. Running Apollo 5.0 with SVL Simulator. Retrieved from <https://www.svlsimulator.com/docs/system-under-test/apollo5-0-instructions/>
- [57] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, et al. 2023. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*. USENIX Association, Boston, MA, 663–679. Retrieved from <https://www.usenix.org/conference/osdi23/presentation/li-zhouhan>
- [58] Yun Liang, Huynh Phung Huynh, Kyle Rupnow, Rick Siow Mong Goh, and Deming Chen. 2015. Efficient GPU spatial-temporal multitasking. *IEEE Trans. Parallel Distrib. Syst.* 26, 3 (March 2015), 748–760. DOI : <https://doi.org/10.1109/TPDS.2014.2313342>
- [59] Zhen Lin, Lars Nyland, and Huiyang Zhou. 2016. Enabling efficient preemption for SIMT architectures with lightweight context switching. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 898–908. DOI : <https://doi.org/10.1109/SC.2016.76>
- [60] LLVM. 2021. User Guide for AMDGPU Backend. Retrieved from <https://llvm.org/docs/AMDGPUUsage.html>
- [61] Justin Luitjens. CUDA Streams—Best Practices and Common Pitfalls. Retrieved from <http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf>. (n. d.).
- [62] Lingxiao Ma, Z. Xie, Zhi Yang, J. Xue, Youshan Miao, Wei Cui, W. Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 881–897.
- [63] Meta AI. 2024. Llama-3.2-3B. Retrieved from <https://huggingface.co/meta-llama/Llama-3.2-3B>. (2024). Hugging Face Model.
- [64] MLC team. 2023-2025. MLC-LLM. (2023-2025). Retrieved from <https://github.com/mlc-ai/mlc-llm>
- [65] Pavlo Molchanov, Shalini Gupta, Kihwan Kim, and Kari Pulli. 2015. Multi-sensor System for driver's hand-gesture recognition. *11th IEEE International Conference and Workshops on Automatic Face and Gesture Recognition 1* (2015), 1–8.
- [66] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. 2018. Accelerating deep learning workloads through efficient multi-model execution. In *NeurIPS Workshop on Systems for Machine Learning*. 20.
- [67] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency model serving with software-defined GPU scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*. Association for Computing Machinery, New York, NY, USA, 595–610. DOI : <https://doi.org/10.1145/3600006.3613163>
- [68] NVIDIA. NVIDIA TensorRT. Retrieved from <https://developer.nvidia.com/tensorrt>. (n. d.).
- [69] NVIDIA. NVIDIA/open-gpu-kernel-modules: NVIDIA Linux open GPU kernel module source. <https://github.com/NVIDIA/open-gpu-kernel-modules>. (n. d.). Retrieved from <https://github.com/NVIDIA/open-gpu-kernel-modules>
- [70] NVIDIA. 2016. NVIDIA Tesla P100. Retrieved from <http://www.nvidia.com/object/pascal-architecture-whitepaper.html>
- [71] NVIDIA. 2021. CUDA Toolkit: Develop, Optimize and Deploy GPU-Accelerated Apps. Retrieved from <https://developer.nvidia.com/cuda-toolkit>
- [72] NVIDIA. 2024. Blackwell Instruction Set. Retrieved from <https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#blackwell-blackwell-instruction-set>
- [73] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. 2017. TensorFlow-serving: Flexible, high-performance ML serving. In *Workshop on ML Systems at NIPS 2017*.
- [74] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. In *Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*.
- [75] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative preemption for multitasking on a shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. Association for Computing Machinery, Istanbul, Turkey, 593–606. DOI : <https://doi.org/10.1145/2694344.2694346>
- [76] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2017. Dynamic resource management for efficient utilization of multitasking GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. Association for Computing Machinery, Xi'an, China, 527–540. DOI : <https://doi.org/10.1145/3037697.3037707>
- [77] Adam Paszke, S. Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, N. Gimelshein, L. Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *33rd International Conference on Neural Information Processing Systems (NeurIPS'19)*.
- [78] Reid Pinkham, Andrew Berkovich, and Zhengya Zhang. 2021. Near-sensor distributed DNN processing for augmented and virtual reality. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 11, 4 (2021), 663–676. DOI : <https://doi.org/10.1109/JETCAS.2021.3121259>

- [79] Qwen Team. 2024. Qwen2.5-3B-Instruct. Retrieved from <https://huggingface.co/Qwen/Qwen2.5-3B-Instruct>. (2024). Hugging Face Model.
- [80] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. OpenAI.
- [81] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 779–788. DOI: <https://doi.org/10.1109/CVPR.2016.91>
- [82] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. (2018). Retrieved from <http://arxiv.org/abs/1804.02767> cite arxiv:1804.02767Comment: Tech Report.
- [83] Steve Rennich. CUDA C/C++ Streams and Concurrency. Retrieved from <https://developer.download.nvidia.cn/CUDA/training/StreamsAndConcurrencyWebinar.pdf>. (n. d.).
- [84] ROCm Core Technology. 2022. AMD GPU kernel driver with KFD. Retrieved from <https://github.com/RadeonOpenCompute/ROCK-Kernel-Driver>
- [85] ROCm Core Technology. 2022. AMD GPU kernel driver with KFD: unmap\_queues\_cpsch. Retrieved from [https://github.com/RadeonOpenCompute/ROCK-Kernel-Driver/blob/master/drivers/gpu/drm/amd/amdkfd/kfd\\_device\\_queue\\_manager.c](https://github.com/RadeonOpenCompute/ROCK-Kernel-Driver/blob/master/drivers/gpu/drm/amd/amdkfd/kfd_device_queue_manager.c)
- [86] ROCm Developer Tools. 2022. HIP: C++ Heterogeneous-Compute Interface for Portability. Retrieved from <https://github.com/ROCm-Developer-Tools/HIP>
- [87] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated model-less inference serving. In *USENIX Annual Technical Conference (ATC'21)*. 397–411.
- [88] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Márton Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, et al. 2020. LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving. In *IEEE 23rd International Conference on Intelligent Transportation Systems Conference (ITSC'20)*. 1–6. DOI: <https://doi.org/10.1109/ITSC45102.2020.9294422>
- [89] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)*. Association for Computing Machinery, New York, NY, USA, 233–248. DOI: <https://doi.org/10.1145/2043556.2043579>
- [90] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. *27th ACM Symposium on Operating Systems Principles* (2019).
- [91] Weihang Shen, Mingcong Han, Jialong Liu, Rong Chen, and Haibo Chen. 2025. XSched: Preemptive scheduling for diverse XPU. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 671–692. Retrieved from <https://www.usenix.org/conference/osdi25/presentation/shen-weihang>
- [92] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. 2023. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 701–718.
- [93] Sudipta Saha Shubha, Haiying Shen, and Anand Iyer. 2024. USHER: Holistic interference avoidance for resource optimized ML inference. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 947–964. Retrieved from <https://www.usenix.org/conference/osdi24/presentation/shubha>
- [94] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556. Retrieved from <https://arxiv.org/abs/1409.1556>
- [95] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, fine-grained GPU sharing for ML applications. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys'24)*. Association for Computing Machinery, New York, NY, USA, 1075–1092. DOI: <https://doi.org/10.1145/3627703.3629578>
- [96] I. Tanasić, Isaac Gelado, Javier Cabezas, A. Ramírez, N. Navarro, and M. Valero. 2014. Enabling preemptive multiprogramming on GPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA'14)*. IEEE Press, Minneapolis, Minnesota, USA, 193–204.
- [97] TESLARATI. 2021. AMD confirms Tesla's new Model S and Model X will boast RDNA 2 GPUs. Retrieved from <https://www.teslarati.com/tesla-model-s-model-x-mcu3-specs-amd-gpu-confirmed-video/>
- [98] Apache TVM. 2021. Apache TVM: An End to End Machine Learning Compiler Framework for CPUs, GPUs and accelerators. Retrieved from <https://tvm.apache.org/>
- [99] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*. Association for Computing Machinery, Columbus, OH, USA, 372–383. DOI: <https://doi.org/10.1145/3352460.3358307>

- [100] Guibin Wang, YiSong Lin, and Wei Yi. 2010. Kernel Fusion: An effective method for better power efficiency on multithreaded GPU. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing (GREENCOM-CPSCOM'10)*. IEEE Computer Society, USA, 344–350. DOI : <https://doi.org/10.1109/GreenCom-CPSCOM.2010.102>
- [101] Jiali Wang, Yankui Wang, Mingcong Han, and Rong Chen. 2025. Colocating ML inference and training with fast GPU memory handover. In *USENIX Annual Technical Conference (ATC'25)*.
- [102] Yuxuan Wang, R. J. Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J. Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, Samy Bengio, et al. 2017. Tacotron: A fully end-to-end text-to-speech synthesis model. arXiv:1703.10135. Retrieved from <https://arxiv.org/abs/1703.10135>
- [103] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 358–369. DOI : <https://doi.org/10.1109/HPCA.2016.7446078>
- [104] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. 2017. FLEP: Enabling flexible and efficient preemption on GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. Association for Computing Machinery, Xi'an, China, 483–496. DOI : <https://doi.org/10.1145/3037697.3037742>
- [105] Yecheng Xiang and Hyoseung Kim. 2019. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. 392–405. DOI : <https://doi.org/10.1109/RTSS46320.2019.00042>
- [106] Feng Yan, Yuxiong He, Olatunji Ruwase, and Evgenia Smirni. 2018. Efficient deep neural network serving: Fast and furious. *IEEE Transactions on Network and Service Management* 15, 1 (2018), 112–126. DOI : <https://doi.org/10.1109/TNSM.2018.2808352>
- [107] Jinrong Yang, Zimeng Wang, Rong Chen, and Haibo Chen. 2025. A system-level abstraction and service for flourishing AI-powered applications. In *Proceedings of the 16th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'25)*. Association for Computing Machinery, New York, NY, USA, 9. DOI : <https://doi.org/10.1145/3725783.3764406>
- [108] Ming Yang, Shige Wang, Joshua Bakita, Thanh Vu, F. Donelson Smith, James H. Anderson, and Jan-Michael Frahm. 2019. Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 305–317. DOI : <https://doi.org/10.1109/RTAS.2019.00033>
- [109] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: A native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 768–781.
- [110] Wai Chee Yau. 2017. How Zendesk Serves TensorFlow Models in Production. Retrieved from <https://zendesk.engineering/how-zendesk-serves-tensorflow-models-in-production-751ee22f0f4b>
- [111] Tsung Tai Yeh, Matthew D. Sinclair, Bradford M. Beckmann, and Timothy G. Rogers. 2021. Deadline-aware offloading for high-throughput accelerators. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 479–492. DOI : <https://doi.org/10.1109/HPCA51647.2021.00048>
- [112] Juheon Yi and Youngki Lee. 2020. Heimdall: mobile GPU coordination platform for augmented reality applications. In *26th Annual International Conference on Mobile Computing and Networking (MobiCom'20)*. 1–14.
- [113] Sebastian Zepf, Javier Hernandez, Alexander Schmitt, Wolfgang Minker, and Rosalind W. Picard. 2020. Driver emotion recognition for intelligent vehicles: A survey. *ACM Comput. Surv.* 53, 3 (July 2020). DOI : <https://doi.org/10.1145/3388790>
- [114] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In *Symposium on Networked Systems Design and Implementation*. Retrieved from <https://api.semanticscholar.org/CorpusID:258559393>
- [115] Chen Zhao, Wu Gao, Feiping Nie, and Huiyang Zhou. 2022. A survey of GPU multitasking methods supported by hardware architecture. *IEEE Transactions on Parallel and Distributed Systems* 33 (2022), 1451–1463. Retrieved from <https://api.semanticscholar.org/CorpusID:239197299>
- [116] Hengyu Zhao, Yubo Zhang, Pingfan Meng, Hui Shi, Erran L. Li, Tiancheng Lou, and Jishen Zhao. 2019. Towards safety-aware computing system design in autonomous vehicles. arXiv:1905.08453. Retrieved from <https://arxiv.org/abs/1905.08453>
- [117] Wenyi Zhao, Quan Chen, and Minyi Guo. 2018. KSM: Online application-level performance slowdown prediction for spatial multitasking GPGPU. *IEEE Comput. Archit. Lett.* 17, 2 (July 2018), 187–191. DOI : <https://doi.org/10.1109/LCA.2018.2851207>
- [118] Wenyi Zhao, Quan Chen, Hao Lin, Jianfeng Zhang, Jingwen Leng, Chao Li, Wenli Zheng, Li Li, and Minyi Guo. 2019. Themis: Predicting and reining in application-level slowdown on spatial multitasking GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 653–663. DOI : <https://doi.org/10.1109/IPDPS.2019.00074>

- [119] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. HSM: A hybrid slowdown model for multitasking GPUs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. Association for Computing Machinery, Lausanne, Switzerland, 1371–1385. DOI: <https://doi.org/10.1145/3373376.3378457>
- [120] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*.
- [121] Husheng Zhou, Soroush Bateni, and Cong Liu. 2018. S3DNN: Supervised streaming and scheduling for GPU-accelerated real-time DNN workloads. *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (2018)*, 190–201.
- [122] Husheng Zhou, Guangmo Tong, and Cong Liu. 2015. GPES: A preemptive execution system for GPGPU computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 87–97. DOI: <https://doi.org/10.1109/RTAS.2015.7108420>
- [123] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. 2022. {ROLLER}: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 233–248.

Received 1 November 2024; revised 30 May 2025; accepted 5 September 2025